Modelling Software-based Systems

Lecture 5

The Call-as-Event Paradigm

Telecom Nancy (IL et LE)

Dominique Méry
Telecom Nancy,Université de Lorraine

23 novembre 2026

dominique.mery@loria.fr

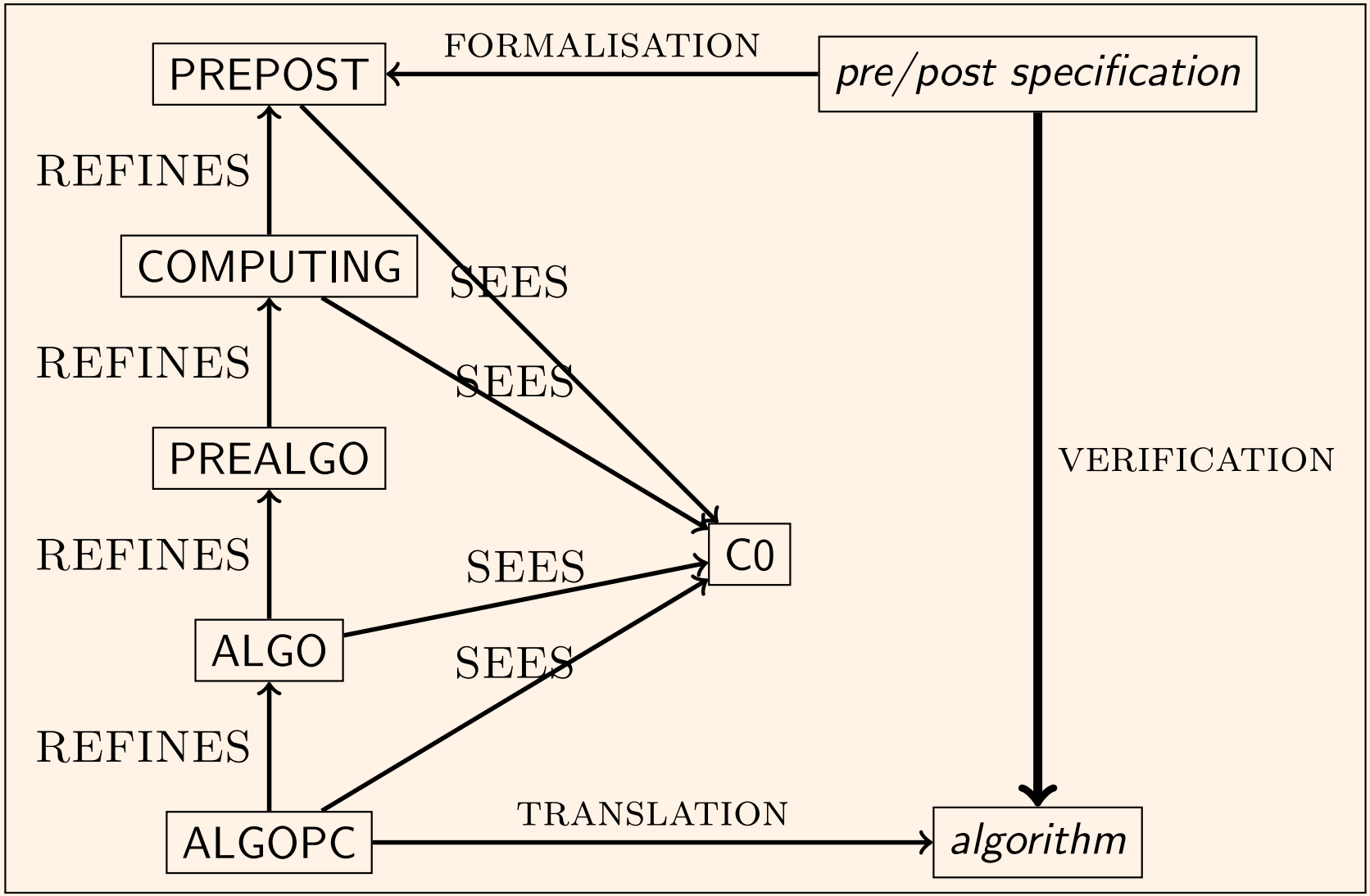
General Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
   - Main idea
   - Justification
5. Transformations of Event B Models into programs
   - Three main possible transfomartions
   - Integrated Development Framework
6. Case study : the binary search problem
7. Deriving the Recursive Program
   - Three possible events
   - Transforming the recursive algorithm
   - In the Spec# world
8. Case studies
   - Binomial coefficients
   - Sorting
   - Floyd's algorithm
   - Developing sequential algorithms
9. Summary

Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

The Iterative Pattern



General context C0

CONTEXT $C0$
SETS
U
CONSTANTS
$x, v, d0, f, D$
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
  REFINES computing,
  WHEN
    $grd1 : x \in dom(vv)$
  THEN
    $act1 : r := vv(x)$
  END
END

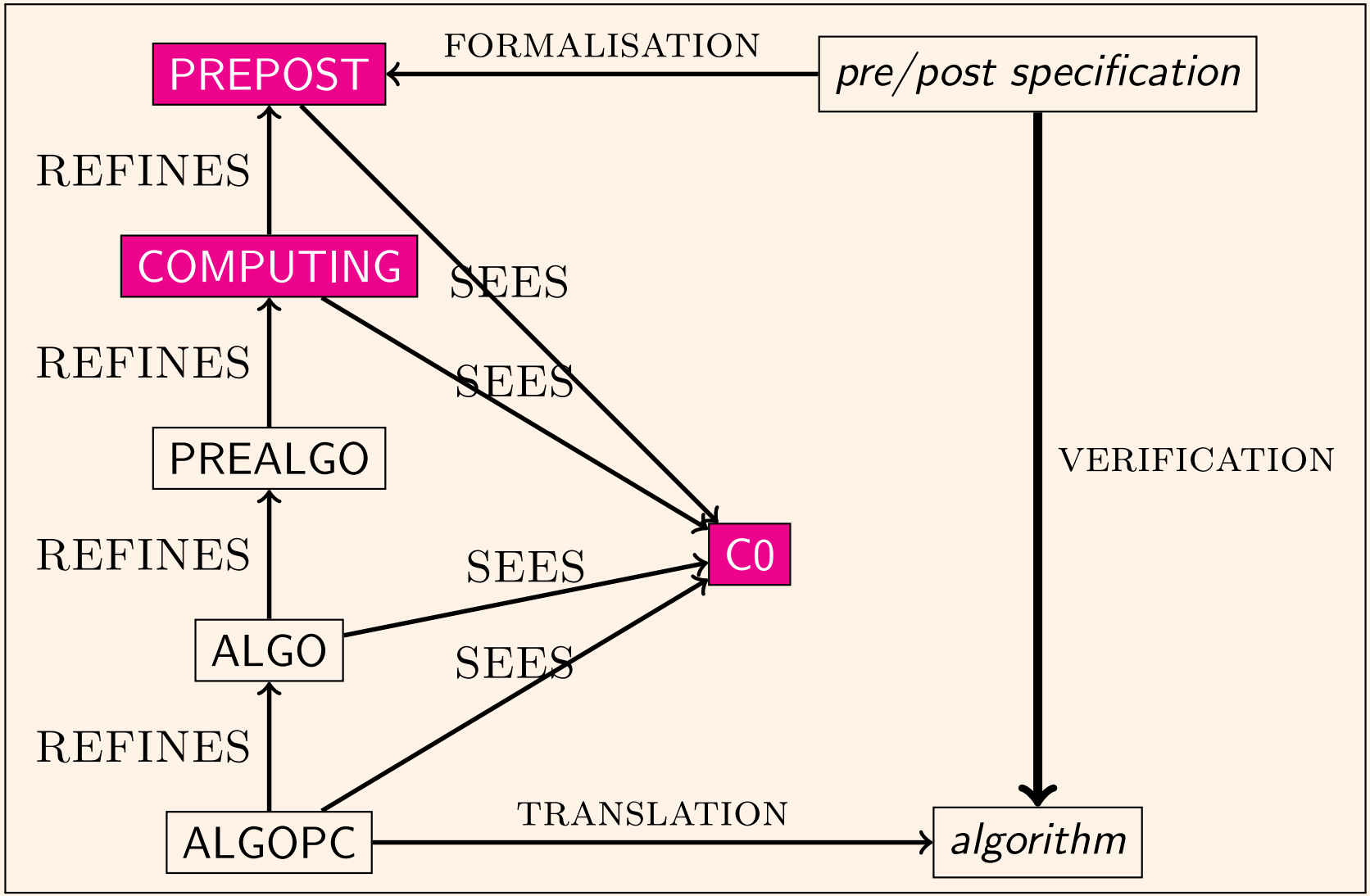
computing is imply observing that the result is computed *simulating* the sequence vv.

First Refinement COMPUTING : Inductive Computation

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.

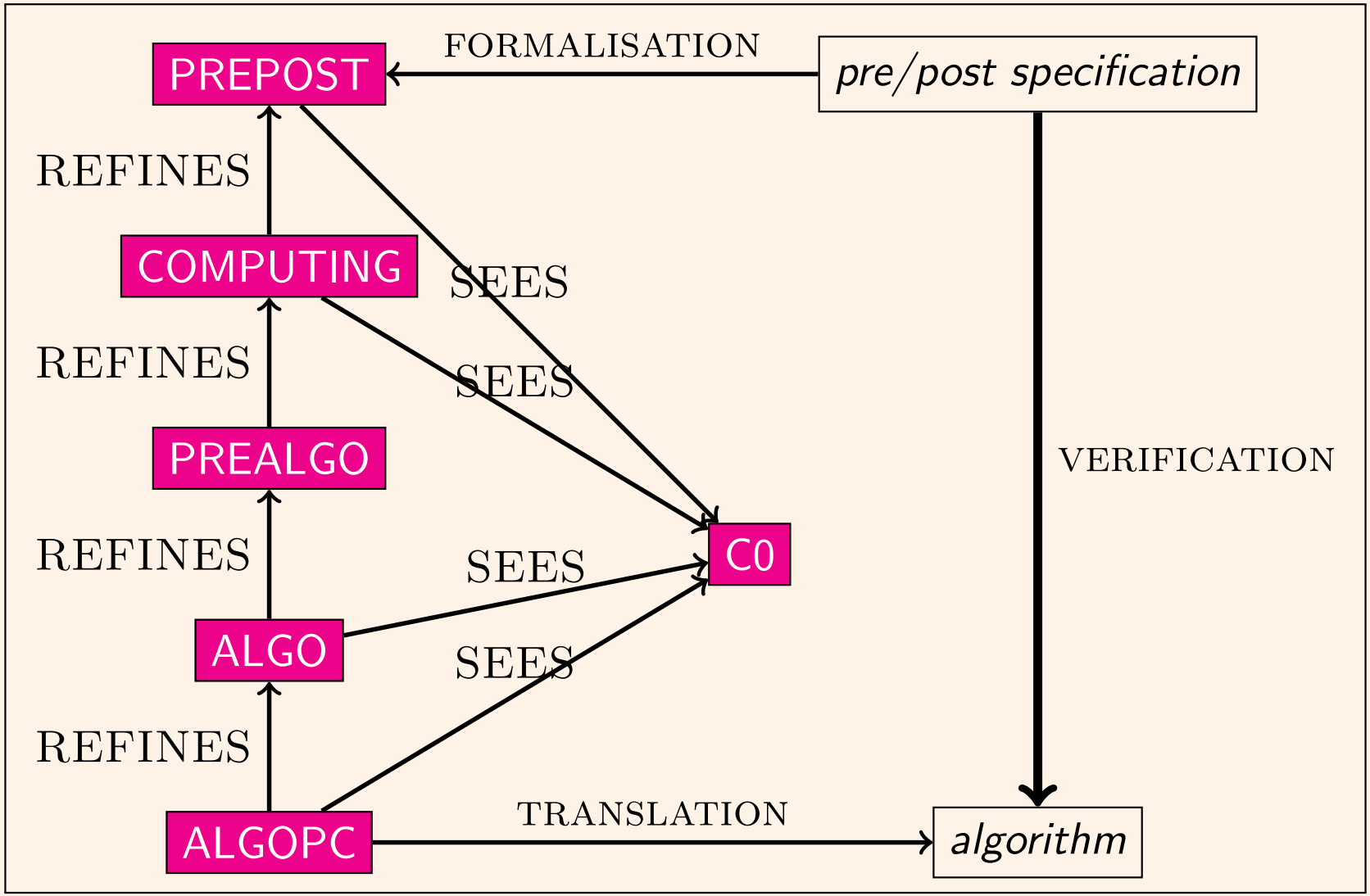
The Iterative Pattern



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.

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);}
```

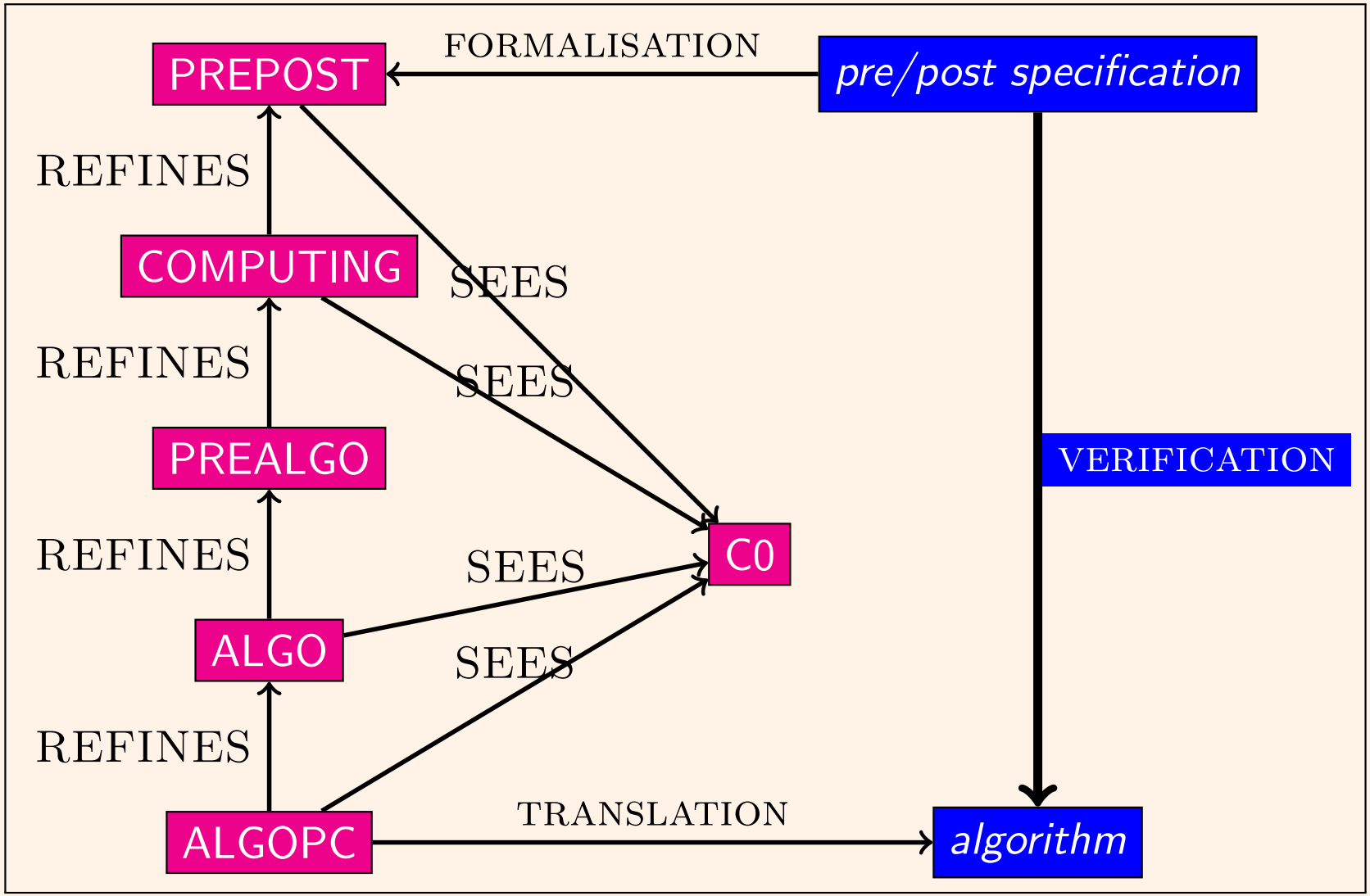
Summary for proof obligations

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 $\forall n \in \mathbb{N} : z_n = n * n * n$.

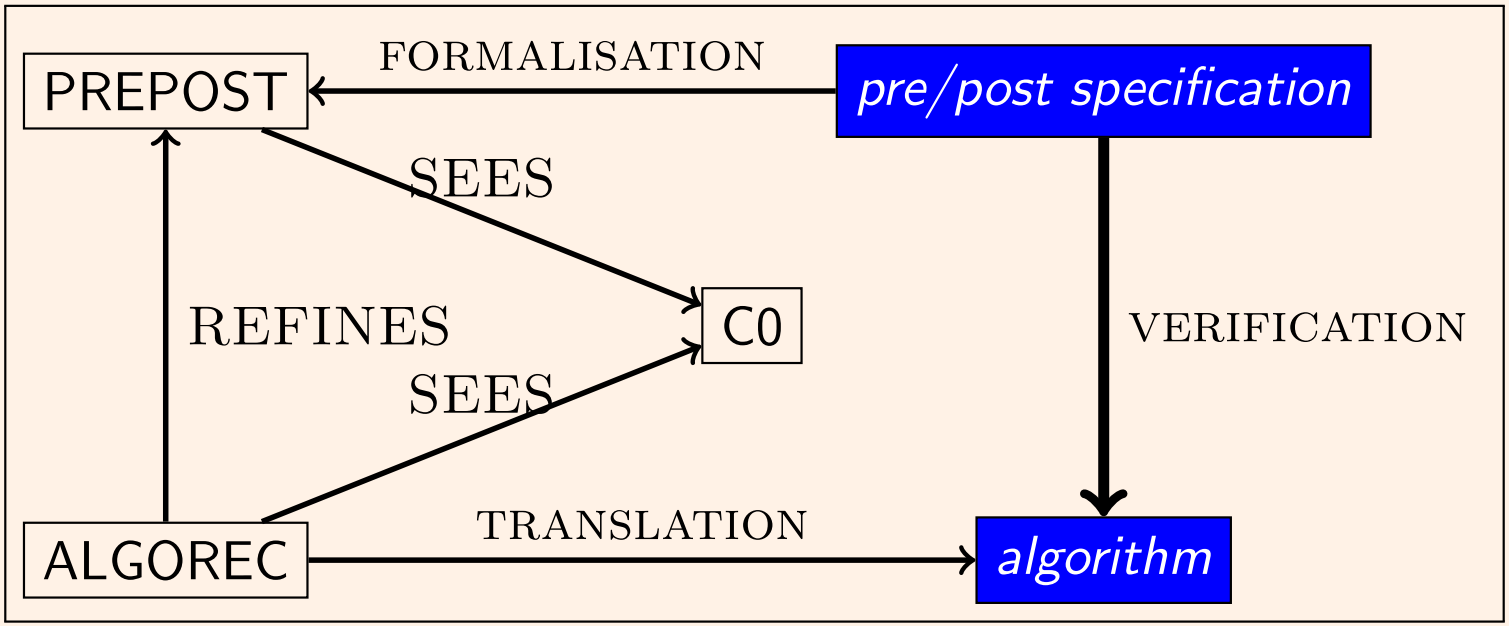
The Iterative Pattern



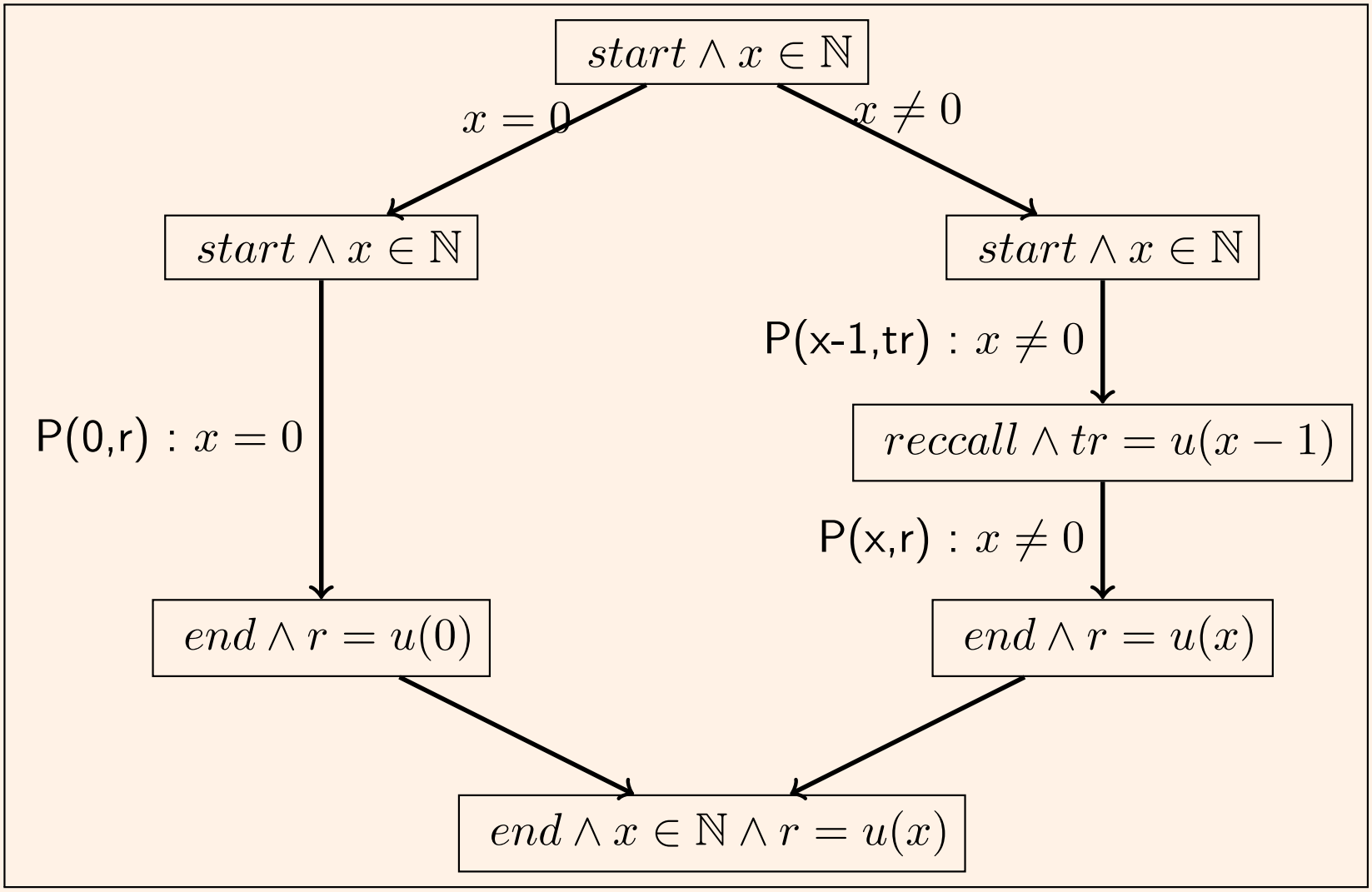
Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

The recursive pattern



The refinement diagram



Ideas for producing algorithms

- **EVENT** rec%PROC(h(x),y)%P(y) is simply simulating the recursive call of the same function.
- The invariant is defined in a simpler way by analysing the inductive structure and a control variable is introduced for structuring the inductive computation.

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

The computation of the function x^2

VARIABLES

r, c, tr

INVARIANTS

$inv1 : r \in \mathbb{N}$

$inv2 : c = end \Rightarrow r = n * n$

$inv3 : c = callrec \Rightarrow n \neq 0$

$inv4 : c = callrec \Rightarrow tr = (n-1) * (n-1)$

$inv5 : c \in C$

$inv6 : tr \in \mathbb{N}$

$inv7 : c = end \Rightarrow r = n * n$

$inv8 : c = end \wedge n \neq 0$
$\Rightarrow tr = (n-1) * (n-1) \wedge r = tr + 2 * (n-1) + 1$

$inv9 : c = callrec \Rightarrow n * n = tr + 2 * (n-1) + 1$

The computation of the function x^2

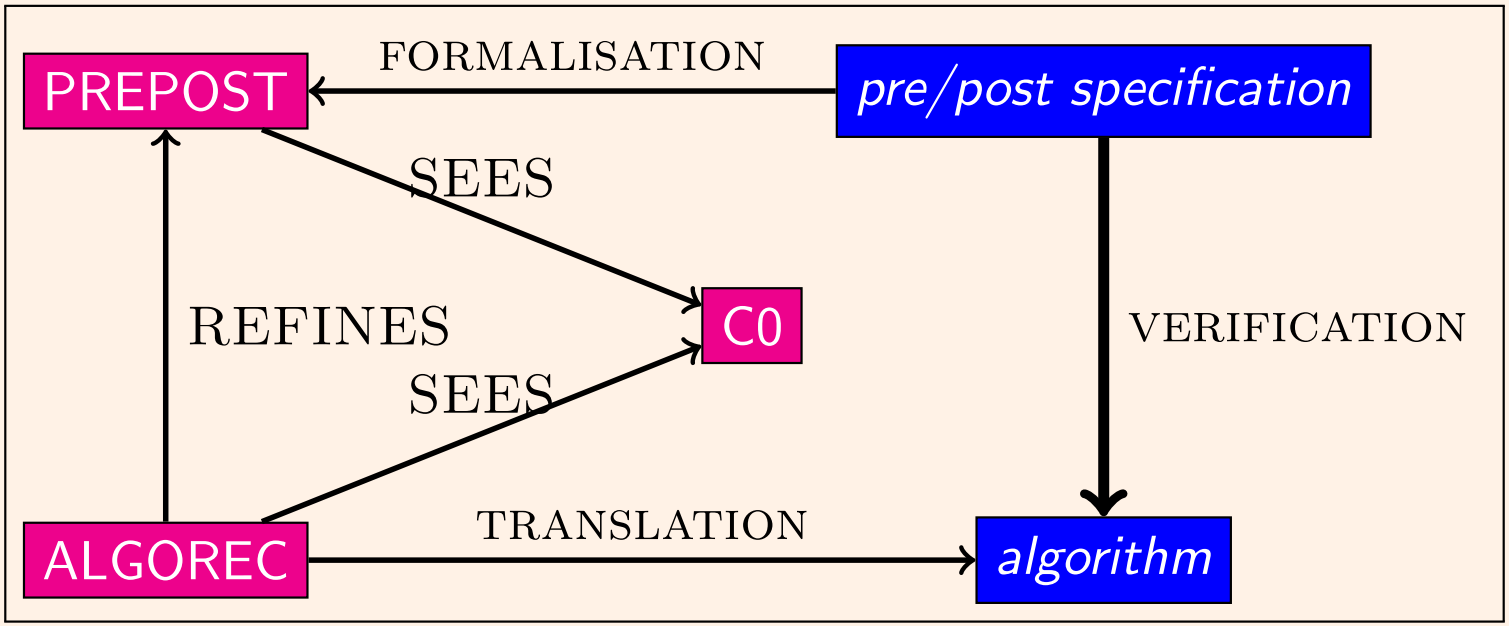
EVENT INITIALISATION
BEGIN
$act1 : r := 0$
$act2 : c := start$
$act3 : tr :\in \mathbb{N}$
END
EVENT square0
REFINES square(n ;r)
WHEN
$grd1 : c = start$
$grd2 : n = 0$
THEN
$act1 : c := end$
$act2 : r := 0$
END

EVENT squaren
REFINES square(n ;r)
WHEN
$grd1 : c = callrec$
THEN
$act1 : r := tr + 2 * (n - 1) + 1$
$act2 : c := end$
END
EVENT rec%square(n-1 ;tr)
WHEN
$grd1 : c = start$
$grd2 : n \neq 0$
THEN
$act1 : c := callrec$
$act2 : tr := (n - 1) * (n - 1)$
END

Summary for proof obligations

Name	Total	Automatic	Interactive
cae-square	34	32	2
square0	3	2	1
specquare	2	2	0
square	29	28	1

The recursive pattern



Comments

- Proofs are easier and simpler
- Invariant is simple to find
- Translation is automatic
- Students are not happy with it and tools are not always set for this verification.

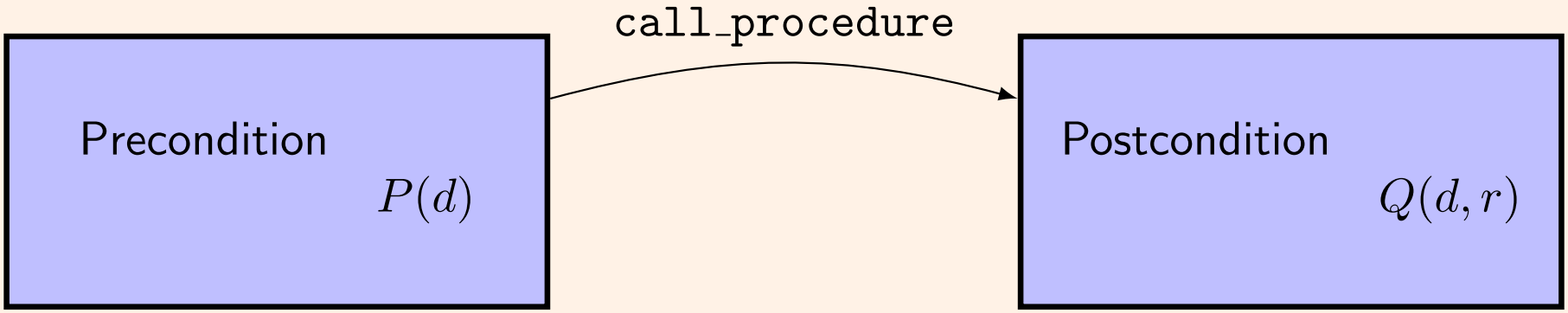
Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Topics

- Providing patterns for helping the use of refinement.
- Making proofs as simple as possible.
- Simplifying the discovery of invariants
- Combining the efforts of program refinement as supported by Event B and program verification as supported by the Spec# programming system.
- Proposing an architecture Integrated Development Framework, which induces a methodology and which improves the usability of formal verification tools for the specification, the construction and the verification of correct sequential algorithms.
- we focus on the transformation of the final concrete specification into an executable algorithm :
  1. transforming an Event B specification into a recursive algorithm
  2. transforming from that recursive program to an iterative version of the same program.

The call as event idea



- `call_procedure` is simulated by the event call_procedure :

EVENT CALL_PROCEDURE
WHEN
$P(d)$
THEN
$r :| Q(d, r')$
END

- $P(d)$ is specifying the inputs of the procedure.
- $Q(d, r)$ is expressing the relation between the inputs d and the results r.

Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Call as event

- A pre/post specification is a pair of assertions $(pre, post)$
- $(pre, post)$ is defining a *transformation* over variables that is correct with respect to *HOARE* triples :

$$\{pre\}transformation\{post\}$$

- **Problem** : Design of a program PROG simulating the *transformation* with respect to $(pre, post)$:

$$\{pre\}\text{PROG}\{post\}$$

- **Idea** : Writing a *one-shot* relation simulating the relation over *flexible* variables :

$$\forall x, y.pre(x) \wedge R(x, y) \Rightarrow post(x, y)$$

- **event** : Expressing the relation :

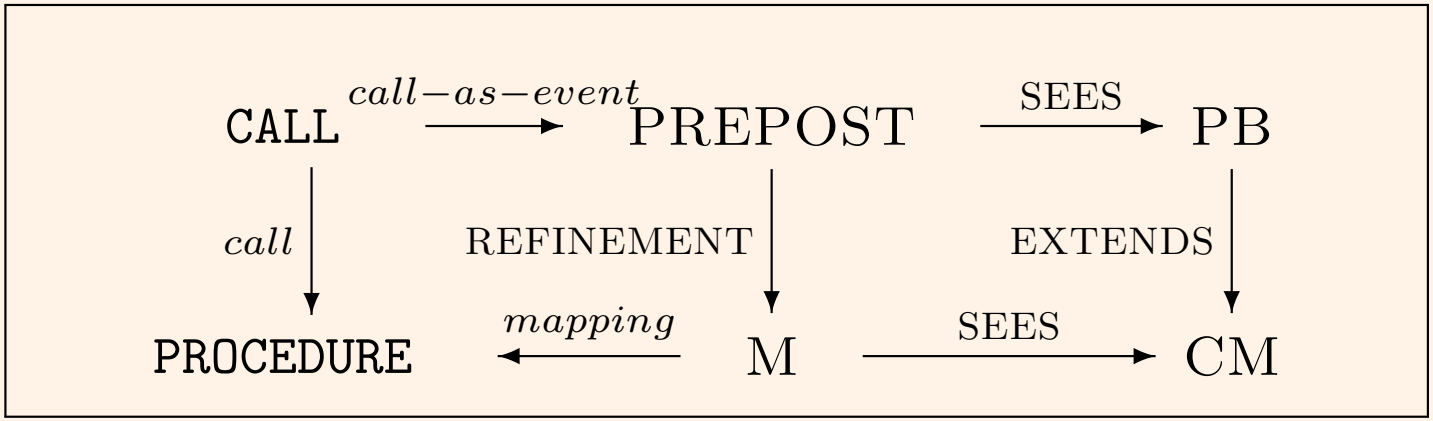
EVENT PROG
WHEN
$pre(x)$
THEN
$y :| post(x, y')$
END

Development of correct sequential algorithms

PROCEDURE PROC$(x;$ **VAR** $y)$
PRECONDITION $P(x)$
POSTCONDITION $Q(x,y)$

- Using the design-by-contract approach
- Introduction of concepts of programming (call-by-value,...) and of modelling (constants, axioms, ...)
- Using diagrams to improve the communication with students through definitions (dynamic programming)
- Organizing the global interactions between modelling and proving.

Call as Event Guideline



- CALL is the call of the PROCEDURE
- PREPOST is the machine containing the events stating the pre- and postconditions of CALL and PROCEDURE, and M is the refinement machine of PREPOST, with events including control points defined in CM.
- The *call-as-event* transformation produces a model PREPOST and a context PB from CALL.
- The *mapping* transformation allows us to derive an algorithmic procedure that can be mechanized.
- PROCEDURE is a node corresponding to a procedure derived from the refinement model M. CALL is an instantiation of PROCEDURE using parameters x and y.
- M is a refinement model of PREPOST, which is transformed into PROCEDURE by applying structuring rules. It may contain events corresponding to calls of other procedures.

Context for modeling the problem PB

- Two domains D_1 and D_2 that define the global set of possible values of the current system for its inputs and outputs.
- A list of constants x that specify the input of the system under development $\widetilde{P}$, which is the set of values for x defining the precondition, and $\widetilde{Q}$, which is a binary relation over $D1 \times D_2$ defining the postcondition for the problem.
- A list of axioms that assigns types to constants and adds knowledge to the RODIN environment.
- The theorems that state the existence of solutions with respect to preconditions : for instance, theorem $th1.i$ states that there is always a solution y when the input value x satisfies the precondition P.

Context for modeling the problem PB

CONTEXT PB
SETS
D_1, D_2
CONSTANTS
$x, n, \widetilde{P}, \widetilde{Q}$
AXIOMS
$axm1 : x \in D_1$
x belongs to a general set of the problem domain
$axm2 : n \in NAT1$
n is the number of different possible preconditions, n is not equal to 0 and it is generally 1
$axm3 : \widetilde{P} \in 1..n \longrightarrow \mathbb{P}(D_1)$
$\widetilde{P}$ *is a collection of sets defining the precondition*
$axm4 : \widetilde{Q} \subseteq D_1 \times D_2$
$\widetilde{Q}$ *is a binary relation over* $Q_1 \times Q_2$ *defining the postcondition*
$axm5 : x \in \bigcup_{i \in 1..n} \widetilde{P}(i)$
x is supposed to satisfy the precondition $P(i)$

THEOREMS
$th1.i : \forall a \cdot a \in \widetilde{P}(i) \Rightarrow (\exists b \cdot a \mapsto b \in \widetilde{Q})$
there is at least one solution for each data x satisfying the precondition P_i
END

From calls to events

EVENT call_i
WHEN
$P_i(x)$
THEN
$y :| (Q(x, y'))$
END

- For each call CALL_i, y is set to a value satisfying $Q(x, y)$ when x satisfies $P(x)$.
- The variable y is initialized by a value satisfying the assertion $Init(y, x, D_1, D_2)$.
- The machine PREPOST, which sees and uses the data defined in the CONTEXT PB and which encapsulates the n cases corresponding to the n preconditions.

Machine PREPOST

MACHINE PREPOST
SEES PB
VARIABLES
y
INVARIANTS
$inv1 : y \in D_2$
EVENTS
INITIALISATION
BEGIN
$act1 : y :\in D_2$
END
...
EVENT call$_i$
WHEN
$P_i(x)$
THEN
$y : |(Q(x, y'))$
END
...
END

- call$_i$ is corresponding to the ith call.
- The relation defined by Q may be non computable
- The refinement can be leading to an non computable solution.

Technical Justifications : defining traces

- **Let M be an Event B machine and C a context seen by M.**
- **Let y be the list of variables of M,**
- **Let E be the set of events of M,**
- **let $Init(y)$ be the predicate defining the initial values of y in M.**

The temporal framework of M is defined by the TLA specification denoted $\mathcal{S}pec(M)$:
$Init(y) \wedge \Box[Next]_y \wedge \mathrm{WF}_y(Next)$**, where** $Next \equiv \exists e \in E.BA(e)(y, y')$**.**

Technical Justifications : liveness properties

Suppose that PB is a context and PREPOST is a machine corresponding to a problem stating calls of a procedure. Suppose that the following diagram is validated :



We assume that the preconditions are defined by P, i.e., $P_1, \ldots, P_n$, and the postcondition is defined by Q.
Then for any i in $1..n$, PREPOST satisfies $P_i(x, y) \rightsquigarrow Q(x, y)$.

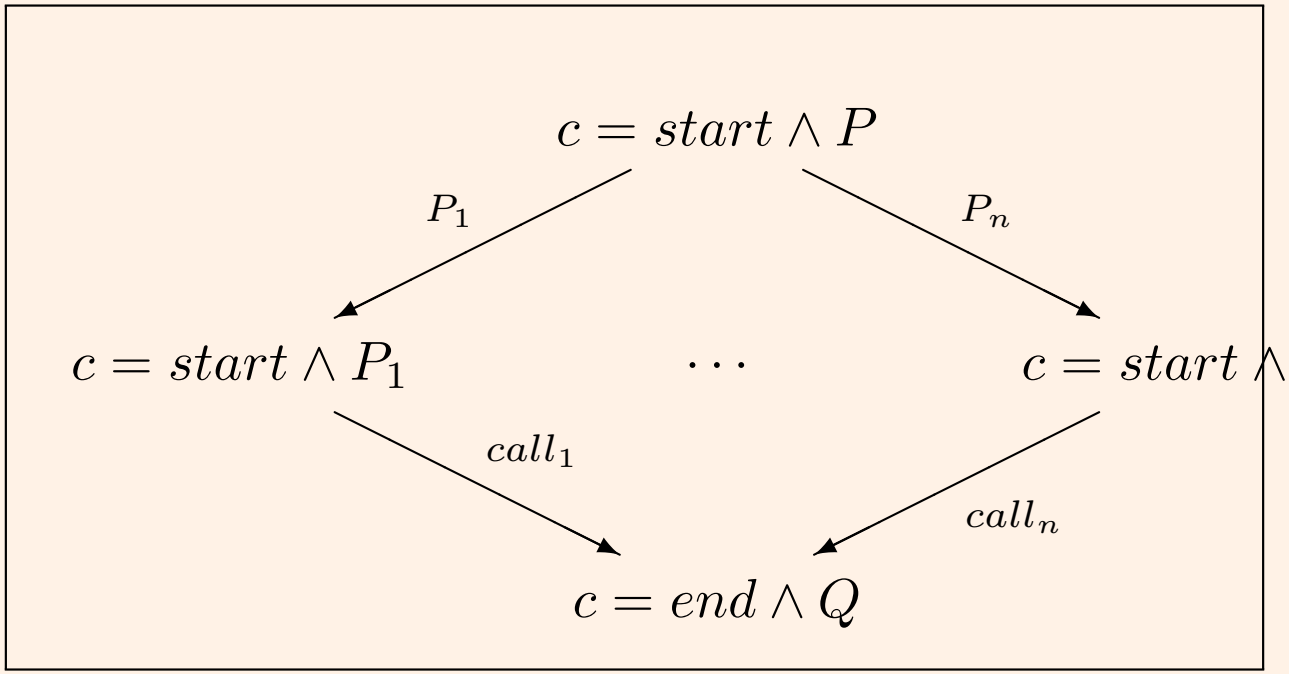
Technical Justifications : refinement diagrams

A refinement diagram for M, P, and Q over L and A is an acyclic labelled graph over A with labels from G or E satisfying the following rules.

- **There is a unique input node P with at least one outgoing arrow.**
- **There is a unique output node Q with no outgoing arrows.**
- **If R is related to S by a unique arrow labelled $e \in E$, then**
  - **It satisfies the property $R \rightsquigarrow S$**
  - $\forall c, x, c', x'.R(c,x) \wedge I(M)(c,x) \wedge BA(e)(c,x,c',x') \Rightarrow S(c',x')$
  - $\forall c, x.R(c,x) \wedge I(M)(c,x) \Rightarrow \exists c', x'.BA(e)(c,x,c',x')$
  - **If $R \equiv c = l1 \wedge A(x)$ and $S \equiv c = l2 \wedge B(x)$, then $l1 \neq l2$ and ℓ(R)=$l1$, ℓ(S)=$l2$.**
- **If R is related to S_1, ..., S_p, then**
  - **Each arrow R to S_i is labelled by a guard $g_i \in G$.**
  - **For any i in $1..p$ the following conditions hold.**
    $$\begin{pmatrix} R \wedge I(M) \wedge g_i(x) \Rightarrow S_i \\ \forall j.j \in 1..p \wedge j \neq i \wedge R \wedge I(M) \wedge g_i(x) \Rightarrow \neg g_j(x) \end{pmatrix}$$
  - $R \wedge I(M) \Rightarrow \exists i \in 1..p.g_i$.
- **For each $e \in E$, there is only one instance of e in the diagram.**

We use $PRE(D)$ for P and $POST(D)$ for Q.

Technical Justifications : a simple refinement diagram



Technical Justifications : refinement diagram and liveness

Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M.

1. **If M satisfies $P \leadsto Q$ and $Q \leadsto R$, it satisfies $P \leadsto R$.**
2. **If M satisfies $P \leadsto Q$ and $R \leadsto Q$, it satisfies $(P \vee R) \leadsto Q$.**
3. **If I is invariant for M and if M satisfies $P \wedge I \leadsto Q$, then M satisfies $P \leadsto Q$.**
4. **If I is invariant for M and if M satisfies $P \wedge I \Rightarrow Q$, then M satisfies $P \leadsto Q$.**
5. **Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M. If $P \xrightarrow{e} Q$ is a link of D for the machine M, then M satisfies $P \leadsto Q$.**
6. **Let M be a machine, and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M. If P and Q are two nodes of D such that there is a path in D from P to Q and any path from P can be extended in a path containing Q, then M satisfies $P \leadsto Q$.**

Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M.
Then M satisfies $(c = start \wedge PRE(D)) \leadsto (c = end \wedge POST(D))$.

Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Case 1 :Basic Events

EVENT e
  WHEN
    $\ell = \ell_1$
    $g_{\ell_1,\ell_2}(x)$
  THEN
    $\ell := \ell_2$
    $x := f_{\ell_1,\ell_2}(x)$
  END1

- If the event e is a basic event controlling the state of the variable x, guarded by $g_{\ell_1,\ell_2}(x)$ and modified by the assignment $x := f_{\ell_1,\ell_2}$ where f is a function, the event e takes the form below.
- the function f_{ℓ_1,ℓ_2} is implementable.
- If the event e labels the link $\ell_1 \xrightarrow{e} \ell_2$ then the statement act_{ℓ_2} is defined as **WHEN** $g_{\ell_1,\ell_2}(x)$ **THEN** $x := f_{\ell_1,\ell_2}(x)$.

Case 2 : Recursive Call of the Procedure

EVENT rec%PROC(h(x),y)%P(y)
ANY y
WHEN
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END1

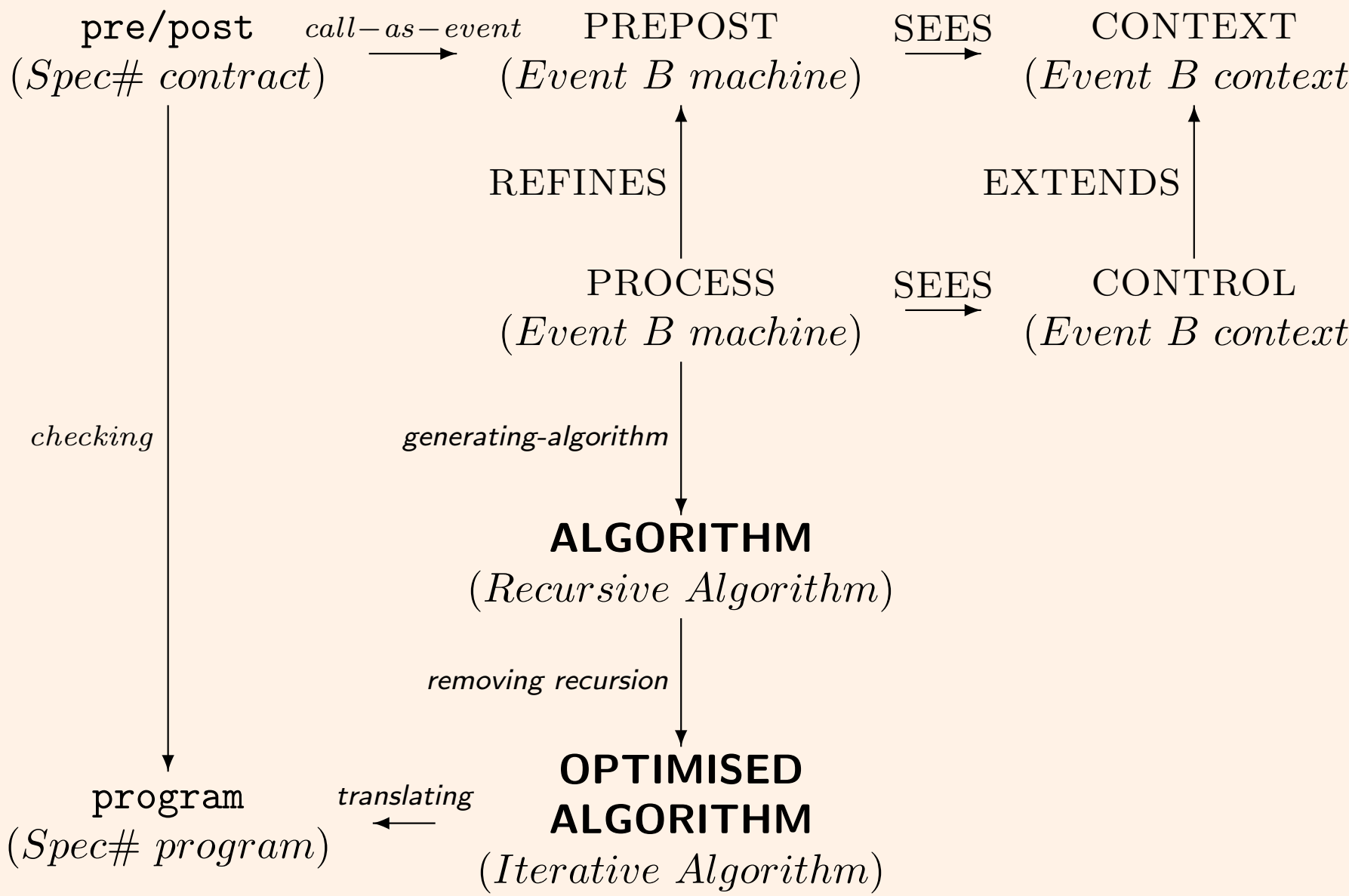
- The definition of the event e is not executable and the translation is driven by instances of the control variable ℓ in the guard (as $\ell = \ell_1$) and in the assignment ($\ell := \ell_2$).
- The statement act_{ℓ_2} is therefore defined as : PROC$(h(x), y)$.
- The choice of the event name is the responsibility of the writer of the Event B models, who must identify the case corresponding to a recursive call.

Case 3 : Non Recursive Call

EVENT call%APROC(h(x),y)%P(y)
ANY y
WHEN
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END1

- the event e can be transformed into a call of another procedure.
- The call is expressed by an event e, which we name $call\%\text{APROC}(h(x), y)\%P(y)$ and the statement act_{ℓ_2} is defined as $\text{APROC}(h(x), y)$.
- APROC is defined or to be defined in another framework.

Integrated Development Framework



ANY t **WHERE** $G(t,x)$ **THEN** $x :|(R(x,x',t))$ **END**

(t is a local parameter and the event actions establish $x :|(R(x,x',t))$
$(\exists t \cdot (G(t,x) \ \wedge \ R(x,x',t)))$

MACHINE $specquare$
SEES $square0$
VARIABLES
r
INVARIANTS
$inv1 : r \in \mathbb{N}$
EVENTS
EVENT INITIALISATION
BEGIN
$act1 : r := 0$
END
EVENT square_computing
BEGIN
$act1 : r := n * n$
END
END

CONTEXT $square0$
CONSTANTS
n
AXIOMS
$axm1 : n \in \mathbb{N}$
END

- *square0* is a context defining properties of a natural number n
- *specsquare* is a machine with an event square_computing computing the square function for n and assigning the value to r.
- The **SEES** clause related the context and the machine.

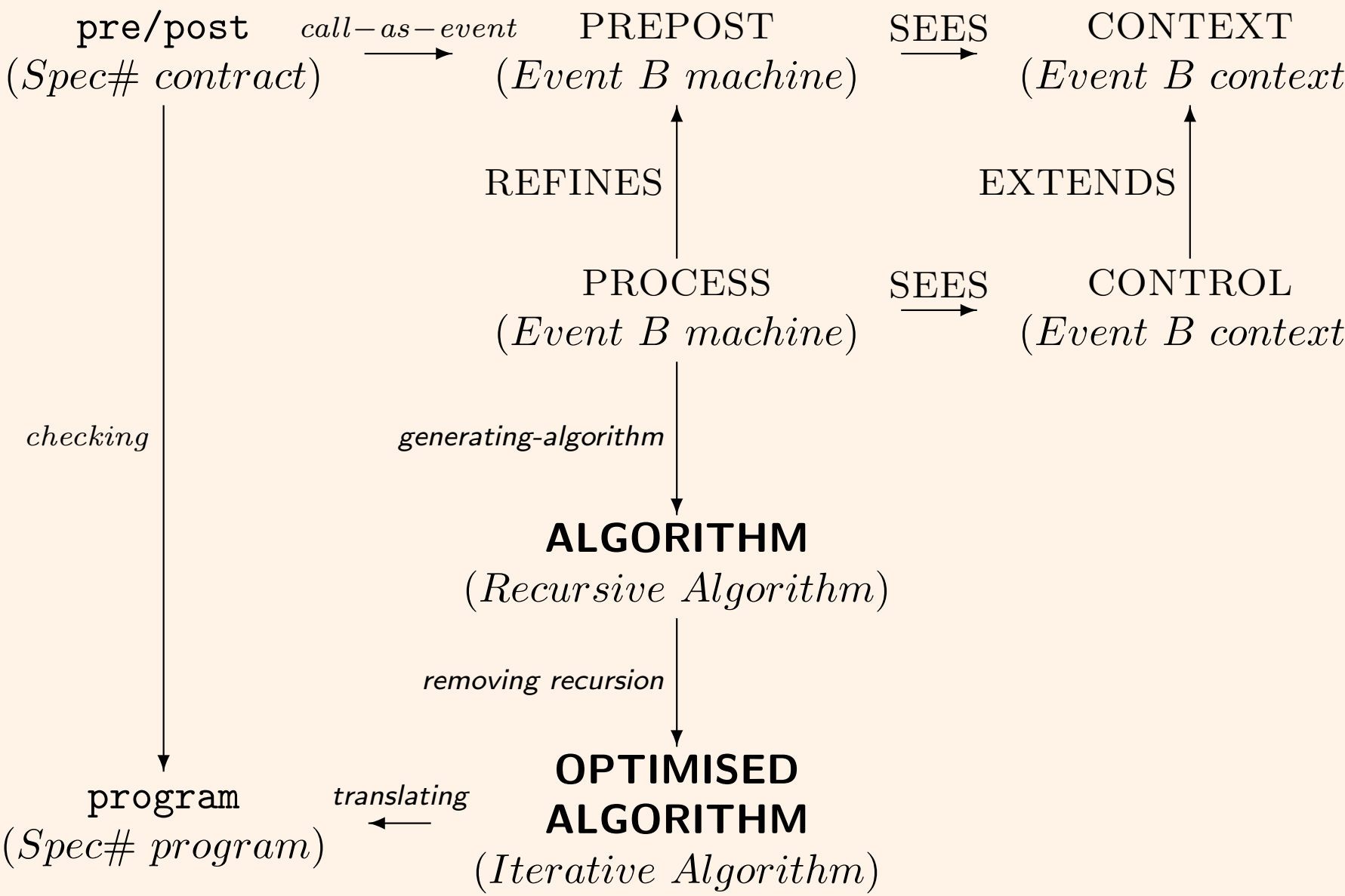
Spec#

- The Spec# programming language extends C# 2.0 through a rich assertion language that allows the specification of objects through class invariants, field annotations, and method specifications.
- The Spec# compiler statically enforces non-null types, emits run-time checks for method contracts and invariants and records the contracts as metadata for consumption by downstream tools.
- The Spec# static program verifier (SscBoogie) : generates logical verification conditions from a Spec# programÂ uses an automatic reasoning engine (Z3) to analyse the verification conditions proving the correctness of the program or finding errors.

Spec# Specification : Sorting an Array

```
public static void sortArray(int[] !st, int[] !pi)
 requires st != pi && st.Length == pi.Length;
 requires forall{int i in (0:st.Length); st[i] == pi[i]};
 modifies st[*];
 ensures forall{int j in (1:st.Length);(st[j-1] <= st[j])};
 ensures forall{int w in (0:st.Length);
        (count{int v in (0:st.Length);st[v] == pi[w]}
        == count{int u in (0:st.Length); pi[u] == pi[w]})};
```

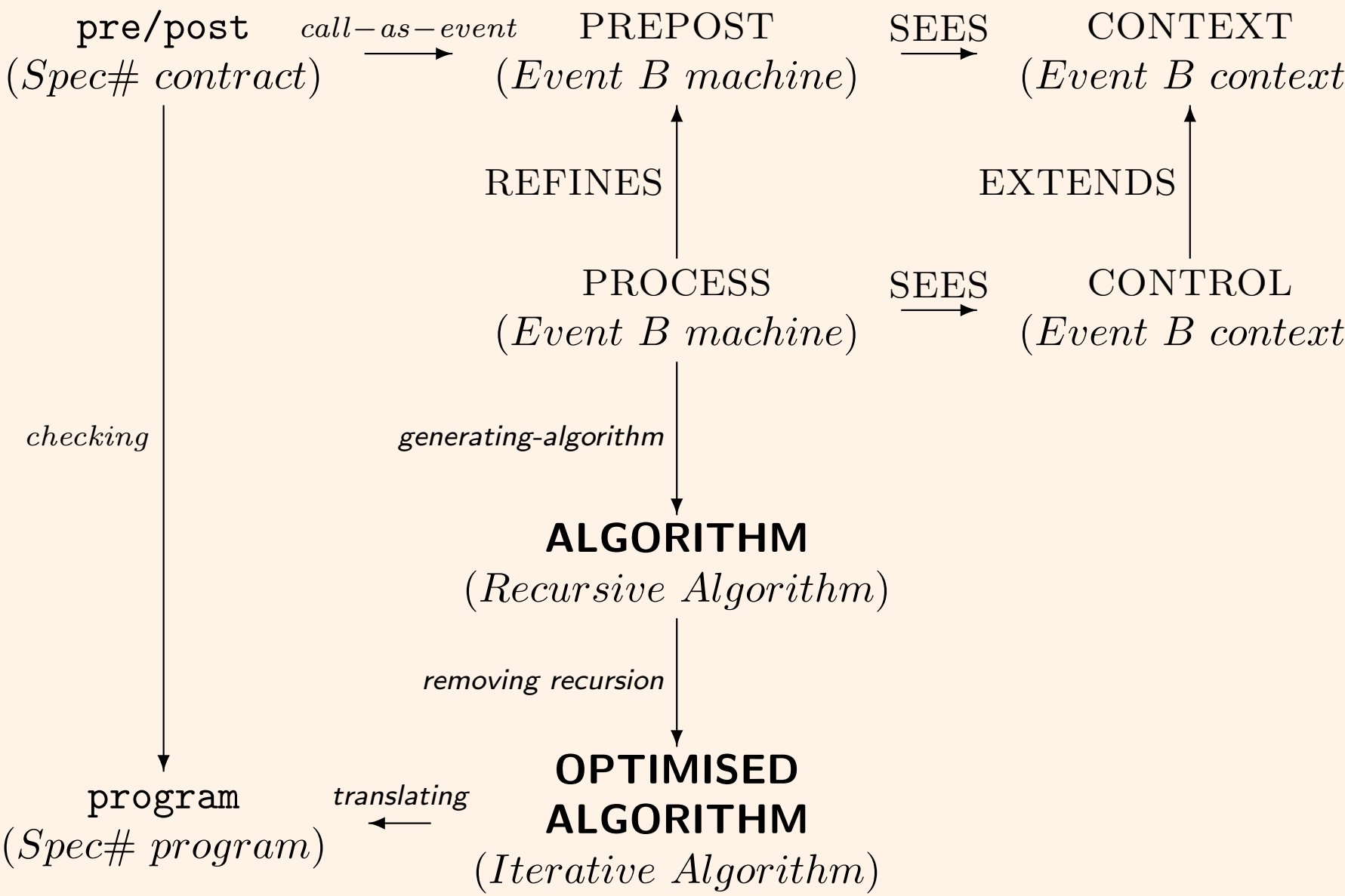
Integrated Development Framework



Integrated Development Framework

- The EVENT B machine PREPOST contains events, which have the same contract as that expressed in the original pre/post contract. This machine SEES the EVENT B CONTEXT, which expresses static information about the machine.
- The EVENT B machine PROCESS refines PREPOST generating a concrete specification that satisfies the contract. This machine SEES the EVENT B context CONTROL, which adds control information for the new machine.
- The labelled actions REFINES, SEES and EXTENDS, are supported by the RODIN platform and are checked *completely* using the proof assistant provided by RODIN.
- Transformation of an EVENT B machine into a concrete recursive algorithm (represented by the arrow labelled *generating-algorithm*).
- Transformation of this recursive algorithm into its equivalent partially annotated and iterative algorithm (represented by the arrow labelled *removing recursion*).

Integrated Development Framework



Implementing Event B models

Our integrated development framework for implementing abstract EVENT B models brings together the strengths of the refinement based approaches and verification based approaches to software development :

1. Splitting the abstract specification to be solved into its component specifications.
2. Refining these specifications into a concrete model using EVENT B and the RODIN platform.
3. Transforming the concrete model into recursive and iterative algorithms that can be directly implemented as real source code.
4. Verifying the iterative algorithm in the automatic program verification environment of Spec#.

Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Specifying the binary search problem

$$
\begin{array}{l}
\text{PROCEDURE} \quad binsearch(t, val, lo, hi, ok, result) \\
\text{PRE} \left(\begin{array}{l} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \\ l, h \in 0..t.Length \\ lo \leq hi \end{array} \right) \\
\text{POST} \left(\begin{array}{l} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array} \right)
\end{array}
$$

The two possible resulting *calls* to the procedure $binsearch(t, val, lo, hi; ok, result)$:

- **EVENT** find is $binsearch(t, val, lo, hi; ok, result)$ with $ok = TRUE$
- **EVENT** fail is $binsearch(t, val, lo, hi; ok, result)$: with $ok = FALSE$

Events find and fail

EVENT find
  ANY j
  WHERE
    $grd1 : j \in lo \,..\, hi$
    $grd2 : t(j) = val$
  THEN
    $act1 : ok := TRUE$
    $act2 : result := j$
  END

EVENT fail
  WHEN
    $grd1 : \forall k \cdot k \in lo \,..\, hi \Rightarrow t(k) \neq val$
  THEN
    $act1 : ok := FALSE$
END

- The two events form the machine called *binsearch1* (which corresponds to the PREPOST machine).
- The machine is refined to obtain *binsearch2* (which corresponds to PROCESS).
- This refined machine contains a new control variable, l, which *simulates* how the binary search is achieved.

Refinement for Computation

1. $\begin{pmatrix} l = start \\ lo = hi \\ t(lo) = val \end{pmatrix} \xrightarrow{m_1} \begin{pmatrix} l = end \\ lo = hi \\ ok = TRUE \ \wedge \ result = lo \end{pmatrix}$
2. $\begin{pmatrix} l = start \\ lo = hi \\ t(lo) \neq val \end{pmatrix} \xrightarrow{m_2} \begin{pmatrix} l = end \\ lo = hi \\ ok = FALSE \end{pmatrix}$
3. $\begin{pmatrix} l = start \\ lo < hi \end{pmatrix} \xrightarrow{split} \begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \end{pmatrix}$
4. $\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge t(result) = val \end{pmatrix}$

Refinement for Computation

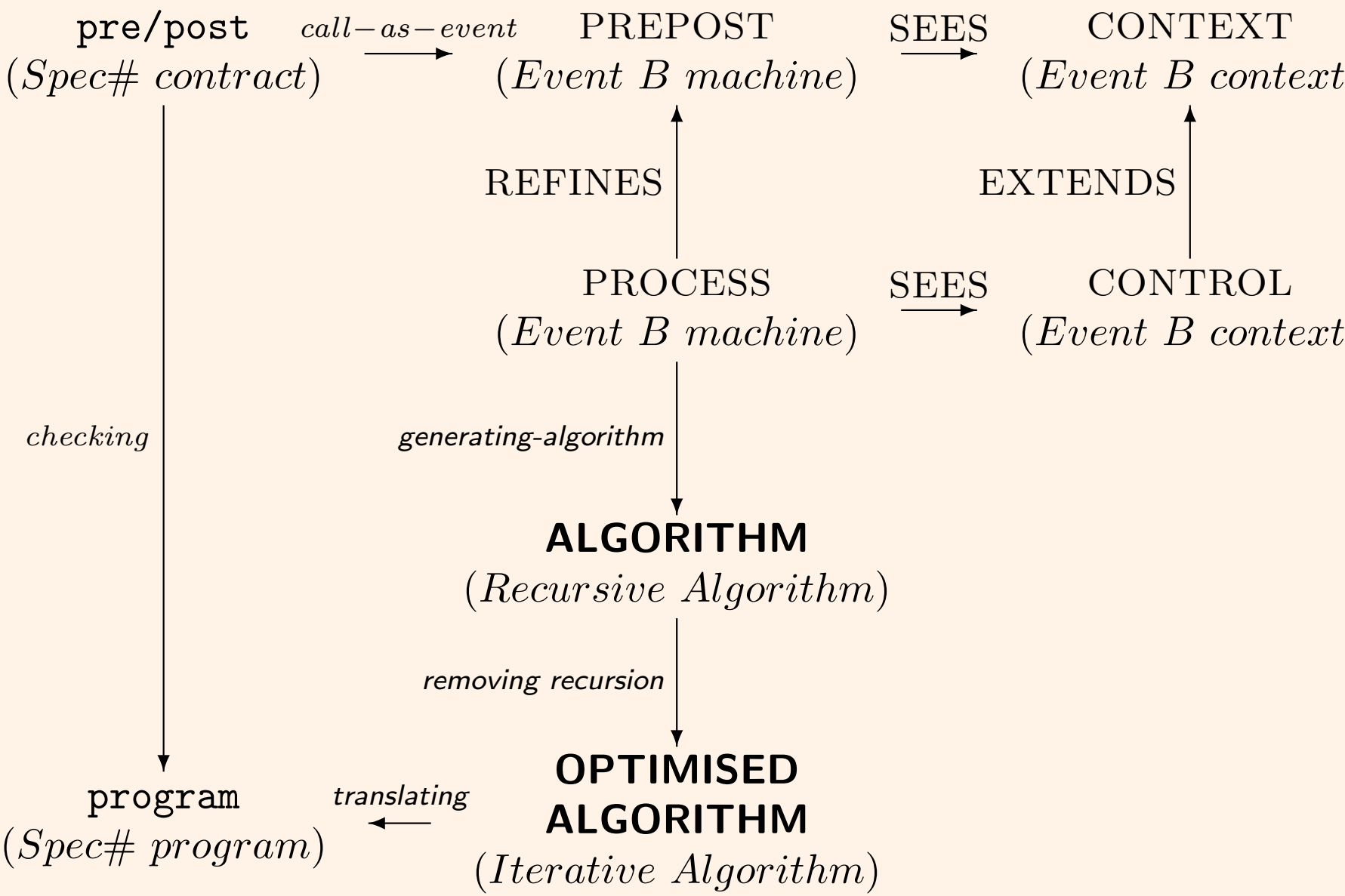
1. $$\begin{pmatrix} l = middle \\ lo < hi \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \\ \wedge\ ok = FALSE \wedge (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

2. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val = t(mi) \end{pmatrix} \xrightarrow{m_3} \begin{pmatrix} l = end \\ ok = TRUE \ \wedge \ result = mi \end{pmatrix}$$

3. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge \ t(result) = val \end{pmatrix}$$

4. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \\ \wedge\ ok = FALSE \wedge (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

Integrated Development Framework



Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Case 1 :Basic Events

EVENT e
  WHEN
    $\ell = \ell_1$
    $g_{\ell_1,\ell_2}(x)$
  THEN
    $\ell := \ell_2$
    $x := f_{\ell_1,\ell_2}(x)$
  END1

- If the event e is a basic event controlling the state of the variable x, guarded by $g_{\ell_1,\ell_2}(x)$ and modified by the assignment $x := f_{\ell_1,\ell_2}$ where f is a function, the event e takes the form below.
- the function f_{ℓ_1,ℓ_2} is implementable.
- If the event e labels the link $\ell_1 \xrightarrow{e} \ell_2$ then the statement act_{ℓ_2} is defined as **WHEN** $g_{\ell_1,\ell_2}(x)$ **THEN** $x := f_{\ell_1,\ell_2}(x)$.

Case 2 : Recursive Call of the Procedure

EVENT rec%PROC(h(x),y)%P(y)
ANY y
WHEN
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END1

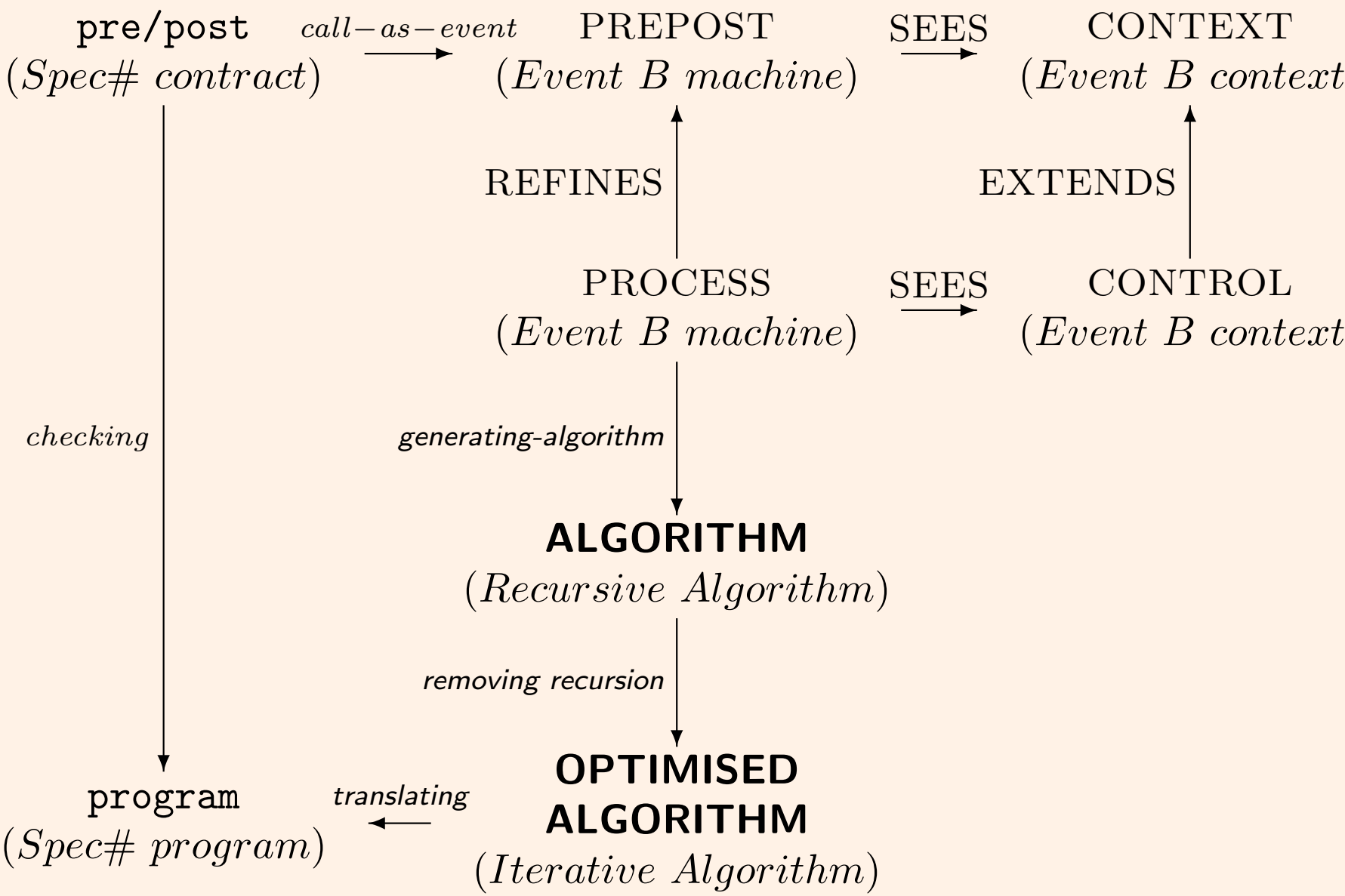
- The definition of the event e is not executable and the translation is driven by instances of the control variable ℓ in the guard (as $\ell = \ell_1$) and in the assignment ($\ell := \ell_2$).
- The statement act_{ℓ_2} is therefore defined as : PROC$(h(x), y)$.
- The choice of the event name is the responsibility of the writer of the Event B models, who must identify the case corresponding to a recursive call.

Case 3 : Non Recursive Call

EVENT call%APROC(h(x),y)%P(y)
ANY y
WHEN
$\ell = \ell_1$
$g_{\ell_1,\ell_2}(x, y)$
THEN
$\ell := \ell_2$
$x := f_{\ell_1,\ell_2}(x, y)$
END1

- the event e can be transformed into a call of another procedure.
- The call is expressed by an event e, which we name $call\%\text{APROC}(h(x), y)\%P(y)$ and the statement act_{ℓ_2} is defined as $\text{APROC}(h(x), y)$.
- APROC is defined or to be defined in another framework.

Integrated Development Framework



Problems with the recursive algorithm

- Correct by construction according to the Event B side
- Limits of the Spec# tools with verifying recursive program
- Transforming recursive algorithms into iterative algorithm
- Applying the Spec# tools

Transforming the recursive algorithm into an iterative one

Theorem

The transformation is sound with respect to the pre and post specification.

```
PROCEDURE APROC(x; VAR y)
PRECONDITION P(x)
POSTCONDITION Q(x, y)
BEGIN
LOCAL VARIABLES z
IF C(x) THEN
  y := g(x);
ELSE
  z := h(x, z);
  IF D(x, z) THEN
    y := f(x, z)
  ELSEIF E(x, z) THEN
    APROC(f1(x), y)
  ELSE
    APROC(f2(x), y)
  ENDIF
END
```

```
PROCEDURE BPROC(x; VAR y)
PRECONDITION P(x)
POSTCONDITION Q(x, y)
BEGIN
LOCAL VARIABLES z
WHILE not C(x) ∧ not D(x, z) DO
  z := h(x, z);
  IF E(x, z) THEN
    x := f1(x);
  ELSE
    x := f2(x);
  ENDIF
ENDDO
IF C(x) THEN
  y := g(x);
ELSEIF D(x, z) THEN
  y := f(x, z);
ELSEIF E(x, z) THEN
  y := f1(x);
ELSE
  y := f2(x);
  ENDIF
END
```

PROCEDURE binsearch(t,val,lo,hi,ok,result)

PROCEDURE $binsearch(t, val, lo, hi, ok, result)$

PRECONDITION $\left(\begin{array}{l} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \wedge lo, hi \in 0..t.Length \wedge lo \leq hi \end{array} \right)$

POSTCONDITION $\left(\begin{array}{l} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array} \right)$

BEGIN

WHILE $not \left(\begin{array}{l} lo = hi \wedge t(lo) = val \\ \vee \ lo = hi \wedge t(lo) \neq val \\ \vee \ lo < hi \wedge mi = (lo+hi)/2 \wedge t(mi) = val \end{array} \right)$ **DO**

$mi := (lo + hi)/2;$

$middle : \left\{ \left(\begin{array}{l} mi = (lo+hi)/2 \\ val < t(mi) \Rightarrow \forall k.k \in mi..hi \Rightarrow t(k) \neq val \\ val > t(mi) \Rightarrow \forall k.k \in lo..mi \Rightarrow t(k) \neq val \end{array} \right) \right\}$

IF $mi + 1 \leq hi \wedge val > t(mi)$ **THEN**

$lo := mi + 1$

ELSEIF $lo \leq mi - 1 \wedge val < t(mi)$ **THEN**

$hi := mi - 1$

ENDDO

IF $lo = hi \wedge t(lo) = val$ **THEN**

$result := lo; ok := true$

ELSEIF $lo = hi \wedge t(lo) \neq val$ **THEN**

$ok := false$

ELSEIF $lo < hi \wedge t(mi) = val$ **THEN**

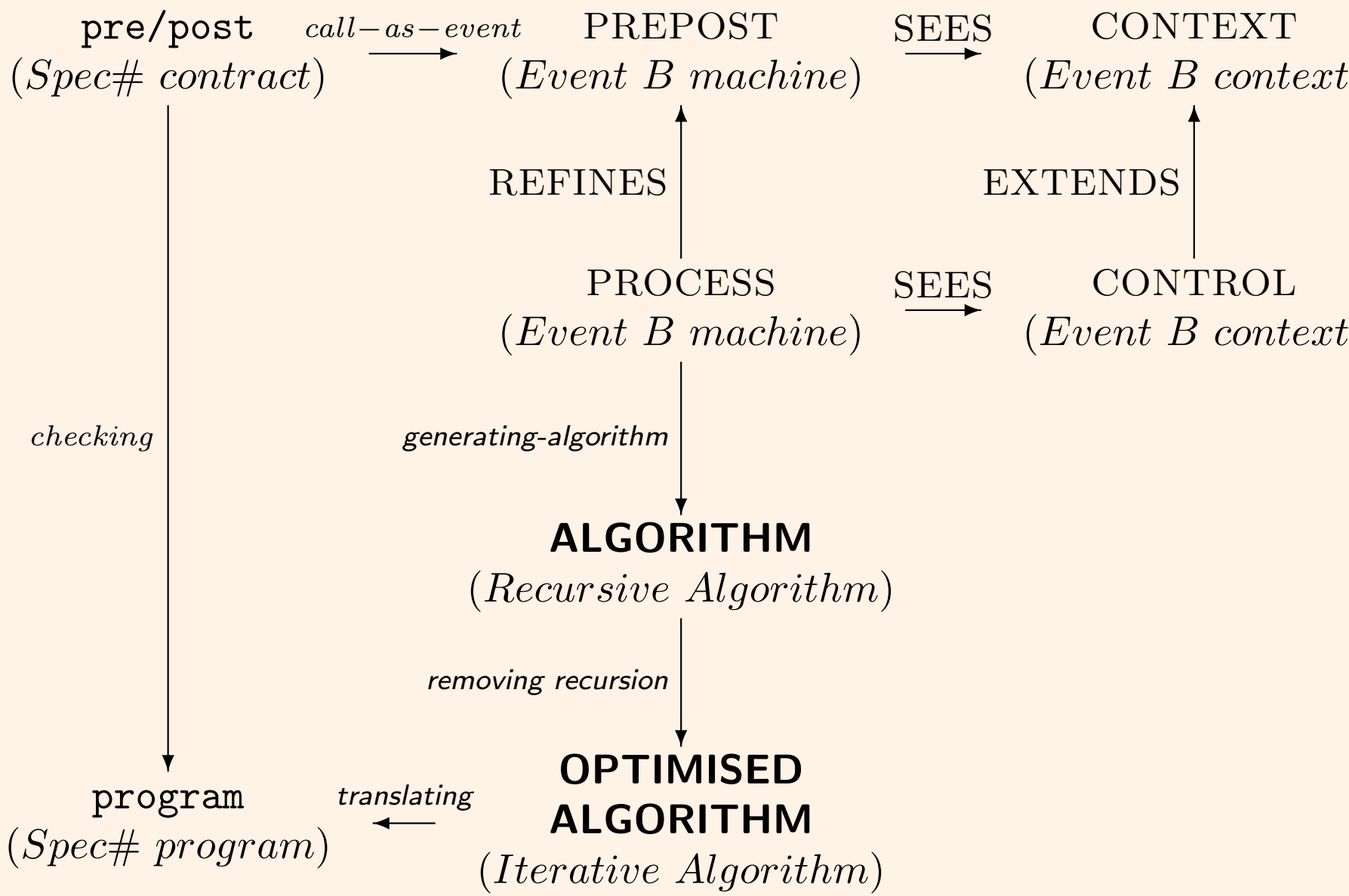
$result := mi; ok := true$

ELSE $ok := false$

ENDIF

END

General Framework Integrated Development Framework



Interpreting the algorithms within Spec#

- This is almost a one-to-one mapping : returning a value of -1 when our iterative algorithm sets OK to $false$ and returning the index where the value is found when our iterative algorithm sets OK to $true$.
- The algorithm verified as correct, in less than 2 seconds using the Spec# programming system (version 2011-10-03).
- No user interaction is required in the 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.
- Prior to formalising our transformation rules, our initial attempt at writing this iterative C# program contained an error.
- This error in the loop body, was due to our omission to check that the values of $mi + 1$ and $mi - 1$ were within the array bounds before narrowing the search space.
- This error was immediately detected by the Spec# programming system. The automatic verification of the final program is available online at `http://www.rise4fun.com/SpecSharp/psP4`.

Conclusion

- Our integrated development framework indicates where our transformations are used for producing a program that is *correct-by-construction*.
- The translation of the PROCESS machine into a recursive algorithm is straightforward and removes the control variable used to relate events when generating the code.
- Our experience shows that our approach assists students in developing and understanding the tasks of software specification and verification.
- It also makes different forms of formal software development more accessible to the Software Engineers, helping them to build correct and reliable software systems.
- *EB2RC* : Update Site : http://www.cs.nuim.ie/∼zcheng/EB2RC/TEB2RC/ : http://www.cs.nuim.ie/∼zcheng/EB2RC Zheng Cheng [zcheng@cs.nuim.ie] and Rosemary Monahan [rosemary.monahan@nuim.ie]

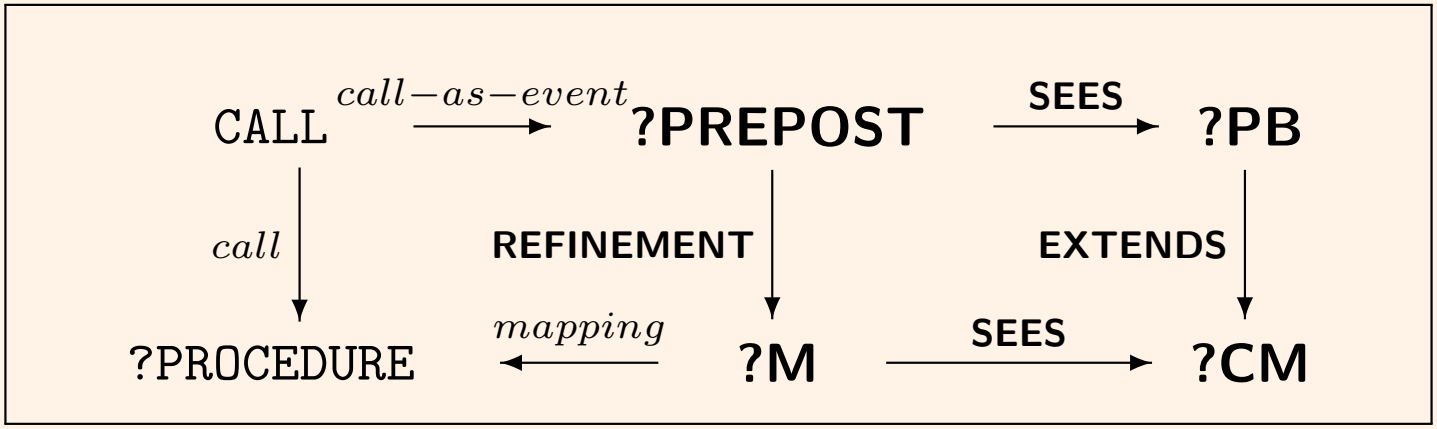
Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Three problems to solve

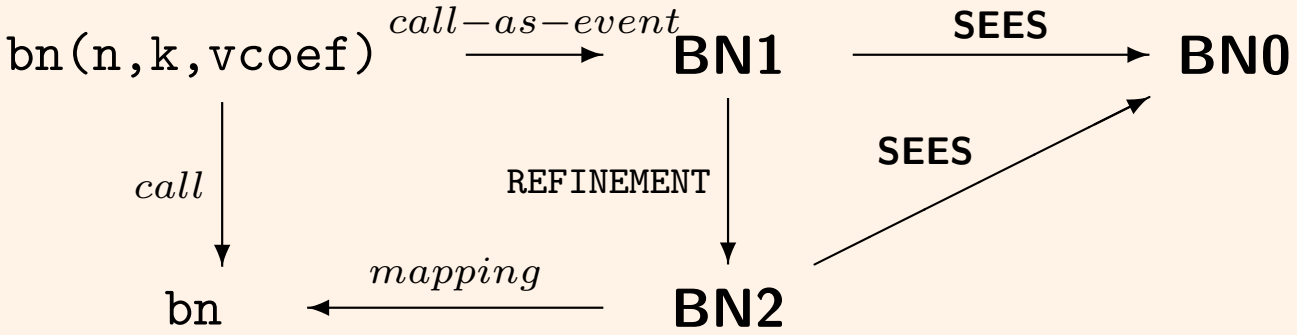
- Presenting the method : Computing binomial coefficients
- Illustrating one classical example : Sorting by insertion
- Using the dynamic programming : Floyd's algorithm

Filling the Call as Event Guideline



- **PB : using textbooks for deriving axioms and theorems.**
- **CALL : using a programming language like JML or Spec#**

Problem 1 : Computing the binomial coefficients



- **The computation of binomial coefficients is based on Pascal's triangle and we define it as a partial function c.**
- **Data n and k are defined in the context called $BN0$.**
- **The call $bn(n, k, vcoef)$ is translated as an event which is simply setting $vcoef$ to $c(n \mapsto k)$.**
- **The refinement $BN2$ produces a collection of events analysing the different steps of the computations required for computing the value of $c(n \mapsto k)$.**

Comments on the application 1

- **Pascal's triangle provides a graphical guide for writing d's definition into $BN0$.**
- **We have introduced another graphical structure for supporting the case analysis related to the values of n and k and the introduction of control flow.**
- **The world of mathematics is defining a value $c(n \mapsto k)$ and the world of computing will derive a process using c and its definition for producing the same value.**

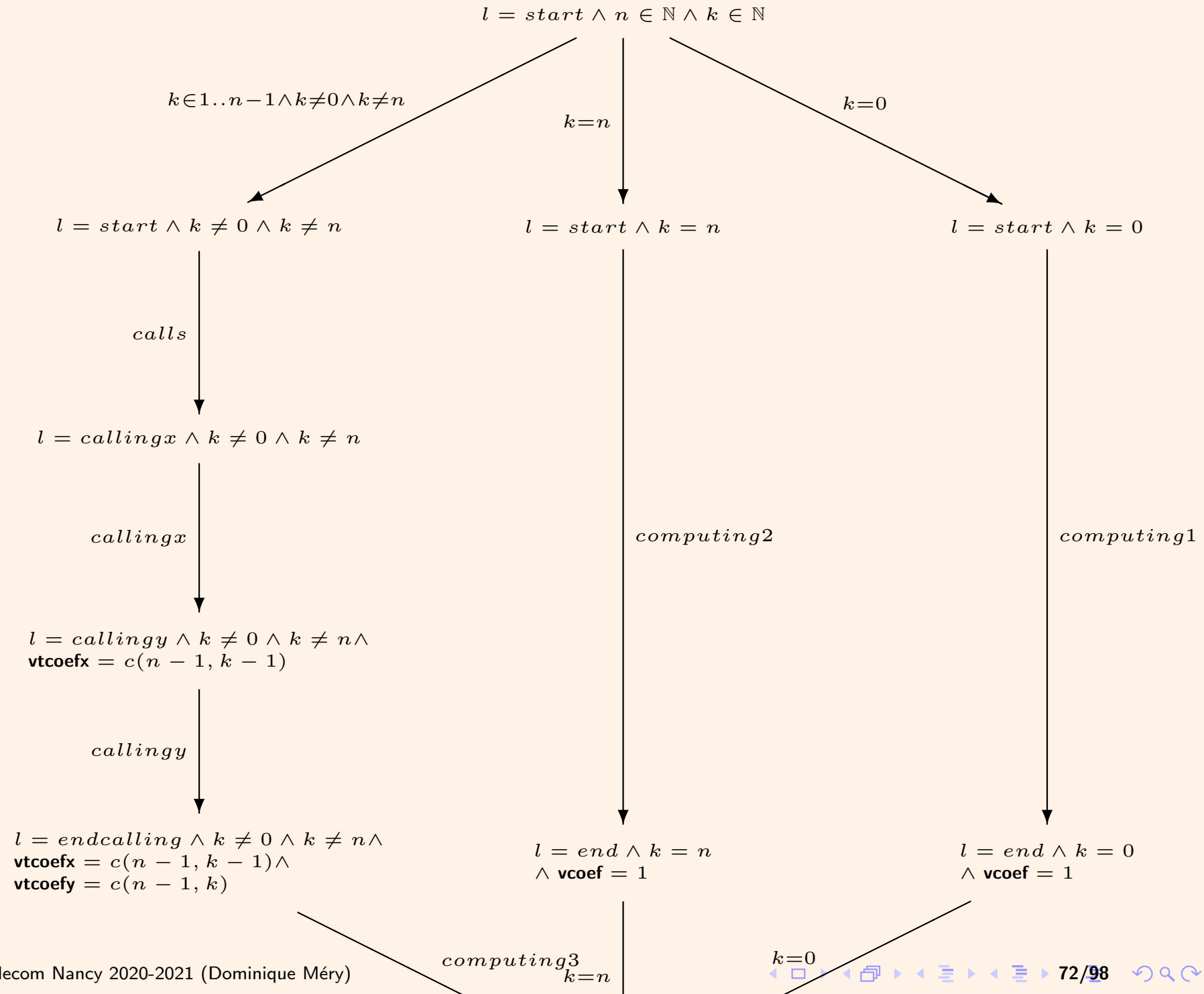
  ...

Comments on the application 1

- **The refinement i is guided by three cases for the call instances :**
  - **Either k is 0,**
  - **or k is n,**
  - **or is neither 0, nor n.**
- **Let us consider the difficult case : $k \neq 0$ and $k \neq n$:**

$$\forall k \in \{1, \ldots n-1\}. \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad (1)$$

  - **Using the same event for computing $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$.**
  - **These events are translated into a recursive call by the mapping.**
  - **the final computing event is computing the value of the sum of the two values.**


$l = start \wedge n \in \mathbb{N} \wedge k \in \mathbb{N}$
$k \in 1..n-1 \wedge k \neq 0 \wedge k \neq n$
$k=n$
$k=0$
$l = start \wedge k \neq 0 \wedge k \neq n$
$l = start \wedge k = n$
$l = start \wedge k = 0$
$calls$
$l = callingx \wedge k \neq 0 \wedge k \neq n$
$callingx$
$computing2$
$computing1$
$l = callingy \wedge k \neq 0 \wedge k \neq n \wedge$
$\mathbf{vtcoefx} = c(n-1, k-1)$
$callingy$
$l = endcalling \wedge k \neq 0 \wedge k \neq n \wedge$
$\mathbf{vtcoefx} = c(n-1, k-1) \wedge$
$\mathbf{vtcoefy} = c(n-1, k)$
$l = end \wedge k = n$
$\wedge \mathbf{vcoef} = 1$
$l = end \wedge k = 0$
$\wedge \mathbf{vcoef} = 1$
$computing3$
$k=n$
$k=0$

Comments on the application 1

- **INVARIANTS**

$inv1 : \ l \in LOC$
$inv3 : \ vtcoefx \in \mathbb{N}$
$inv4 : \ vtcoefy \in \mathbb{N}$
$inv5 : \ l \in \{callx, cally, endcalling\} \Rightarrow k \neq 0 \wedge n \neq 0 \wedge k < n$
$inv6 : \ l = cally \Rightarrow vtcoefx = c(n-1 \mapsto k-1)$
$inv7 : \ l = endcalling \Rightarrow vtcoefy = c(n-1 \mapsto k) \wedge vtcoefx = c(n-1 \mapsto k-1)$
$inv8 : \ l = end \Rightarrow vcoef = c(n \mapsto k)$

- **The refinement produces 42 proof obligations and 2 were manual. The other proof obligations are automatically discharged.**

Teacher's note: The example is simple and the function c is easy to define. The invariant is built by analysing the expression of the computed value.

BINOMIAL0
SETS
LOC
CONSTANTS
$n, k, c, start, end, callx, cally, endcalling$
AXIOMS
$axm1 : n \in \mathbb{N}$
$axm2 : k \in \mathbb{N}$
$axm3 : k \leq n$
$axm4 : c \in 0 \mathinner{..} n \times 0 \mathinner{..} n \pfun \mathbb{N}$
$axm5 : dom(c) = \{i \mapsto j | i \in 0 \mathinner{..} n \land j \in 0 \mathinner{..} n \land j \leq i\}$
$axm6 : \forall j \cdot j \in 0 \mathinner{..} n \Rightarrow c(j \mapsto 0) = 1$
$axm7 : \forall j \cdot j \in 0 \mathinner{..} n \Rightarrow c(j \mapsto j) = 1$
$axm8 : LOC = \{start, end, callx, cally, endcalling\}$
$axm9 : start \neq end$
$axm10 : start \neq callx$
$axm11 : start \neq cally$
$axm12 : start \neq endcalling$
$axm13 : end \neq callx$
$axm14 : end \neq cally$
$axm15 : end \neq endcalling$
$axm16 : callx \neq cally$
$axm17 : callx \neq endcalling$
$axm18 : cally \neq endcalling$
$axm19 : k \leq n$
$axm20 : \forall i, j \cdot i \in 0 \mathinner{..} n \land j \in 0 \mathinner{..} n \land i < j \Rightarrow c(j \mapsto i) = c(j-1 \mapsto i) + c(j-1 \mapsto i-1)$
END

MACHINE $binomial1$
SEES $binomial0$
VARIABLES
$vcoef$
INVARIANTS
$myteminv1vcoef \in \mathbb{N}$**EVENTS**
INITIALISATION
BEGIN
$act1 : \ vcoef :\in \mathbb{N}$
END
EVENT computing
BEGIN
$act1 : \ vcoef := c(n \mapsto k)$
END
END

MACHINE $binomial2$
REFINES $binomial1$
SEES $binomial0$
VARIABLES
$vcoef, l, vtcoefx, vtcoefy$
INVARIANTS
$inv1 : \ l \in LOC$
$inv3 : \ vtcoefx \in \mathbb{N}$
$inv4 : \ vtcoefy \in \mathbb{N}$
$inv5 : \ l \in \{callx, cally, endcalling\} \Rightarrow k \neq 0 \wedge n \neq 0 \wedge k < n$
$inv6 : \ l = cally \Rightarrow vtcoefx = c(n-1 \mapsto k-1)$
$inv7 : \ l = endcalling \Rightarrow vtcoefy = c(n-1 \mapsto k) \wedge vtcoefx = c(n-1 \mapsto k-1)$
$inv8 : \ l = end \Rightarrow vcoef = c(n \mapsto k)$
$inv2 : \ l = start \Rightarrow vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N}$

Events

INITIALISATION
BEGIN
$act1 : \ vcoef :\in \mathbb{N}$
$act2 : \ l := start$
$act4 : \ vtcoefx :\in \mathbb{N}$
$act5 : \ vtcoefy :\in \mathbb{N}$
END
EVENT computing1 **REFINES** $computing$
WHEN
$grd1 : \ k = 0$
$grd2 : \ l = start$
THEN
$act1 : \ vcoef := 1$
$act2 : \ l := end$
END

Events

EVENT computing2 **REFINES** $computing$

WHEN

$grd1 : \ l = start$

$grd2 : \ k = n$

THEN

$act1 : \ vcoef := 1$

$act2 : \ l := end$

END

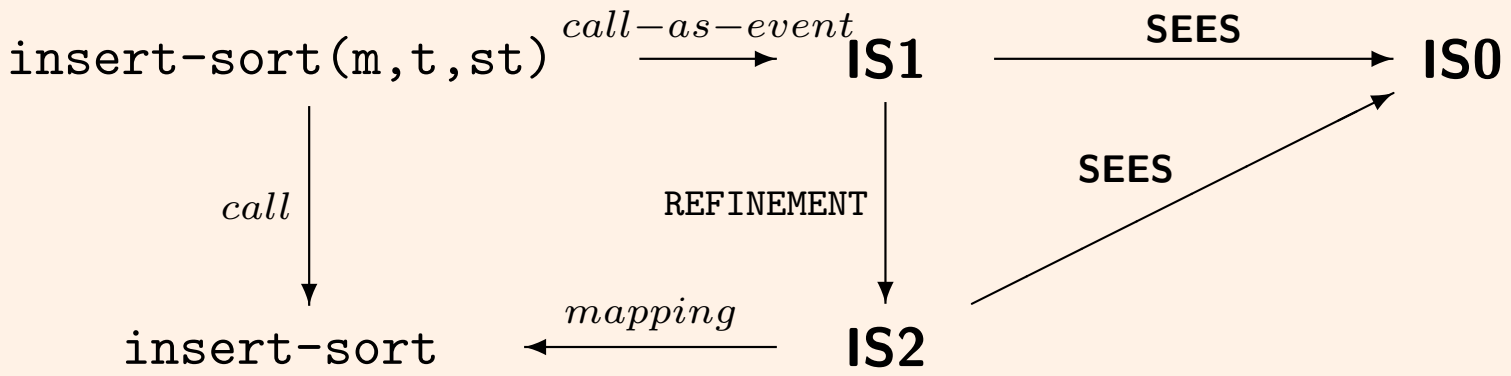
Events

EVENT callingx
  WHEN
    $grd1 : \ l = start$
    $grd2 : \ k \neq 0$
    $grd3 : \ k \neq n$
  THEN
    $act1 : \ vtcoefx := c(n-1 \mapsto k-1)$
    $act2 : \ l := cally$
  END
EVENT callingy
  WHEN
    $grd1 : \ l = cally$
  THEN
    $act1 : \ vtcoefy := c(n-1 \mapsto k)$
    $act2 : \ l := endcalling$
  END

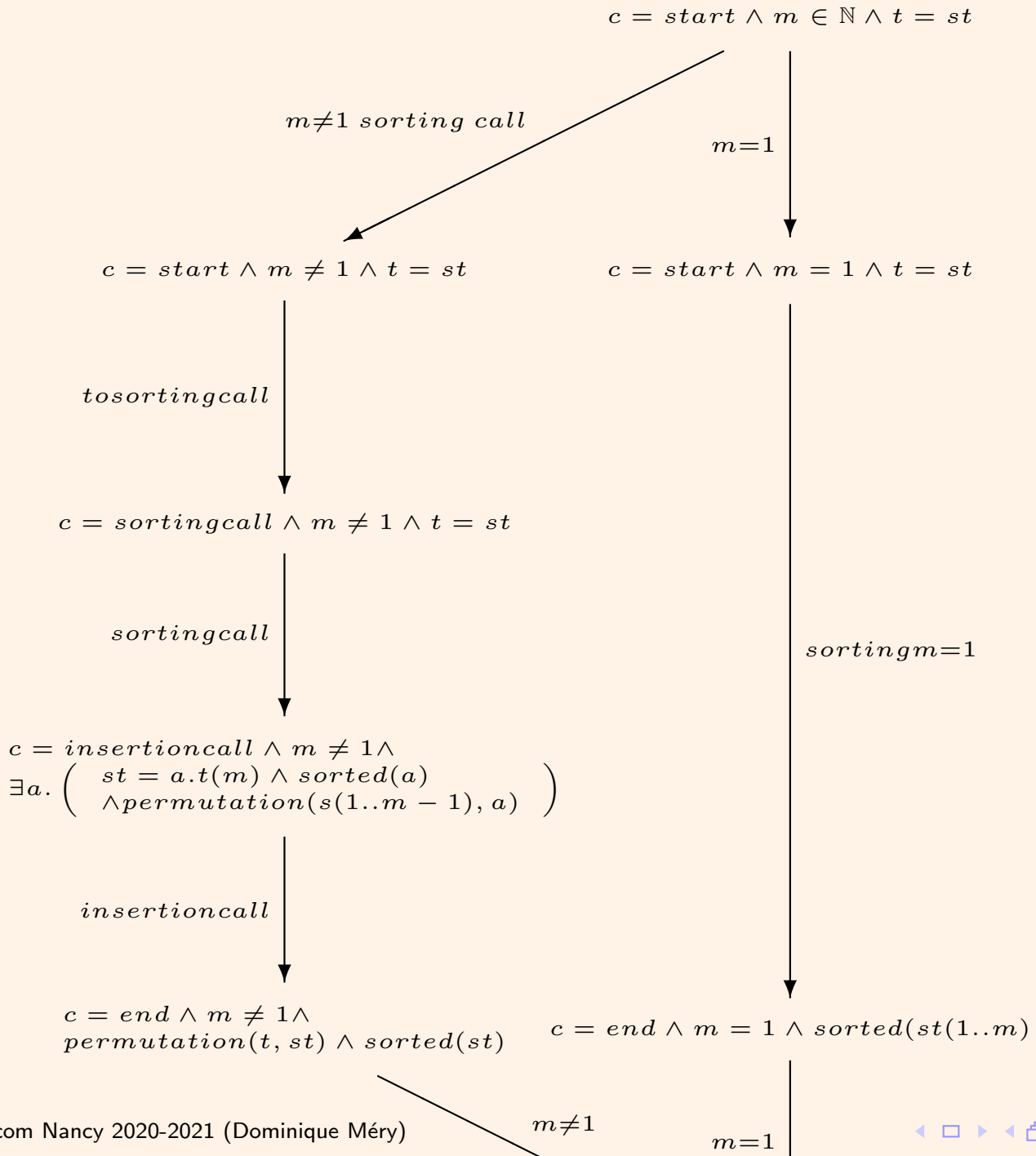
Events

EVENT computing3 **REFINES** $computing$
  WHEN
    $grd1 : \ l = endcalling$
  THEN
    $act1 : \ vcoef := vtcoefx + vtcoefy$
    $act2 : \ l := end$
  END
END

Problem 2 : Sorting by insertion



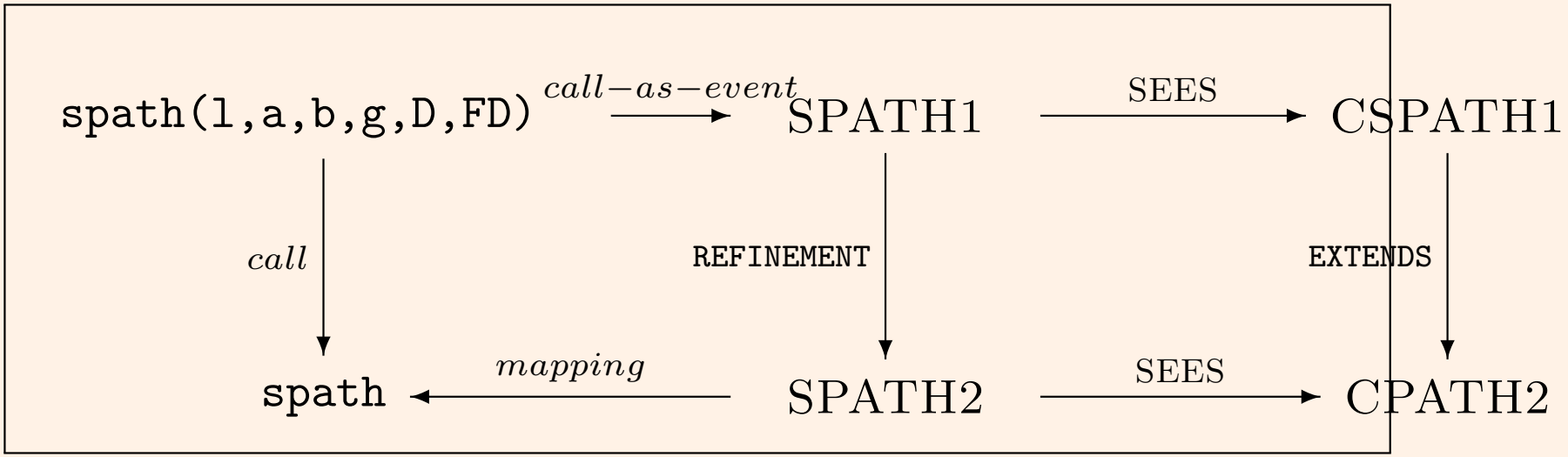
- **The problem is to sort an array t between 1 and m, where $dom(t) = 1..n$ and $m \leq n$.**
- **The sorting can be done by sorting the array from 1 to $m-1$ and then to insert the value $t(m)$ at the right position in $1..m$.**
- **The insertion event is considered as a call of procedure : the subproblem is solved in the same way by applying the guideline.**


$c = start \wedge m \in \mathbb{N} \wedge t = st$
$m \neq 1$ sorting call
$m = 1$
$c = start \wedge m \neq 1 \wedge t = st$
$c = start \wedge m = 1 \wedge t = st$
$tosortingcall$
$c = sortingcall \wedge m \neq 1 \wedge t = st$
$sortingcall$
$sortingm=1$
$c = insertioncall \wedge m \neq 1 \wedge$
$\exists a. \left(\begin{array}{l} st = a.t(m) \wedge sorted(a) \\ \wedge permutation(s(1..m-1), a) \end{array} \right)$
$insertioncall$
$c = end \wedge m \neq 1 \wedge$
$permutation(t, st) \wedge sorted(st)$
$c = end \wedge m = 1 \wedge sorted(st(1..m)$
$m \neq 1$
$m = 1$

Comments on Problem 2

- **The diagram gives the different events of the refinement model $IS2$; it contains an event called sortingcall, which is sorting the array t between the value 1 to $m-1$**
- **the event insertioncall which is inserting the value $t(m)$ at the right position in the array sorted between 1 and $m-1$.**
- **This last event can not be translated into an algorithmic expression and should be considered as defining a new problem which is the insertion of a value in a sorted array.**
- **We re-apply the guideline by starting a new development for solving the insertion problem.**
- **We use a diagram for illustrating the insertion of $t(m)$ into the values of $st(1..m-1)$.**

Problem 3 : Floyd's algorithm



- `spath` is built from events of SHORTESTPATH2
- `FD` states if the path exists
- `D` contains the cost of the minimal path, if it exists

The Event B Context

- **The distance function d is defined inductively from bottom to top according to the dynamic programming principle**
- **When $d(k \mapsto i \mapsto j) = v$: there is a path from i to j with cost v and intermediate nodes are smaller than k.**
- **The optimality property is derived from the definition of d itself,**
- $\boxed{axm1 : d \in N \times N \times N \nrightarrow \mathbb{N}}$

Axioms for the partial function d

- $axm5 : \forall i \cdot i \in N \Rightarrow 0 \mapsto i \mapsto i \in dom(d) \land d(0 \mapsto i \mapsto i) = 0$
- $axm6 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land (k-1 \mapsto i \mapsto k \notin dom(d) \lor k-1 \mapsto k \mapsto j \notin dom(d)) \end{array} \right) \\ \Rightarrow \\ \left(k \mapsto i \mapsto j \in dom(d) \land d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \right) \end{array} \right)$
- $axm7 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land k-1 \mapsto i \mapsto k \in dom(d) \\ \land k-1 \mapsto k \mapsto j \in dom(d) \\ \land d(k-1 \mapsto i \mapsto j) \leq d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \Rightarrow \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \land d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right)$
- $axm8 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land k-1 \mapsto i \mapsto k \in dom(d) \\ \land k-1 \mapsto k \mapsto j \in dom(d) \\ \land d(k-1 \mapsto i \mapsto j) > d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \Rightarrow \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \land d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right)$
- $axm9 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \notin dom(d) \\ \land k-1 \mapsto i \mapsto k \in dom(d) \\ \land k-1 \mapsto k \mapsto j \in dom(d) \end{array} \right) \\ \Rightarrow \\ k \mapsto i \mapsto j \in dom(d) \end{array} \right)$

SHORTESTPATH1 : pre/post specification

INVARIANTS

$inv1 : D \in N \times N \nrightarrow \mathbb{N}$

$inv2 : FD \in BOOL$

EVENT shortestpathOK

WHEN

$grd1 : l \mapsto a \mapsto b \in dom(d)$

THEN

$act1 : D(a \mapsto b) := d(l \mapsto a \mapsto b)$

$act2 : FD := TRUE$

END

EVENT shortestpathKO

WHEN

$grd1 : l \mapsto a \mapsto b \notin dom(d)$

THEN

$act1 : FD := FALSE$

END

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;

  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1);
       if (FD1 == 1) {
          shortestpath(l-1,a,l,g,&D2,&FD2);
          if (FD2==1) {
            shortestpath(l-1,l,b,g,&D3,&FD3);
            if (FD3==1) {
            if (D1 < D2+D3)
              {*D= D1;}
            else
              {*D=D2+D3;};
            *FD = 1;}
            else
              {*D=D1;*FD=1;}}
          else
            {*D=D1;*FD=1;}}
       else
         {
           if  ( FD2 == 1 && FD3==1) {*D=D2+D3; *FD=1;}
       else
         {*FD=0;};}
    }}
```

Proof obligations

Model	Total	Auto	Manual	Reviewed	Undischarged
CSPATH1	8	8	0	0	0
SPATH1	5	4	1	0	0
SPATH2	493	317	176	0	0
Global	506	329	177	0	0

- Proof Obligations are related to d.
- The prover provides an effective help for completing the invariant.

Naming events in the refinement

MACHINE $specmax$
SEES $control0$
VARIABLES m
INVARIANTS
$inv1 : m \in \mathbb{N}$
EVENTS
EVENT INITIALISATION
BEGIN
$act1m :\in \mathbb{N}$
END
EVENT maximum
BEGIN
$act1m : |(m' \in ran(f) \wedge (\forall j \cdot j \in 0 \,..\, i \Rightarrow m' \geq f(j)))$ **END**
END



MACHINE $refmax$ **REFINES** specmax**SEES** $control0$

VARIABLES $m, l, temp, ftemp$

INVARIANTS

$inv1 : l \in CONTROLS$

$inv2 : m \in \mathbb{N}$

$inv3 : temp \in \mathbb{N}$

$inv4 : ftemp \in \mathbb{N}$

$inv5 : l = call1 \Rightarrow i \neq 0$

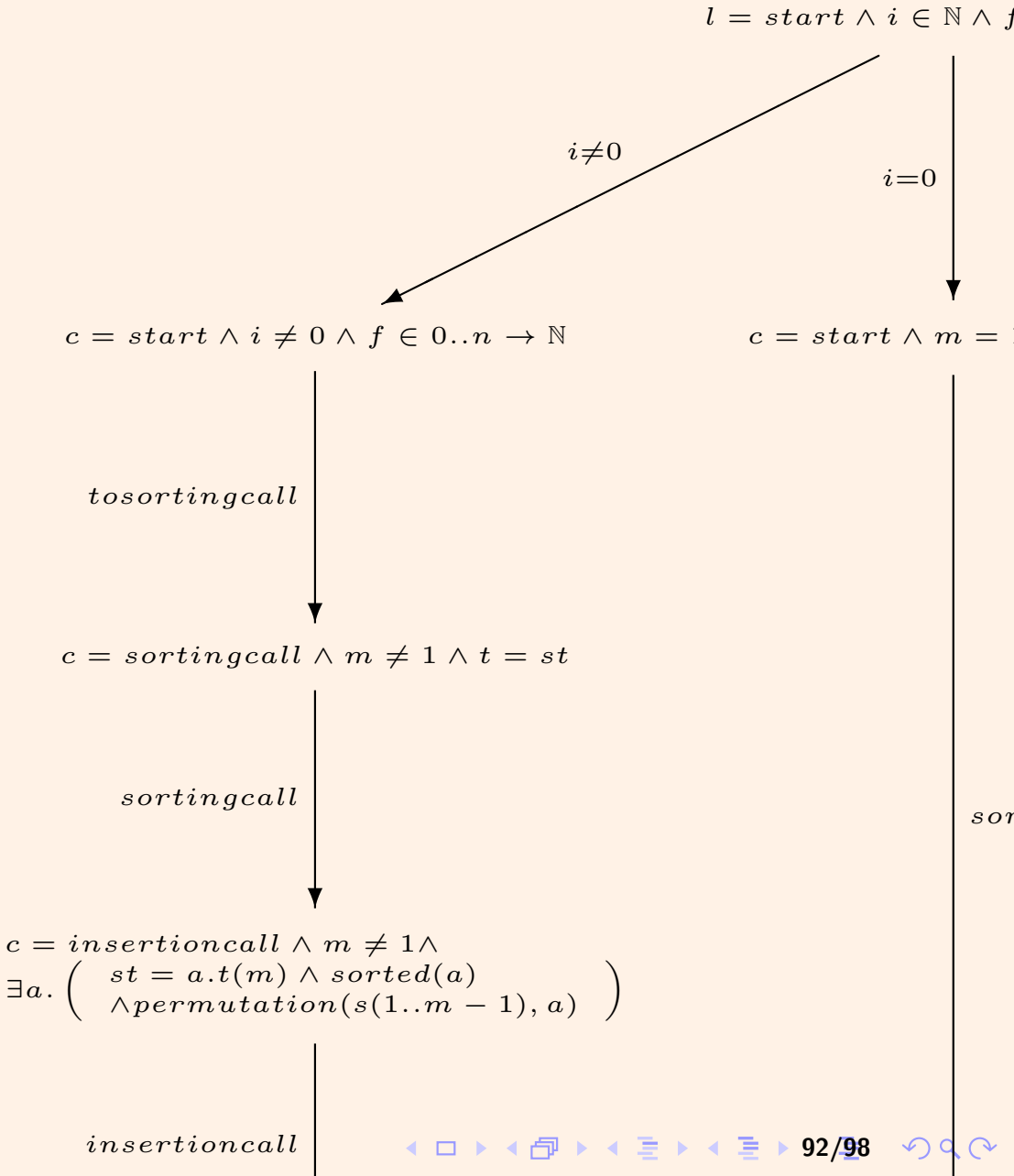
$$inv6 : l = call2 \Rightarrow \left(\begin{array}{l} temp \in ran(f) \\ \wedge \\ (\forall k \cdot k \in 0 \,..\, i-1 \Rightarrow temp \geq f(k)) \end{array} \right)$$

$$inv7 : l = call3 \Rightarrow \left(\begin{array}{l} ftemp \in ran(f) \\ \wedge \\ (\forall k \cdot k \in 0 \,..\, i \Rightarrow ftemp \geq f(k)) \end{array} \right)$$

$inv8 : l = end \Rightarrow m \in ran(f) \wedge (\forall k \cdot k \in 0 \,..\, i \Rightarrow m \geq f(k))$

Events for refmax

EVENT INITIALISATION
BEGIN
$act1 : m :\in \mathbb{N}$
$act2 : temp :\in \mathbb{N}$
$act3 : ftemp :\in \mathbb{N}$
$act4 : l := start$
END
EVENT maximum(f,n,i ;m)%i=0
REFINES maximum
WHEN
$grd1 : l = start$
$grd2 : i = 0$
THEN
$act1 : l := end$
$act2 : m := f(0)$
END
EVENT ELSE
WHEN



Events for refmax

EVENT rec%maximum(f,n,i-1 ;temp)%i/=0
  WHEN
    $grd1l = call1$
  THEN
    $act2l := call2$
    $act1temp : |(temp' \in ran(f) \wedge (\forall j \cdot j \in 0 \,..\, i-1 \Rightarrow temp' \geq f(j)))$
  END
EVENT case1
  WHEN
    $grd1l = call2$
    $grd2f(i) < temp$
  THEN
    $act2l := call3$
    $act1ftemp := temp$
  END
EVENT case2
  WHEN
    $grd1l = call2$
    $grd2f(i) \geq temp$
  THEN
    $act1ftemp := f(i)$
    $act2l := call3$
  END

Events for refmax

EVENT maximum(f,n,i ;m)%i/= 0
  REFINES maximum
  WHEN
    $grd1l = call3$
  THEN
    $act2l := end$
    $act1m := ftemp$
  END

Rules for naming events in the refinement

- **EVENT** rec%proc(x,y)%pre(proc)(x)%post(proc)(x,y) : a general form for naming an event corresponding to a recursive call
- **EVENT** call%aproc(x,y)%pre(aproc)(x)%post(aproc)(x,y) : a general form for naming an event corresponding to a call of APROC

Rules for the library

- g is a function defined in the context of the current development
- there exists a development of g as a collection of Event-B models and integrate the code and the models
- checking the application of g

Current Summary

1. Limits of the Inductive Paradigm
2. The Call-as-Event Paradigm
3. Introduction
4. Call as event
5. Transformations of Event B Models into programs
6. Case study : the binary search problem
7. Deriving the Recursive Program
8. Case studies
9. Summary

Summary on call as event

- Applying one-step refinement.
- Relating programming and modelling concepts
- Complex algorithms
- Correctness is simpler to get by using the prover.