Modelling Software-based Systems

Lecture 2

Correctness by Construction with the Modelling Language Event-B using the Refinement

Master Informatique

Dominique Méry
Telecom Nancy,Université de Lorraine

18 septembre 2024

dominique.mery@loria.fr

General Summary

Documents for lectures

Link for the course

https://arche.univ-lorraine.fr/course/view.php?id=12474

- password : mery2020

Documents

- Slides of lectures
- Tutorials
- Event-B archives related to tutorials.

Current Summary

Correctness by Construction

- Correctness by Construction is a method of building software -based systems with **demonstrable correctness** for security- and safety-critical applications.
- Correctness by Construction advocates a **step-wise refinement** process from specification to code using tools for checking and transforming models.
- Correctness by Construction is an approach to software/system construction
  - starting with an abstract model of the problem.
  - progressively adding details in a step-wise and checked fashion.
  - each step guarantees and proves the correctness of the new concrete model with respect to requirements

The Cleanroom Method as CbC

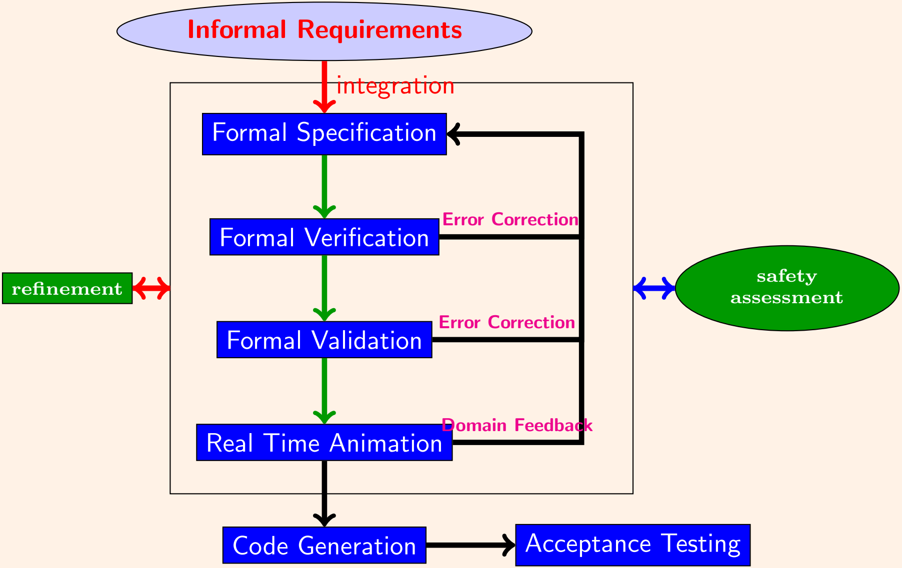
- The **Cleanroom** method, developed by Harlan Mills and his colleagues at IBM and elsewhere, attempts to do for software what cleanroom fabrication does for semiconductors : to achieve quality by keeping defects out during fabrication.
- In semiconductors, **dirt** or **dust** that is allowed to **contaminate** a chip as it is being made cannot possibly be removed later.
- But we try to do the equivalent when we write programs that are full of bugs, and then attempt to remove them all using debugging.

The Cleanroom Method as CbC

The Cleanroom method, then, uses a number of techniques to develop software carefully, in a well-controlled way, so as to avoid or eliminate as many defects as possible before the software is ever executed. Elements of the method are :

- specification of all components of the software at all levels ;
- stepwise refinement using constructs called "box structures" ;
- verification of all components by the development team ;
- statistical quality control by independent certification testing ;
- no unit testing, no execution at all prior to certification testing.

Critical System Development Life-Cycle Methodology



Overview of Methodology

- Informal Requirements : Restricted form of natural language.
- Formal Specification : Modeling language like Event-B , Z, ASM, VDM, TLA+...
- Formal Verification : Theorem Prover Tools like PVS, Z3, SAT, SMT Solver...
- Formal Validation : Model Checker Tools like ProB, UPPAAL , SPIN, SMV ...
- Real-time Animation : **Our proposed approach ... Real-Time Animator ...**
- Code Generation : **Our proposed approach ... EB2ALL : EB2C, EB2C++, EB2J, EB2C# ...**
- Acceptance Testing : Failure Mode, Effects and Critically analysis(FMEA and FMEA), System Hazard Analyses(SHA)

Case Studies

- *Colin Boyd and Anish Mathuria. Protocols Authentication and Key Establisment. Springer 2003.*
- *C. C. Marquezan and L. Z. Granville. Self-* and P2P for Network Management - Design Principles and Case Studies. Springer Briefs in Computer Science. Springer, 2012.*
- *Pacemaker Challenge Contribution*

Current Summary

Problems for Modelling systems

- Systems are generally very complex
- Invariant should be strong enough for proving safety properties
- Problems for modelling : finding suitable mathematical structures, listing events or actions of the system, proving proof obligations, ...

Solution : refining models

- To understand more and more the system
- To distribute the complexity of the system
- To distribute the difficulties of the proof
- To improve explanations
- Validation (step by step)
- Refinement (invariant & behavior)

Refinement of models

- we can add more details (like superposition),
- we can add new events (we can observe more transformations),
- we prove that the concrete behaviors are abstract ones
  $\leadsto$ we got the abstract invariant for free.
- each new event refines SKIP
- no deadlock
- abstract events occur (new events decrease something)

Refinement of a model by another one

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\mathbf{I(X)} & & \mathbf{I(X')} \\
\mathbf{X} \arrow[rr, "\mathbf{abstract\_ev(X,X')}"] \arrow[d, dash, dashed] & & \mathbf{X'} \arrow[d, dash, dashed] \\
\mathbf{Y} \arrow[rr, "\mathbf{concrete\_ev(Y,Y')}"] & & \mathbf{Y'} \\
\mathbf{J(X,Y)} & & \mathbf{J(X',Y')}
\end{tikzcd}
$$

Refinement of a model by another one (I)

$$
\begin{tikzcd}
I(x) & x \arrow[r, "ae(x{,}x')"] \arrow[d, leftrightarrow] & x' \arrow[d, leftrightarrow] & I(x') \\
J(x,y) & y \arrow[r, "ce(y{,}y')"] & y' & J(x',y')
\end{tikzcd}
$$

$$
\begin{tikzcd}
I(x) & AM \arrow[r, "\text{SEES}"] & AC & \mathcal{T}h_i \\
J(x,y) & CM \arrow[u, "\text{REFINES}"] \arrow[r, "\text{SEES}"] & CC \arrow[u, "\text{EXTENDS}"'] & \mathcal{T}h_{i+1}
\end{tikzcd}
$$

1

Refinement of a model by another one (II)

$$
\begin{tikzcd}
I(x) & x \arrow[r, "x=x'"] \arrow[d, leftrightarrow] & x' \arrow[d, leftrightarrow] & I(x') \\
J(x,y) & y \arrow[r, "ce(y,y')"] & y' & J(x',y')
\end{tikzcd}
$$

$$
\begin{tikzcd}
I(x) & & x & & I(x) \\
J(x,y) & y \arrow[ur] \arrow[rr, "ce(y,y')"] & & y' \arrow[ul] & J(x,y')
\end{tikzcd}
$$

Proof obligations for refinement

(REF1) : refinement of initial conditions

$$\text{INITC}(y) \Rightarrow \exists x * (\text{INIT}(x) \quad \wedge \quad \text{J}(x,y)) \quad :$$

The initial condition of the refinement model imply that there exists an abstract value in the abstract model such that that value satisfies the initial conditions of the abstract one and implies the new invariant of the refinement model.

Proof obligations for refinement

(REF2) : refinement of events

$$\mathrm{I}(x) \;\wedge\; \mathrm{J}(x,y) \;\wedge\; ce(y,y') \Rightarrow \exists x'.(ae(x,x') \;\wedge\; \mathrm{J}(x',y')) \;:$$

The invariant in the refinement model is preserved by the refined event and the activation of the refined event triggers the corresponding abstract event.

Proof obligations for refinement

(REF3) : refinement of stuttering steps

$$\mathrm{I}(x) \ \wedge \ \mathrm{J}(x,y) \ \wedge \ ce(y,y') \Rightarrow \mathrm{J}(x,y') \quad :$$

The invariant in the refinement model is preserved by the refined event but the event of the refinement model is a new event which was not visible in the abstract model ; the new event refines $skip$.

Proof obligations for refinement

(REF4) : Refinement does not introduce more blocking states

$$\mathrm{I}(x) \;\wedge\; \mathrm{J}(x,y) \;\wedge\; (G_1(x) \vee \ldots \vee G_n(x)) \;\Rightarrow\; H_1(y) \vee \ldots \vee H_k(y) \quad :$$

The guards of events in the refinement model are strengthened and we have to prove that the refinement model is not more blocked than the abstract.

Proof obligations for refinement

(REF5) : Well-definedness of variant

$$\mathrm{I}(x) \;\; \wedge \;\; \mathrm{J}(x,y)) \;\Rightarrow\; V(y) \in \mathbb{N}$$

Proof obligations for refinement

(REF6) : Well behaviour of new events

$$\mathrm{I}(x) \;\wedge\; \mathrm{J}(x,y) \;\wedge\; ce(y,y') \;\Rightarrow\; V(y') < V(y) \;:$$

New events should not block forever abstract ones.

Proof obligations for refinement

(REF7) : Feasibility of refined events

$$\Gamma(s,c) \;\vdash\; I(x) \;\wedge\; J(x,y) \;\wedge\; grd\,(E) \;\Rightarrow\; \exists y' \cdot P(y,y')$$

Current Summary

The factorial model

CONTEXT
  $fonctions$
CONSTANTS $factorial, n$
AXIOMS

$$n \in \mathbb{N} \ \wedge \ factorial \in \mathbb{N} \ \leftrightarrow \ \mathbb{N} \ \wedge \ 0 \mapsto 1 \in factorial \ \wedge$$

$$\forall (i, fn).(i \mapsto fn \in factorial \Rightarrow i+1 \mapsto (i+1) * fi \in factorial) \ \wedge$$

$$\forall f \cdot \left(\begin{array}{l} f \in \mathbb{N} \ \leftrightarrow \ \mathbb{N} \wedge \\ 0 \ \mapsto \ 1 \ \in \ f \ \wedge \\ \forall (n, fn).(n \mapsto fn \in f \Rightarrow n+1 \mapsto (n+1) \times fn \in f) \\ \Longrightarrow \\ factorial \subseteq f \end{array} \right)$$

END

The factorial model

MACHINE
  $specification$
SEES $fonctions$
VARIABLES
  $resultat$
INVARIANT
  $resultat \in \mathbb{N}$
THEOREMS
  $factorial \in \mathbb{N} \longrightarrow \mathbb{N}$;
  $factorial(0) = 1$;
  $\forall n.(n \in \mathbb{N} \Rightarrow factorial(n+1) = (n+1) \times factorial(n))$
INITIALISATION
  $resultat :\in \mathbb{N}$
EVENTS
  $computing1 =$ BEGIN $resultat := factorial(n)$ END
END

Refining $specification$ by $computation$

MACHINE $computation$
REFINES $specification$
SEES $fonctions$
VARIABLES $resultat, fac, x$
INVARIANTS
$inv1 : fac \in \mathbb{N} \nrightarrow \mathbb{N}$
$inv2 : dom(fac) \subseteq 0 \,..\, n$
$inv4 : dom(fac) \neq \varnothing$
$inv5 : \forall i.i \in dom(fac) \Rightarrow fac(i) = factorial(i)$
$inv3 : x \in dom(fac)$
$inv6 : dom(fac) = 0 \,..\, x$
EVENTS
EVENT INITIALISATION
BEGIN
$act1 : resultat :\in \mathbb{N}$
$act2 : fac := \{0 \mapsto 1\}$
$act3 : x := 0$
END
EVENT computing2 REFINES EVENT computing1
WHEN
$grd1 : n \in dom(fac)$
THEN
$act1 : resultat := fac(n)$
END
END

Refining $specification$ by $computation$

```
MACHINE computation
REFINES specification
SEES fonctions
VARIABLES resultat, fac, x
INVARIANTS
  inv1 : fac ∈ ℕ ⇸ ℕ
  inv2 : dom(fac) ⊆ 0 .. n
  inv4 : dom(fac) ≠ ∅
  inv5 : ∀i.i ∈ dom(fac) ⇒ fac(i) = factorial(i)
  inv3 : x ∈ dom(fac)
  inv6 : dom(fac) = 0 .. x
EVENTS
EVENT event2
  WHEN
    grd11 : x ∈ dom(fac)
    grd12 : x + 1 ∉ dom(fac)
    grd13 : n ∉ dom(fac)
  THEN
    act11 : fac(x + 1) := (x + 1) ∗ fac(x)
    act1 : x := x + 1
  END
END
```

Refining $computation$ by $algorithm$

EVENTS
EVENT INITIALISATION
  BEGIN
    $act1 : resultat :\in \mathbb{N}$
    $act2 : fac := \{0 \mapsto 1\}$
    $act3 : cfac := 0$
    $act4 : vfac := 1$
    $act5 : x := 0$
  END
EVENT computing3 REFINES EVENT computing2
  WHEN $grd2 : cfac = n$
  THEN $act1 : resultat := vfac$
  END
EVENT event3 REFINES EVENT event2
  WHEN $grd1 : cfac \neq n$
  THEN
    $act1 : vfac := (cfac + 1) * vfac$
    $act2 : cfac := cfac + 1$
    $act3 : fac(cfac + 1) := (cfac + 1) * fac(cfac)$
    $act4 : x := x + 1$
  END
END

Refining *computation* by *algorithm*

MACHINE $algorithm$ **REFINES** $computation$
SEES $fonctions$
VARIABLES $resultat, vfac, \quad cfac, fac, x$
INVARIANTS

$inv1 : vfac \in \mathbb{N}$
$inv2 : cfac \in \mathbb{N}$
$inv3 : cfac \leq n$
$inv4 : cfac \geq 0$
$inv6 : cfac \in dom(fac)$
$inv5 : vfac = fac(cfac)$
$inv7 : cfac + 1 \notin dom(fac)$
$inv8 : dom(fac) = 0 \,..\, cfac$
$inv9 : x = cfac$

Refining $algorithm$ by $simplealgorithm$

MACHINE $simplealgorithm$ REFINES $algorithm$
SEES $fonctions$
VARIABLES $resultat, vfac, cfac$
THEOREMS $thm1 : vfac = factorial(cfac)$
EVENTS
EVENT INITIALISATION
  BEGIN
    $act1 : resultat :\in \mathbb{N}$
    $act3 : cfac := 0$
    $act4 : vfac := 1$
  END
EVENT computing4 REFINES EVENT computing3
  WHEN
    $grd2 : cfac = n$
  THEN
    $act1 : resultat := vfac$
  END
EVENT event4 REFINES EVENT event3
  WHEN
    $grd1 : cfac \neq n$
  THEN
    $act1 : vfac := (cfac + 1) * vfac$
    $act2 : cfac := cfac + 1$
  END
END

Current Summary

Current Summary

Simple Form of an Event

- An event of the simple form is denoted by :

$$\begin{array}{l} < event\_name > \ \widehat{=} \\ \quad \text{WHEN} \\ \qquad < condition > \\ \quad \text{THEN} \\ \qquad < action > \\ \quad \text{END} \end{array}$$

where
- $< event\_name >$ is an identifier
- $< condition >$ is the firing condition of the event
- $< action >$ is a generalized substitution (parallel "assignment")

Non-deterministic Form of an Event

- An event of the non-deterministic form is denoted by :

$$
\begin{array}{l}
< event\_name > \ \widehat{=} \\
\quad \text{ANY} \ < variable > \ \text{WHERE} \\
\quad\quad < condition > \\
\quad \text{THEN} \\
\quad\quad < action > \\
\quad \text{END}
\end{array}
$$

where
- $< event\_name >$ is an identifier
- $< variable >$ is a (list of) variable(s)
- $< condition >$ is the firing condition of the event
- $< action >$ is a generalized substitution (parallel "assignment")

Shape of a Generalized Substitution

A generalized substitution can be

- Simple assignment : $x := E$
- Generalized assignment : $x : |P(x, x')$
- Set assignment : $x :\in S$

'
- Parallel composition : $\begin{array}{c} T \\ \cdots \\ U \end{array}$

Invariant Preservation Verification (0)

$$\begin{array}{l}\text{INVARIANT} \quad \wedge \quad \text{GUARD} \\ \Longrightarrow \\ \text{ACTION } \textcolor{red}{\text{establishes}} \text{ INVARIANT}\end{array}$$

Invariant Preservation Verification (1)

- Given an event of the simple form :

$$
\begin{array}{l}
\text{EVENT } \mathsf{e} \ \widehat{=} \\
\quad \text{WHEN} \\
\qquad G(x) \\
\quad \text{THEN} \\
\qquad x := E(x) \\
\quad \text{END}
\end{array}
$$

and invariant $I(x)$ to be preserved, the statement to prove is :

$$
\boxed{I(x) \ \wedge \ G(x) \ \Longrightarrow \ I(E(x))}
$$

Invariant Preservation Verification (2)

- Given an event of the simple form :

$$
\begin{array}{l}
\textsf{EVENT } \mathsf{e} \;\widehat{=} \\
\quad \textsf{WHEN} \\
\quad\quad G(x) \\
\quad \textsf{THEN} \\
\quad\quad x :| P(x, x') \\
\quad \textsf{END}
\end{array}
$$

and invariant $I(x)$ to be preserved, the statement to prove is :

$$\boxed{I(x) \;\wedge\; G(x) \;\wedge\; P(x, x') \;\Longrightarrow\; I(x')}$$

Invariant Preservation Verification (3)

- Given an event of the simple form :

```
EVENT e ≙
  WHEN
    G(x)
  THEN
    x :∈ S(x)
  END
```

and invariant $I(x)$ to be preserved, the statement to prove is :

$$\boxed{I(x) \;\wedge\; G(x) \;\wedge\; x' \in S(x) \;\Longrightarrow\; I(x')}$$

Invariant Preservation Verification (4)

- Given an event of the non-deterministic form :

$$
\begin{array}{l}
\text{EVENT e} \;\widehat{=} \\
\quad \text{ANY } v \text{ WHERE} \\
\qquad G(x,v) \\
\quad \text{THEN} \\
\qquad x := E(x,v) \\
\quad \text{END}
\end{array}
$$

and invariant $I(x)$ to be preserved, the statement to prove is :

$$\boxed{I(x) \;\wedge\; G(x,v) \;\Longrightarrow\; I(E(x,v))}$$

Refinement Technique (1)

- Abstract models works with variables x, and concrete one with y
- A gluing invariant $J(x, y)$ links both sets of vrbls
- Each abstract event is refined by concrete one (see below)

Refinement Technique (2)

- Some new events may appear : they refine “skip”
- Concrete events must not block more often than the abstract ones
- The set of new event alone must always block eventually

Correct Refinement Verification (1)

- Given an abstract and a corresponding concrete event

EVENT ae $\widehat{=}$ WHEN $G(x)$ THEN $x := E(x)$ END	EVENT ce $\widehat{=}$ WHEN $H(y)$ THEN $y := F(y)$ END

and invariants $I(x)$ and $J(x,y)$, the statement to prove is :

$$I(x) \;\wedge\; J(x,y) \;\wedge\; H(y) \Longrightarrow G(x) \;\wedge\; J(E(x),F(y))$$

Correct Refinement Verification (2)

- Given an abstract and a corresponding concrete event

EVENT ae $\widehat{=}$ ANY v WHERE $G(x,v)$ THEN $x := E(x,v)$ END	EVENT ce $\widehat{=}$ ANY w WHERE $H(y,w)$ THEN $y := F(y,w)$ END

$$
\begin{array}{l}
I(x) \quad \wedge \quad J(x,y) \quad \wedge \quad H(y,w) \\
\Longrightarrow \\
\exists v \cdot (\, G(x,v) \quad \wedge \quad J(E(x,v), F(y,w)) \,)
\end{array}
$$

Correct Refinement Verification (3)

- Given a new event

$$\begin{array}{l}\text{EVENT new\_e} \quad \widehat{=} \\ \quad \text{WHEN} \\ \quad\quad H(y) \\ \quad \text{THEN} \\ \quad\quad y := F(y) \\ \quad \text{END}\end{array}$$

and invariants $I(x)$ and $J(x,y)$, the statement to prove is :

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; H(y) \;\Longrightarrow\; J(x,F(y))}$$

Current Summary

WHEN P Q THEN S END	WHEN P $\neg Q$ THEN T END	*are merged into*	WHEN P THEN WHILE Q DO S END; T END

```
WHEN
  P
  Q
THEN
  S
END
```

```
WHEN
  P
  ¬Q
THEN
  T
END
```

are merged into

```
WHEN
  P
THEN
  WHILE  Q  DO
    S
  END;
  T
END
```

Side Conditions :

- P must be invariant under S.
- The first event must have been introduced at one refinement step below the second one.
- Special Case : If P is missing the resulting "event" has no guard

algorithms by J.-R. Abrial

WHEN P Q THEN S END	WHEN P $\neg Q$ THEN T END	*are merged into*	WHEN P THEN IF Q THEN S ELSE T END; END

Side Conditions :

- The disjunctive negation of the previous side conditions
- Special Case : If P is missing the resulting "event" has no guard

Applying the rule for the while

EVENT computing4 REFINES EVENT computing3
  WHEN
    $grd2 : cfac = n$
  THEN
    $act1 : resultat := vfac$
  END

EVENT event4 REFINES EVENT event3
  WHEN
    $grd1 : cfac \neq n$
  THEN
    $act1 : vfac := (cfac + 1) * vfac$
    $act2 : cfac := cfac + 1$
  END
END

EVENT computing4   EVENT event4
  WHILE $cfac \neq n$ DO
    $act1 : vfac := (cfac + 1) * vfac$
    $act2 : cfac := cfac + 1$
  END;
END

Applying the INITIALISATION rule

EVENT INITIALISATION
BEGIN
$act1 : resultat :\in \mathbb{N}$
$act3 : cfac := 0$
$act4 : vfac := 1$
END

$init$
$resultat :\in \mathbb{N};$
$cfac := 0;$
$vfac := 1;$

Deriving an algorithm

precondition : $n \in \mathbb{N}$
postcondition : $result = factorial(n)$
local variables : $vfac, cfac \in \mathbb{N}$

$cfac := 0; vfac := 1; result :\in \mathbb{N};$
while $cfac \neq n$ **do**
  Invariant : $vfac = fac(cfac)$
  $vfac := (cfac + 1) * vfac; cfac := cfac + 1;$
;
$result := vfac;$

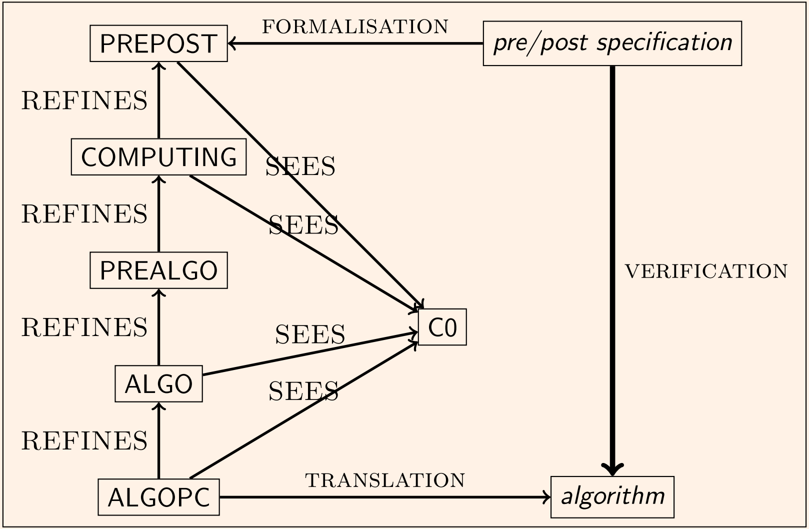
Current Summary

Comments

- Refinement helps in discovering invariants
- Refinement helps in proving invariants
- The choice of the *good* abstraction is not very simple and is a challenge by itself

Current Summary

The Iterative Pattern



General context C0

$\textsf{CONTEXT}$ $C0$
$\textsf{SETS}$
U
$\textsf{CONSTANTS}$
$x, v, d0, f, D$
$\textsf{AXIOMS}$
$axm1 : x \in \mathbb{N}$
$axm25 : D \subseteq U$
$axm24 : f \in D \to D$
$axm23 : d0 \in D$
$axm2 : v \in \mathbb{N} \to D$
$axm3 : v(0) = d0$
$axm4 : \forall n \cdot n \in \mathbb{N} \Rightarrow v(n+1) = f(v(n))$
$th1 : Q(d_0, d) \equiv (d = v(x))$

- the sequence v expresses the post-condition $Q(d_0, d)$ with the precondition $P(d_0)$.
- $Q(d_0, d)$ is equivalent to $d = v(x)$.
- The theorem $th1$ should be proved in the context C0. he

General PREPOST Machine

MACHINE $PREPOST$
SEES $C0$
VARIABLES
r
INVARIANTS
$inv1 : r \in D$
EVENTS
INITIALISATION
BEGIN
$act1 : r :\in D$
END
EVENT computing
BEGIN
$act1 : r := v(x)$
END
END

- The theorem $th1$ is validating the definition of the result r to compute.
- The event computing is expressing the *contract* of the given problem.
- it by a very simple problem that is the computation of the function n^2 using the addition operator.

First Refinement COMPUTING : Inductive Computation

EVENT INITIALISATION
  BEGIN
    $act1 : r :\in D$
    $act3 : vv := \{0 \mapsto d0\}$
    $act5 : k := 0$
  END

INITIALISATION is initializing the variables with respect to the initial values of the sequences of the context.

First Refinement COMPUTING : Inductive Computation

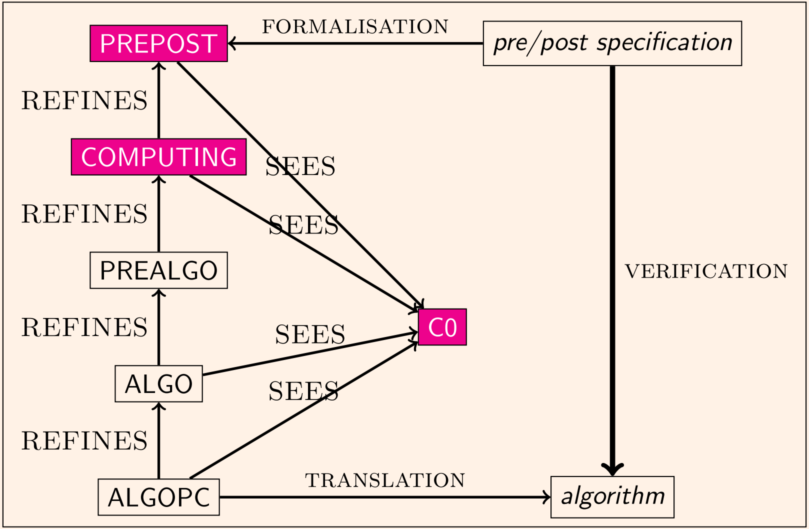
EVENT computing
$\quad \mathrm{REFINES}$ computing,
$\quad \mathrm{WHEN}$
$\quad\quad grd1 : x \in dom(vv)$
$\quad \mathrm{THEN}$
$\quad\quad act1 : r := vv(x)$
$\quad \mathrm{END}$
END

computing is imply observing that the result is computed *simulating* the sequence vv.

EVENT step
WHEN
$grd1 : x \notin dom(vv)$
THEN
$act2 : vv(k+1) := f(vv(k))$
$act4 : k := k+1$
END

step is *simulating* the computation of the values of the sequence vv as a model computation.

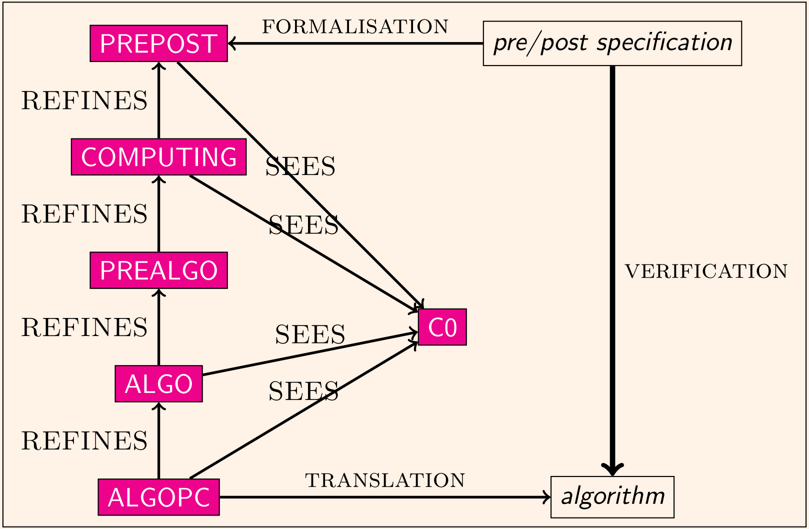
The Iterative Pattern



Completing the machines

- PREALGO : adding new variables for pointing out the necessary values to store cvv
- ALGO : hiding the model variables storing the unnecessary values of sequence vv
- ALGOPC ; adding control variable c

The Iterative Pattern



Listing 1 – Function derived from pattern for the sequence v

```
type(D)  f(int x)
{int  r,k,cv,or,ok,ocv;
  r=0;k=0;cv=0;or=0;ok=k;ocv=cv;
  while (k<x)
        {
          ok=k;ocv=cv;
          k=ok+1;
          cv=f(ocv);
        }
  r=cv;  return(r);}
```

Comments

- The produced algorithm can be now checked using another proof environment as for instance Frama-C.
- The inductive property of the invariant is clearly verified and is easily derived from the Event-B machines.
- The verification is not required, since the system is correct by construction but it is a checking of the process itself
- the project called ITERATIVE-PATTERN ;
- the project is the pattern itself
- The invariants of the Event-B models can be reused in the verification using Frama-C, for instance, and the verification of the resulting algorithm is a confirmation of the translation.

Translation of Event-B Models

Listing 2 – Function derived from pattern power3

```
#include <limits.h>
/*@ requires 0 <= x;
    requires x*x*x <= INT_MAX ;
    ensures \result ==x*x*x;
*/
int power3(int x)
{int r,ocz,cz,cv,cu,ocv,cw,ocw,ct,oct,ocu,k,ok;
  cz=0;cv=0;cw=1;ct=3;cu=0; ocw=cw;ocz=cz;
  oct=ct;ocv=cv;ocu=cu;k=0;ok=k;
       /*@ loop invariant cz == k*k*k;
         @ loop invariant cu  == k;
         @ loop invariant cv+ct==3*(cu+1)*(cu+1);
         @ loop invariant cz+cv+cw==3*(cu+1)*(cu+1)*(cu+1);
         @ loop invariant cv== 3*cu*cu;
         @ loop invariant cw == 3*cu+1;
         @ loop invariant k  <= x;
         @ loop assigns ct,oct,cu,ocu,cz,ocz,k,cv,cw,r,ok;
         @ loop assigns ocv,ocw;*/
  while (k<x)
        {
          ocz=cz;ok=k;ocv=cv;ocw=cw;oct=ct;ocu=cu;
          cz=ocz+ocv+ocw;
          cv=ocv+oct;
          ct=oct+6;
          cw=ocw+3;
          cu=ocu+1;
          k=ok+1;}
  r=cz;return(r);}
```

Translation of Event-B Models

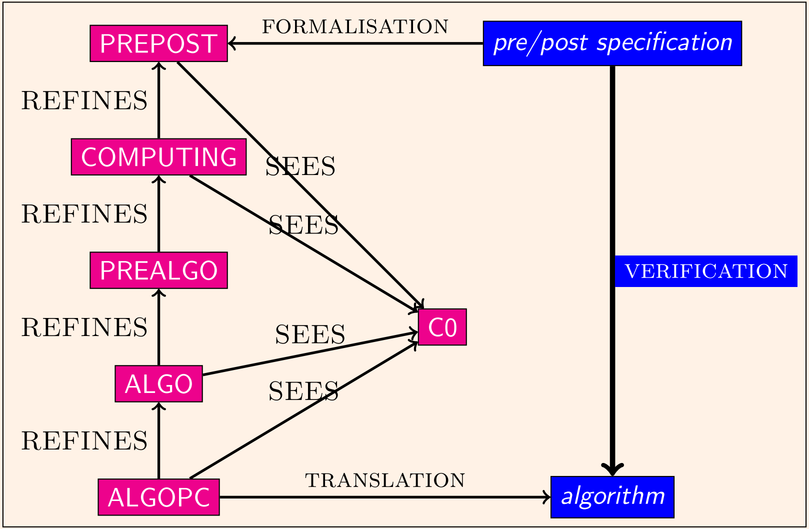
Summary for proof obligations

Name	Total	Automatic	Interactive
ex-induction	40	36	4
C0	2	0	2
PREPOST	4	4	0
COMPUTING	16	14	2
PREALGO	9	9	0
ALGO	6	6	0
ALGOPC	3	3	0

Summary

- The loop invariant is inductive but Frama-C does not prove it completely.
- Not the case with the RODIN platform which is able to discharge the whole set of proof obligations.
- However, the Event-B model is using auxiliary knowledge over sequences used for defining the computing process.
- The most difficult theorem is to prove that $\forall n \in \mathbb{N} : z_n = n * n * n$.

The Iterative Pattern



Current Summary

Conclusion

- Refinement helps in discovering invariants
- Refinement helps in proving invariants
- The choice of the *good* abstraction is not very simple and is a challenge by itself
- Patterns may help in applying refinement in a systematic way
- Summary on proof obligations