Avoiding Programming in Safety Critical Systems

Andy Edmunds
ae2@ecs.soton.ac.uk

In the last Session ...

- We have highlighted ways to address program correctness, where errors are introduced by the *programming* activity.

- If we use automatic code generation we could improve this situation.

- Using Event-B tools (Tasking Event-B) we can generate code automatically.
  - and formal modelling can also help to highlight/remove systematic errors.

How to do this ...

- Event-B; it is modelling, not programming.

- But eventually we can produce code, and use a well trusted compiler to generate executables.

- Since we generate code automatically
  - developers focus on the design, not code.

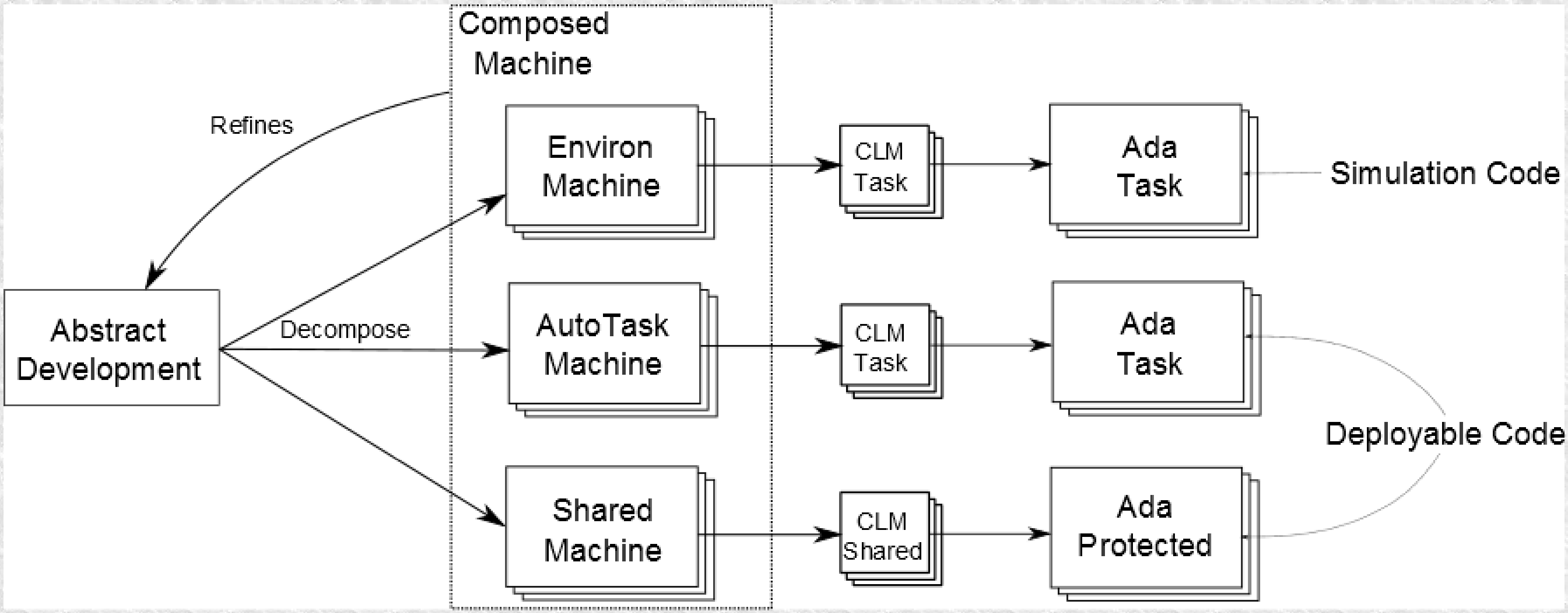
- We could still verify the code with
  - SPARKAda
  - JML
  - …. and so on, if required.

Event-B at the implementation level

- Tasking Event-B
  - Event-B models:
    - Controller Tasks (AutoTask Machine)
    - Shared Protected Objects (Shared Machine)
    - Environment Tasks (Environ Machine)

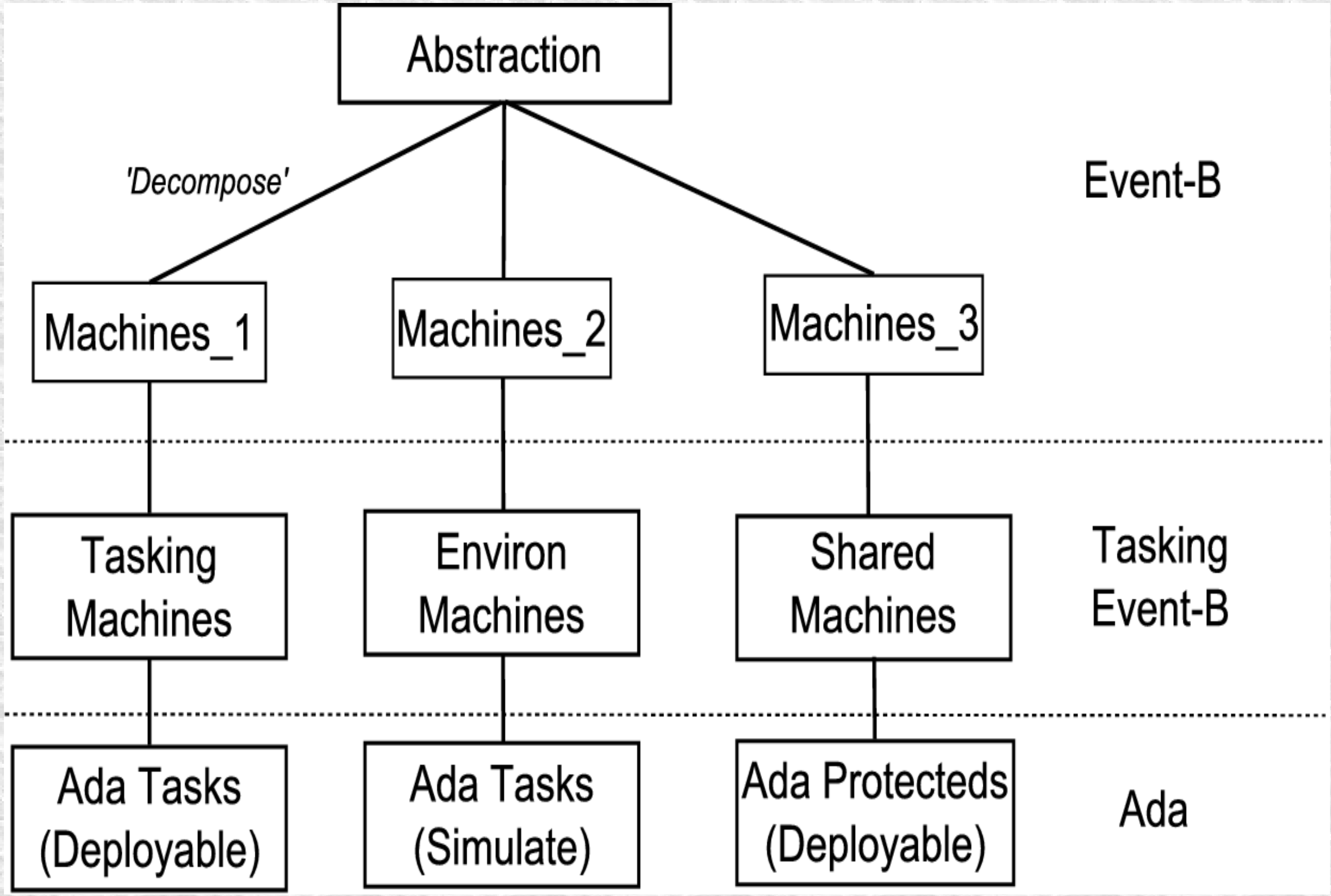
- Use Decomposition to partition the system.
  - Shared Event Style
  - Shared Events model communication, between
    - Controller tasks and Environment tasks.
    - Controller tasks and Protected Objects.
    - Environment tasks and Protected Objects.

Common Language Metamodel (CLM)



Simulation code from the Environ Machines.
Deployable Code from AutoTask and Shared Machines.

What the User Sees!



Targets for Translation ...

Targets: Ada, OpenMP C, FMI C, Java ….

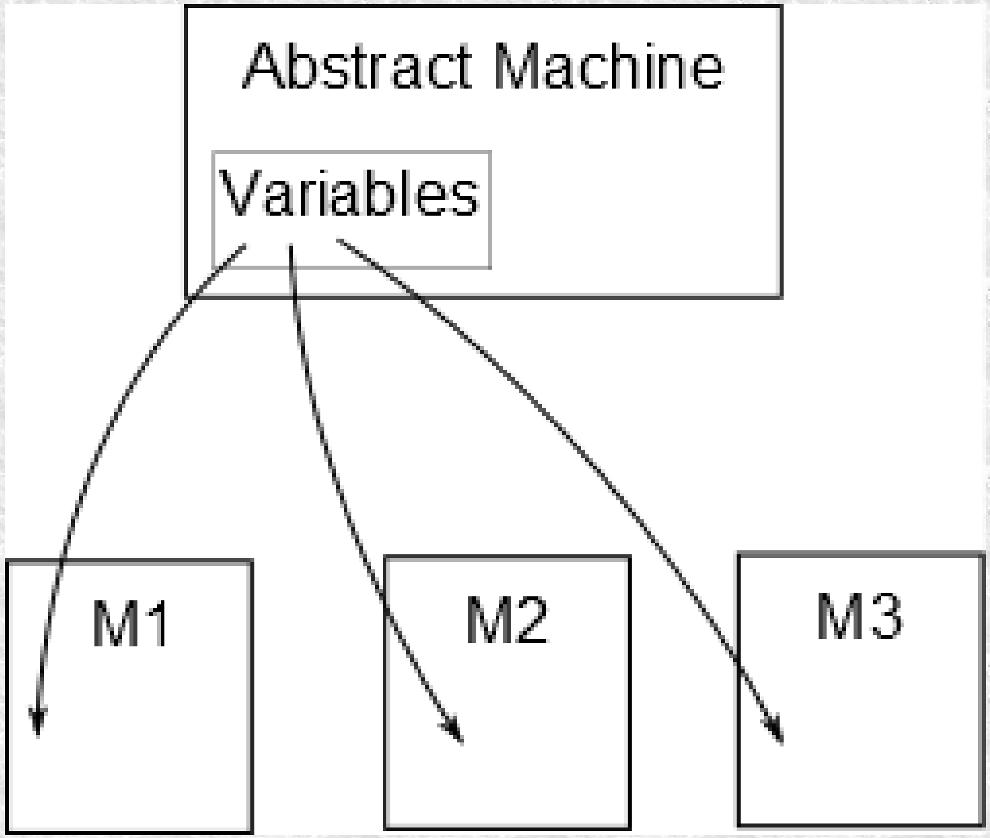
- The approach is suitable for
    - single threaded implementations.
    - multi-tasking implementations (using decomposition).
    - not currently OO, but can be coded.
- Implementable controller code environment simulation.

New Language Extensions

- specify mathematical language translation in a theory.
- hard code other parts in a new plug-in.

Shared Event Decomposition

Tool-driven decomposition



Preparing for Decomposition

A Problematic Decomposition

Cannot
Decompose !!

Machine
M0

Vars A: *x*, B: *x*
Evt =
A := B

Refines

Composed Machine

Machine
M1

Vars A: *x*
Evt =
A := ?

Machine
M2

Vars B: *x*
Evt =
? := B

Preparing for Decomposition

Introduce Parameters

Machine M1

Vars A: *x*, B: *x*

Evt = ANY **p**

WHEN **p** = B

THEN A := **p**

Refines M0

Was A := B

A Model of Communication

Machine M1

Vars A: *x*, B: *x*
Evt = ANY **p**
WHEN **p** = B
THEN A := **p**

Decompose

Refines

Composed Machine

Machine M2

Vars A: *x*
Evt = ANY **p**
WHEN **p** = B & **p**: ***x***
THEN A := **p**

Machine M3

Vars B: *x*
Evt = ANY **p**
WHEN **p** = B & **p**: ***x***
THEN SKIP

A Model of Communication

Incoming parameter

Outgoing parameter

Composed Machine

Machine M2

Vars A: *x*
Evt = ANY **p?**
WHEN **p** = B & **p:** ***x***
THEN A := **p**

Machine M3

Vars B: *x*
Evt = ANY **p!**
WHEN **p** = B & **p:** ***x***
THEN SKIP

An Implementation of the Communication

Incoming parameter

Outgoing parameter

Machine M2

Vars A: *x*
Evt = ANY **p?**
WHEN **p** = B & **p: *x***
THEN A := **p**

Machine M3

Vars B: *x*
Evt = ANY **p!**
WHEN **p** = B & **p: *x***
THEN SKIP

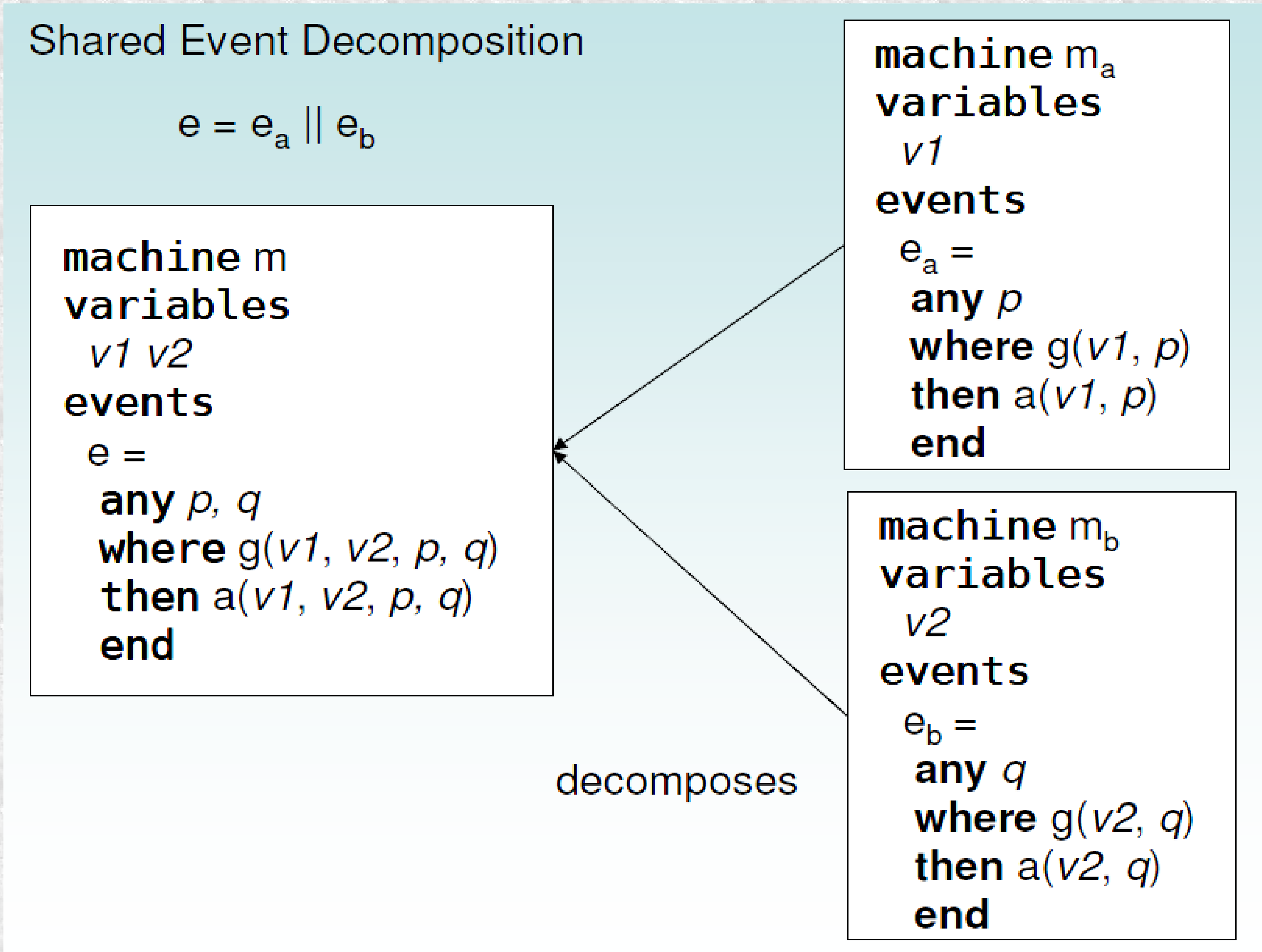
subroutine

Evt(p: *x*){
A := p
}

call

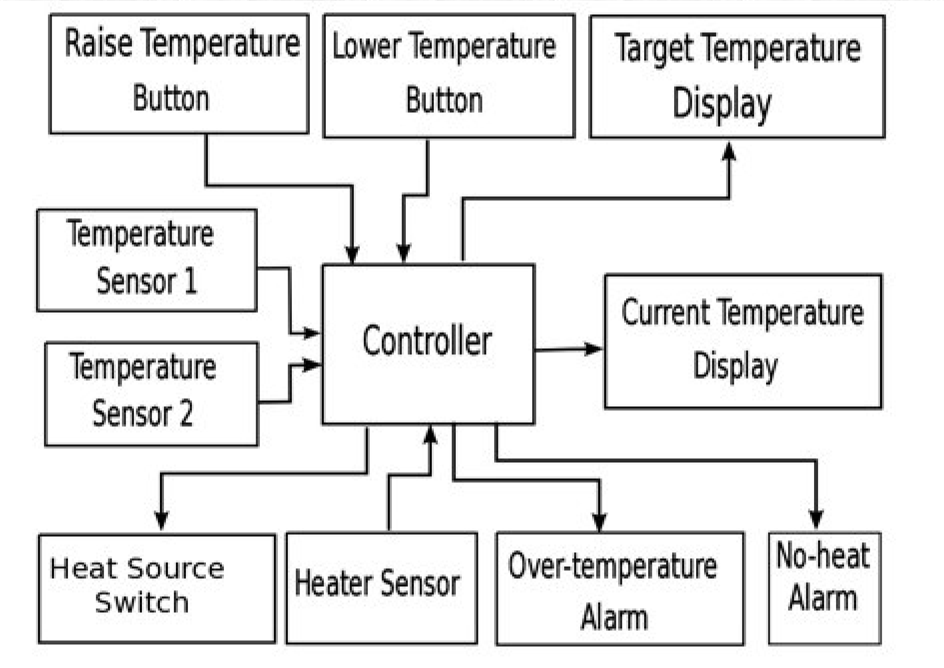
Evt(B);

Event 'Synchronization'



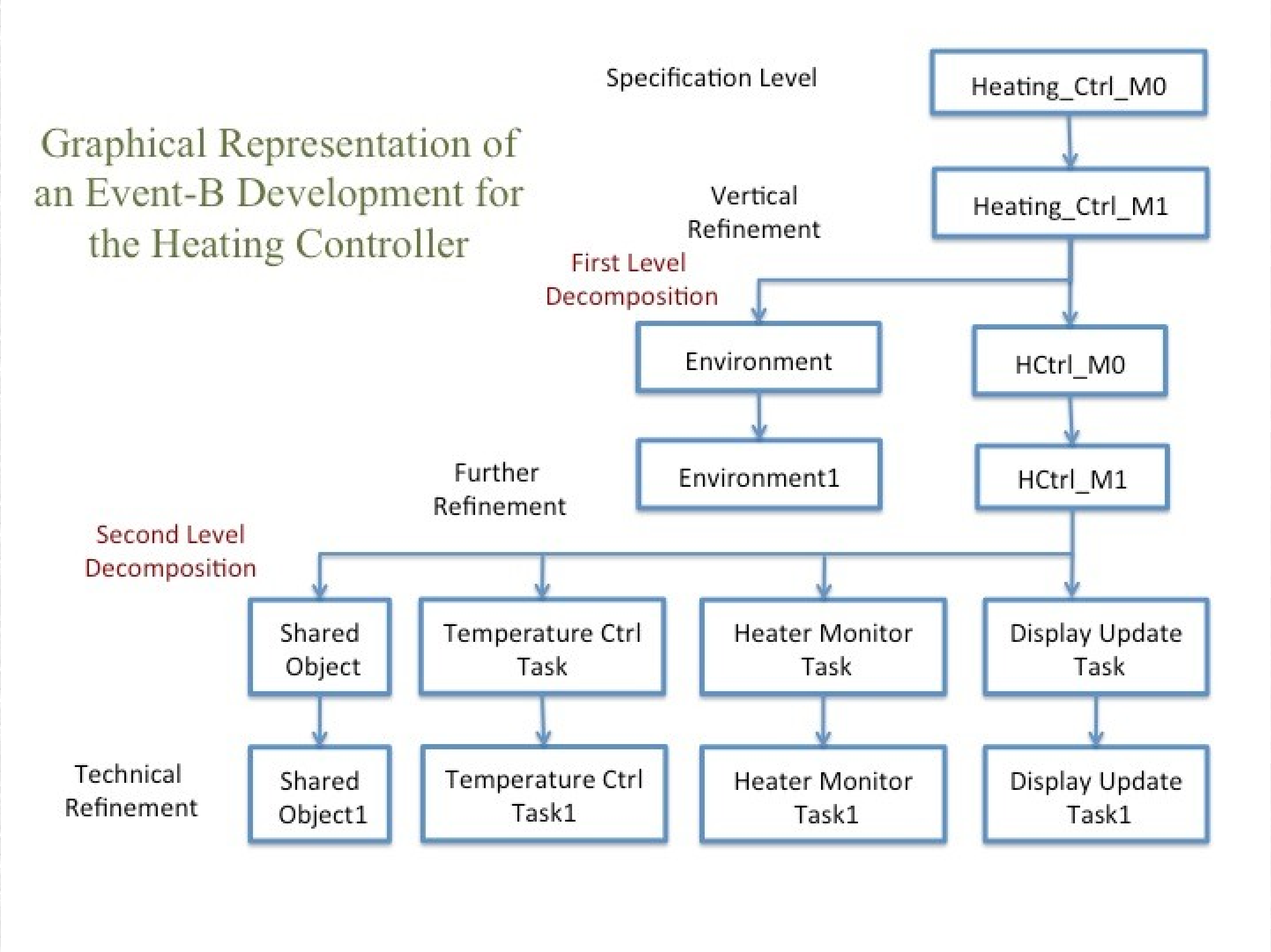
Heater Controller Example

Controller vs Environment



Heater Controller Example

Another View



Tasking Event-B

Adds 'Tasking' Implementation Information to Event-B

```
TaskBody ::=
  TaskBody ; TaskBody
  | if Event
    (elseif Event)*
    else Event
  | do Event [finally Event]
  | Event
  | output String Variable

  Event ::= String

  Variable ::= String
```

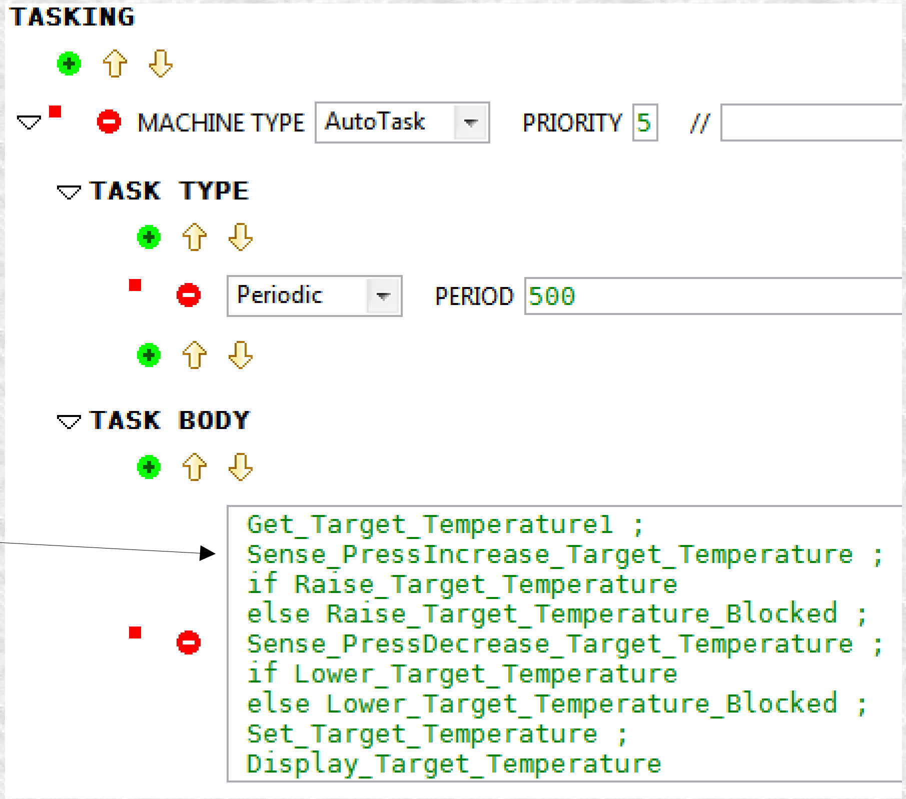
Task Body Syntax:

- Allows use of Branches, Sequence and Loops.
- Has an 'Output' to console.

A Tasking Machine

Implementation level Specification

AutoTasks Machines and Environ Machines



Events:

Used in a
Sequence,
Branch,
Loop,
Output

Tasking Event-B

restrictions

AutoTasks do not communicate with each other.

Communicate through Shared Machines.

No nesting in the Tasking Event-B syntax.

One machine per 'Object'.

...

'in'/ 'out' annotations

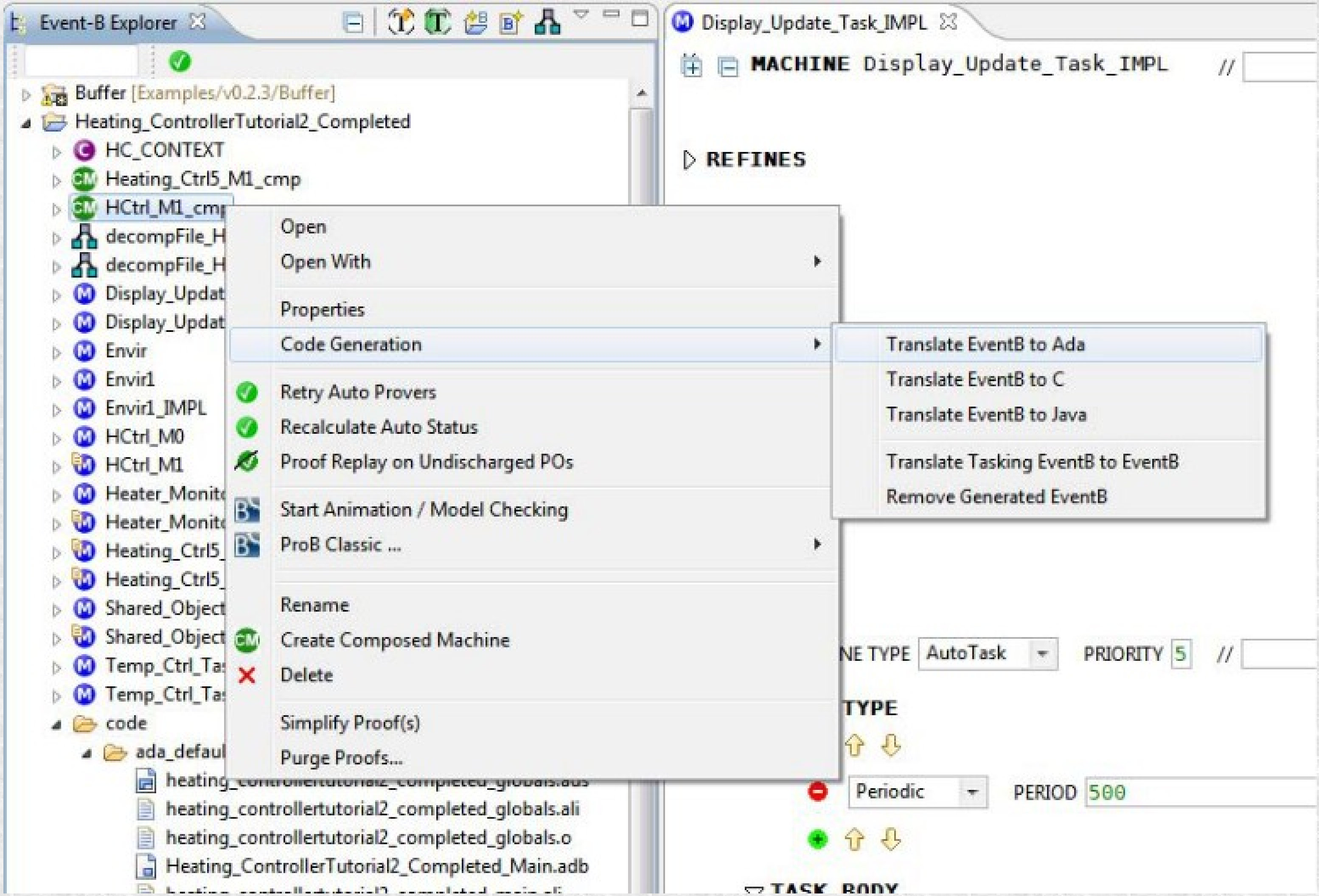
synchronization

```
Get_Target_Temperature1  ≙
     COMBINES EVENT
  Shared Object IMPL.Get Target Temperature1 ||
  Display_Update_Task_IMPL.Get_Target_Temperature1
REFINES
  Get_Target_Temperature1
```

Parameter direction

```
Get_Target_Temperature1  ≙
REFINES
     Get_Target_Temperature1
ANY
     in tm
WHERE
     grd1  :  tm ∈ ℤ  TYPING
THEN
     act1  :  cttm1 ≔ tm
END
```

Code Generation



Generated Code

In the Display Task:

```
Shared_Object: Shared_Object_IMPL; ...
task body Display_Update_Task_IMPL is
  cttm1 : Integer := 0;
  period: constant Time_Span := To_Time_Span(0.5);
  nextTime: Time := clock + period;
  begin
    loop
      delay until nextTime;
          Shared_Object.Get_Temperature1(cttm1);
  ...
```

In the Protected Object:

```
procedure Get_Temperature1(tm:  out Integer) is
  begin
    tm := cttm;
  end Get_Temperature1;
```

Extending Event-B: Adding new types and Translations.

Extend the Event-B mathematical language
- using the Theory Plug-in

Theories are used to define new
- datatypes
- operators
- rewrite rules
- inference rules

We also use it for code generation,
- to translate predicates and expressions.

Defining a Translator:
From Event-B to a 'new' Target Language

```
THEORY AdaRules
TRANSLATOR Ada
Metavariables • a ∈ ℤ, b ∈ ℤ, c ∈ Q, d ∈ Q
Translator Rules
     ...
     trns2:   a − b ↦ a - b
     trns9:   c = d ↦  c = d
     trns19:  a ≠ b ↦  a /= b
     trns21:   a mod b    ↦  a mod b
     trns22:   ¬$c    ↦   not($c)
     trns23:   $c ∨ $d  ↦   ($c) or ($d)
     trns24:   $c ∧ $d  ↦   ($c) and ($d)
     trns25:   $c ⇒ $d  ↦  not($c) or ($d)
Type Rules
     typeTrns1:   ℤ     ↦ Integer
     typeTrns2:   BOOL  ↦ boolean
```

Adding new Types

THEORY Array
TYPE PARAMETERS T
OPERATORS

•**array** : $\mathrm{array}(s : \mathbb{P}(T))$
direct definition
$\mathrm{array}(s : \mathbb{P}(T)) \triangleq \{\, n, f \cdot n \in \mathbb{N} \wedge f \in 0 \,..\, (n-1) \rightarrow s \mid f \,\}$

•**arrayN** : $\mathrm{arrayN}(n : \mathbb{Z}, s : \mathbb{P}(T))$
well-definedness condition $n \in \mathbb{N} \wedge \mathrm{finite}(s)$
direct definition
$\mathrm{arrayN}(n : \mathbb{Z}, s : \mathbb{P}(T)) \triangleq \{\, a \mid a \in \mathrm{array}(s) \wedge \mathrm{card}(s) = n \,\}$

Adding a Translation for the new Type

(In a theory)

```
•update      :   update(a : ℤ↔T, i : ℤ, x : T)
 …
•lookup   :   lookup(a : ℤ↔T, i : ℤ)
 …
•newArray   :   newArray(n : ℤ, x : T)
 …

TRANSLATOR Ada
Metavariables s ∈ ℙ(T), n ∈ ℤ, a ∈ ℤ↔T, i ∈ ℤ, x ∈ T
Translator Rules
     trns1   :   lookup(a,i) ↦ a(i)
     trns2   :   a = update(a,i,x) ↦ a(i) := x
     trns3   :   newArray(n,x) ↦ (others => x)
Type Rules
     typeTrns1   :   arrayN(n,s) ↦ array (0..n-1) of s
```

Using a new Type

```
VARIABLES
    cbuf     private ›
    a    private ›
    b    private ›
INVARIANTS
    inv1:   cbuf ∈ arrayN(maxbuf,ℤ) not theorem TYPING Typing ›
    inv2:   a ∈ ℤ not theorem TYPING Typing ›
    inv3:   b ∈ ℤ not theorem TYPING Typing ›
    inv4:   a ∈ 0‥maxbuf−1 not theorem TYPING NonTyping ›
    inv5:   b ∈ 0‥maxbuf not theorem TYPING NonTyping ›
    inv6:   ∀i· i∈ (0‥seqSize(abuf)) ⇒ prj2(abuf)(i) = cbuf((a+i) mo
EVENTS
    INITIALISATION: internal not extended ordinary ›
        THEN
            act1:   cbuf ≔ newArray(maxbuf,0) ›
            act2:   a ≔ 0 ›
            act3:   b ≔ 0 ›
        END
```

And finally … (almost)

- Writing code for Safety Critical Systems is hard
  - Coding is complemented with extended static-checking, comprehensive testing, FMs, model-checking, SAT/SMT etc …
  - Use safe language subsets.
  - Place restrictions on the implementation.
    - esp. for timing, and concurrency.
  - Employ a rigorous engineering process.

Certification is a gating factor !

… Summing Up

If you write code manually
- much of the development effort is invested in eliminating coding errors.

With automatic code generation
- The modelling process helps to eliminate systemic errors.
- If the translator is 'trusted', coding errors should be absent.

- Certifying a translator is possible, but expensive.