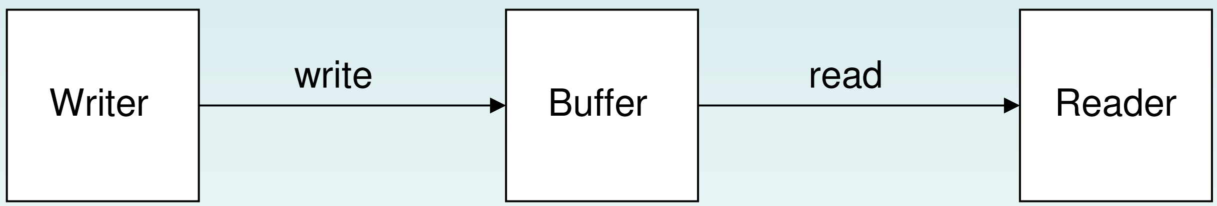
Tasking Event-B for Code Generation

Andy Edmunds ae2@ecs.soton.ac.uk
and
Michael Butler mjb@ecs.soton.ac.uk

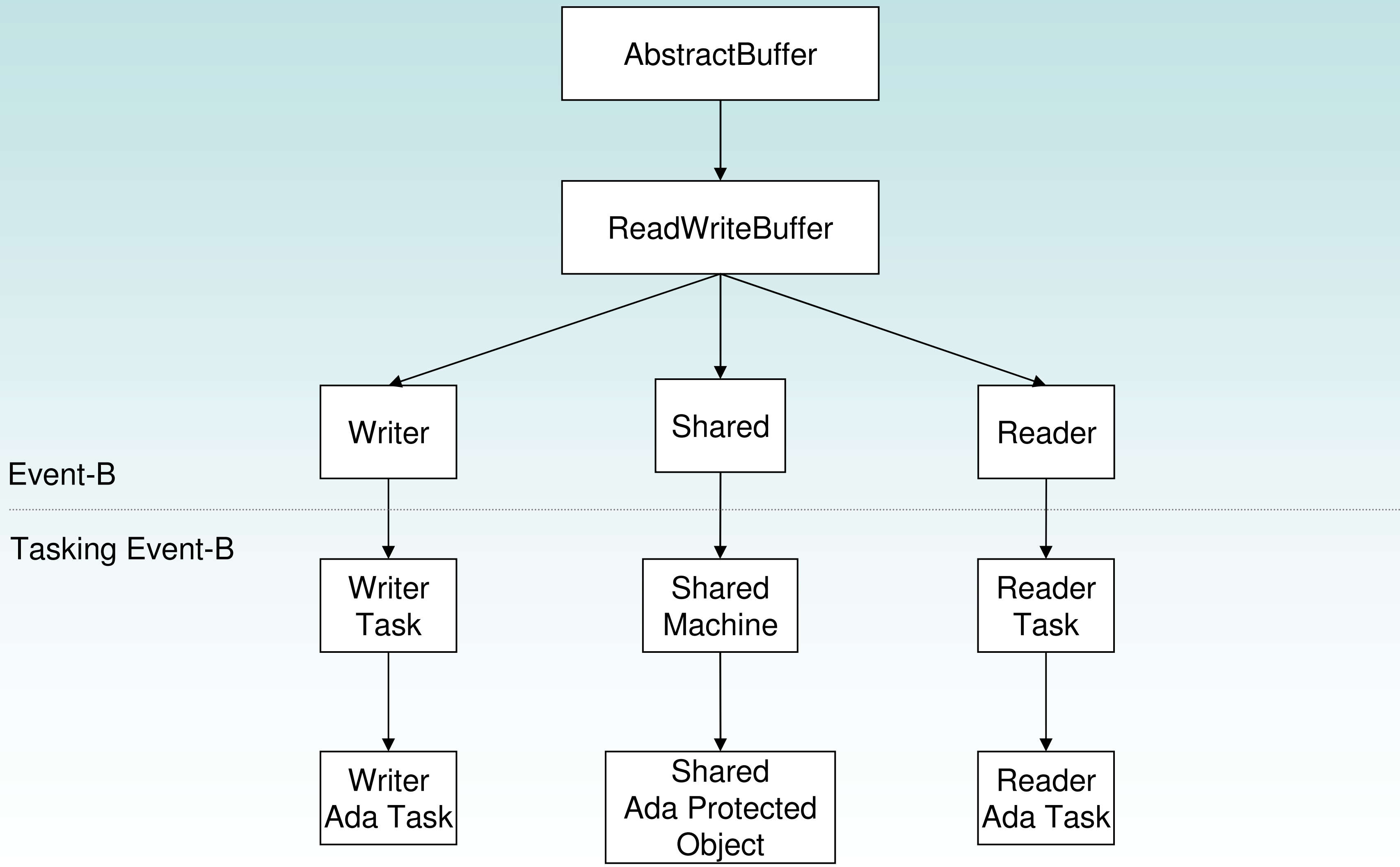
One-Place Buffer Example

“write a single NAT value to buffer”



“read the value from the buffer”

The Route To Code



UNIVERSITY OF Southampton
School of Electronics and Computer Science

Abstract Machine

```
machine AbstractBuffer

variables buff wVal rVal wCount sCount

…

  event write
    where
      buff < 0
    then
      buff ≔ wVal
      sCount ≔ sCount + 1
      wCount ≔ sCount + 1
  end
```

"buff is initially -1"

Parameterised for Decomposition (i)

```
machine ReadWriteBuffer
refines AbstractBuffer

variables buff wVal rVal wCount
sCount

…

  event write refines write
    any p1  p2
    where
      p1 = wVal
      p2 = sCount + 1
      buff < 0
    then
      buff ≔ p1
      sCount ≔ sCount + 1
      wCount ≔ p2
  end
```

was buff ≔ wVal

“The parameter wVal”

University of Southampton — School of Electronics and Computer Science

Parameterised for Decomposition (ii)

```
machine ReadWriteBuffer
refines AbstractBuffer

variables buff wVal rVal wCount
  sCount

…

  event write refines write
    any p1  p2
    where
      p1 = wVal
      p2 = sCount + 1
      buff < 0
    then
      buff ≔ p1
      sCount ≔ sCount + 1
      wCount ≔ p2
  end
```

was wCount ≔ sCount + 1

“The parameter: sCount + 1”

Decomposed Machines

```
machine Writer

variables wVal wCount

…

 event Twrite refines write
  any outAP  inAP
  where
   inAP ∈ ℤ
   outAP ∈ ℤ
   outAP = wVal
  with
   p1 = outAP
   p2 = inAP
  then
   wCount ≔ inAP
 end
```

```
machine Shared

variables buff sCount

…

 event Swrite refines write
  any inFP outFP
  where
   outFP ∈ ℤ
   inFP ∈ ℤ
   outFP = sCount + 1
   buff < 0
  with
   p1 = inFP
   p2 = outFP
  then
   buff ≔ inFP
   sCount ≔ sCount + 1
 end
```

- Refinement: renaming is for clarity,
  - parameters will be ‘paired’ in order of declaration for translation.

Adding the Tasking Constructs

```
tasking machine Writer
priority 5
tasktype periodic(500)
variables wVal wCount

…

body
  w1: ◁ Twrite || Shared.Swrite ▷ ;
  w2: …

  event sync Twrite refines write
    any
      actualOut outAP
      actualIn inAP
    where
      inAP ∈ ℤ
      outAP ∈ ℤ
      outAP = wVal
    then
      wCount ≔ inAP
  end
```

```
machine Shared

variables buff sCount

  …

  event Swrite
    any
      formalIn inFP
      formalOut outFP
    where
      outFP ∈ ℤ
      inFP ∈ ℤ
      outFP = sCount + 1
      buff < 0
    then
      buff ≔ inFP
      sCount ≔ sCount + 1
  end
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Ada Code - Task

tasking machine Writer
priority 5
tasktype periodic(500)
variables wVal wCount …

…

body
w1: ◁ Twrite || Shared.Swrite ▷ ;

w2: TcalcWVal;

Output(“wVal is ”, wVal)

```
task WriterTsk is
  pragma Priority(5);
end WriterTsk;
task body WriterTsk is
  t: Time;
  period: constant Time_Span := To_Time_Span(0.5);
  wVal : Integer := 5;
  wCount : Integer := 0;
  wCount2 : Integer := 0;
  procedure Twrite is
  begin
    wCount2 := wCount2 + 1;
  end;
  procedure TcalcWVal is
  begin
    wVal := wVal * 2;
  end;
begin
  loop
    t := clock;
    Twrite;
    sharedtskInst.Swrite(wVal, wCount);
    TcalcWVal;
    put("wVal ="); put(wVal); New_Line ;
    delay until t + period;
  end loop;
end WriterTsk;
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Ada Code – Protected Body

machine Shared

variables buff sCount

…

event Swrite
any
`formalIn` *inFP*
`formalOut` *outFP*
where
$outFP \in \mathbb{Z}$
$inFP \in \mathbb{Z}$
$outFP = sCount + 1$
$buff < 0$
then
$buff \coloneqq inFP$
$sCount \coloneqq sCount + 1$
end

```
package body SharedTskPkg is
  protected body SharedTsk is
    entry Swrite(inFP: in Integer; outFP: out Integer) when buff < 0 is
    begin
      outFP := sCount + 1;
      buff := inFP;
      sCount := sCount + 1;
    end;
    entry Sread(outFP: out Integer) when buff >= 0 is
    begin
      outFP := buff;
      buff := -1;
    end;
  end SharedTsk;
end SharedTskPkg;
```

"Conditional waiting in implementations"

The Resulting Event-B model

```
machine Writer refines Writer
sees autoGenCTX_Writer

variables
  wVal wCount wCount2 Writer_pc

Invariants
  …
  Writer_pc ∈ Writer_pc_Set

events
  event Twrite refines TWrite
    any outAP  inAP
    where
      inAP ∈ ℤ
      outAP ∈ ℤ
      outAP = wVal
      Writer_pc = w1
    then
      wCount ≔ inAP
      Writer_pc ≔ w2
  end
```

```
machine Shared

variables buff sCount

invariants
  … // various typing

  event Swrite refines write
    any inFP outFP
    where
      outFP ∈ ℤ
      inFP ∈ ℤ
      outFP = sCount + 1
      buff < 0
    then
      buff ≔ inFP
      sCount ≔ sCount + 1
  end
```

University of Southampton School of Electronics and Computer Science

The Resulting Event-B model

```
machine Writer refines Writer
sees autoGenCTX_Writer

variables
  wVal wCount wCount2  write

Invariants
  …
 write ∈ BOOL

events
  event Twrite refines Twrite
    any outAP  inAP
    where
      inAP ∈ ℤ
      outAP ∈ ℤ
      outAP = wVal
      write = TRUE
    then
      wCount ≔ inAP
      write ≔ FALSE
  end
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Tasking Event-B

Tasking Event-B is an extension of Event-B,

- where we have attempted to provide a ‘streamlined’ approach,
  - with a small semantic gap between the Event-B abstract development and the implementation specification.
- using decomposition to handle complexity, and ultimately, a tasking (implementation) specification for code generation.
- currently we have translators that map to Ada, and map to an Event-B model; i.e. the model of the implementation.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Tasking Event-B

Targeting implementations with,

- Multi-tasking capability
- Tasking
  - for shared memory systems.
  - using interleaving atomic executions.
- Sharing data between tasks using ‘protected objects’,
  - using atomic procedure calls,
  - with blocking behaviour.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Modelling/Specifying Tasks

Tasking Machines are an abstraction of,
- Ada tasks
- Java threads
- pthreads etc.

Shared Machines are an abstraction of,
- monitors,
- protected objects etc.

Tasking Machine Algorithmic constructs,
- Loop,
- Branch,
- Sequence,
- Synchronisation.

Tasking Machine Implementation Specifics:
- Task type, task priority.

University of Southampton School of Electronics and Computer Science

Modelling Mutually Exclusive Access

Tasking Machines do not communicate directly with each other,

- communication is only with Shared Machines.
- Shared machines are just Event-B machines.

Protected Object's updates,

- modelled by Shared Event Composition.

Events can map to,

- a subroutine definition.
- part of a subroutine call.
- part of a loop /branch implementation.

Synchronized Events

'Synchronisation' *e* of a local and remote events
decomposition semantics; i.e. guards are conjoined.
parallel updates.

$$e = e_l \parallel_e e_r$$

A **local event** e_l belongs to a **tasking machine**, and only updates the task's state.

A **remote event** e_r belongs to a **shared machine**, and only updates a shared machine's state.

Implementation of Synchronized Events

$e_l \; \|_e \; e_r$

implemented as: $e_l()$; $s.e_r(a_1..a_n)$

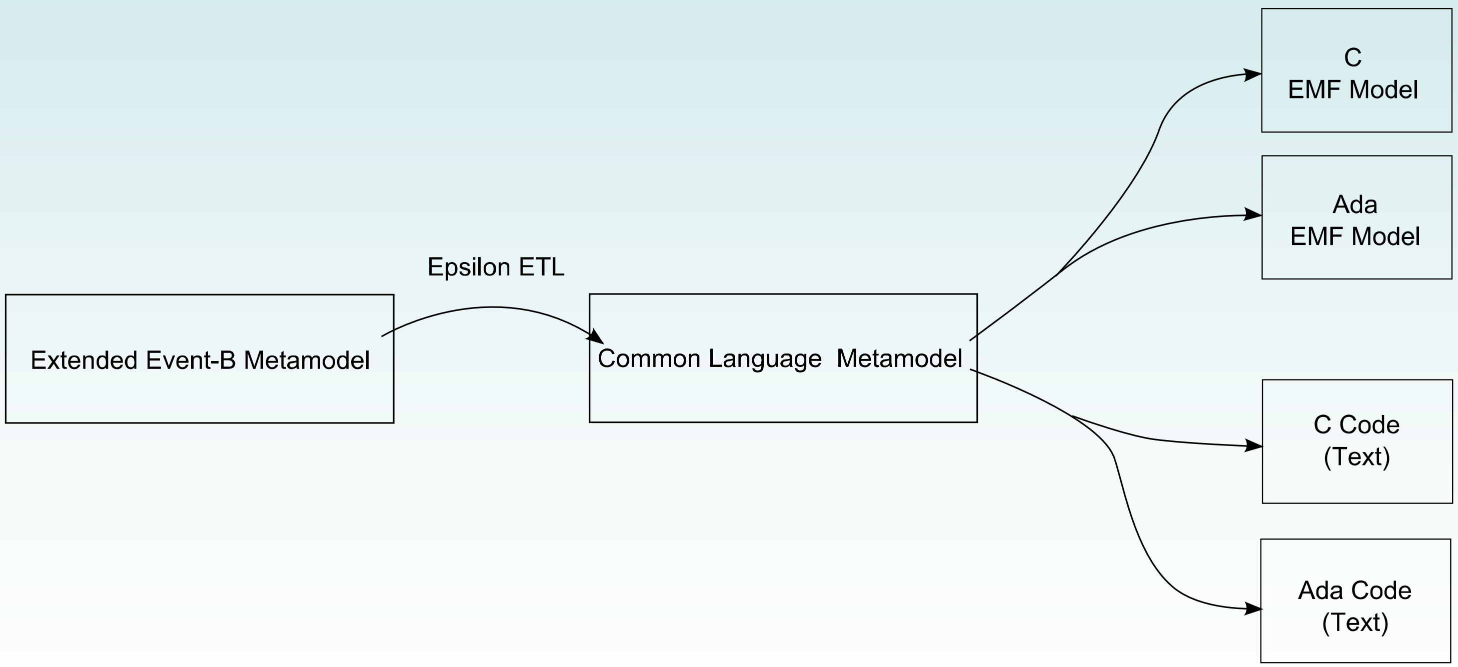
where:
$e_l()$ is a local call derived from an event with no blocking guards.

$s.e_r(a_1..a_n)$ is a call to a shared machine instance ‘s’.
e_r may have blocking guards.
e_r may have *in* or *out* parameters derived from the guards of e_l and e_r .

Implementation - Abstraction

Common Language Metamodel (IL1)

An abstraction of various programming constructs.



Common Language Metamodel

Facilitates translation to multiple targets e.g. Ada/C etc.

Make use of Model-to-Model translation tools.

‘Invisible’ to the user.

Tasking Event-B Notation V1

TaskBody ::=
  TaskBody ; *TaskBody*
  | **if** EventSynch **end**
    [**else if** EventSynch **end**] …
    [**else** EventSynch **end**]
  | **do** EventSynch [**finally** EventSynch] **od**
  | EventSynch
  | Output

More details @
http://wiki.event-b.org/images/TranslationV20100722.pdf

Tasking Event-B Notation V1

tasktype ::= Periodic(p) | One Shot | Repeating | Triggered

priority(n)

Sequence

- modelled using an abstract program counter which,
  - may be derived from labelled events,
  - may use of boolean flags (where feasible).

UNIVERSITY OF Southampton School of Electronics and Computer Science

Branching

IF e_1 END
[ELSE IF e_x END]
[ELSE e_n END]

$$e_i = e_{ir} \parallel_e g_{il} \rightarrow a_{il} \qquad \text{where } i : 1..x..n$$

… in task maps to:

if(g_{1l}) then *body* end
[else if(G_{xl}) then *body* end]
[else *body* end]

body = e_{ir}(); a_{il}

where G_{il} is derived from g_{il}

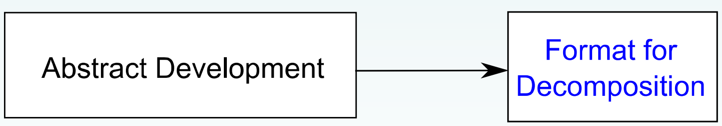
… in protected maps to:

subroutine e_{ir}() is
begin a_{ir} end

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Before Decomposition

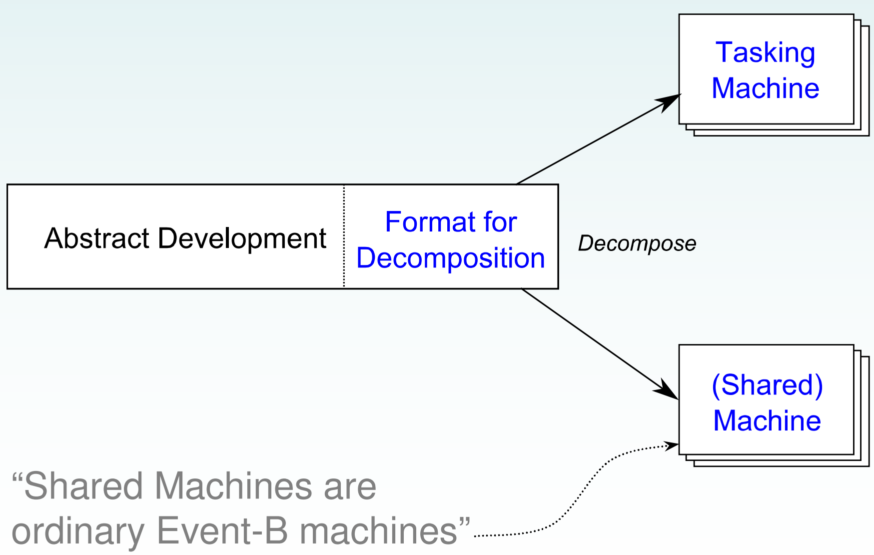
1. Specify the abstract development.
2. Prepare for decomposition. For each event,
   - identify and specify parameters (using event guards),
   - substitute expressions by parameters, in event actions, where applicable.



Decompose

3. Allocate variables to machines during shared event decomposition (typically to multiple Tasking/ Shared Machines)

4. Complete the decomposition.

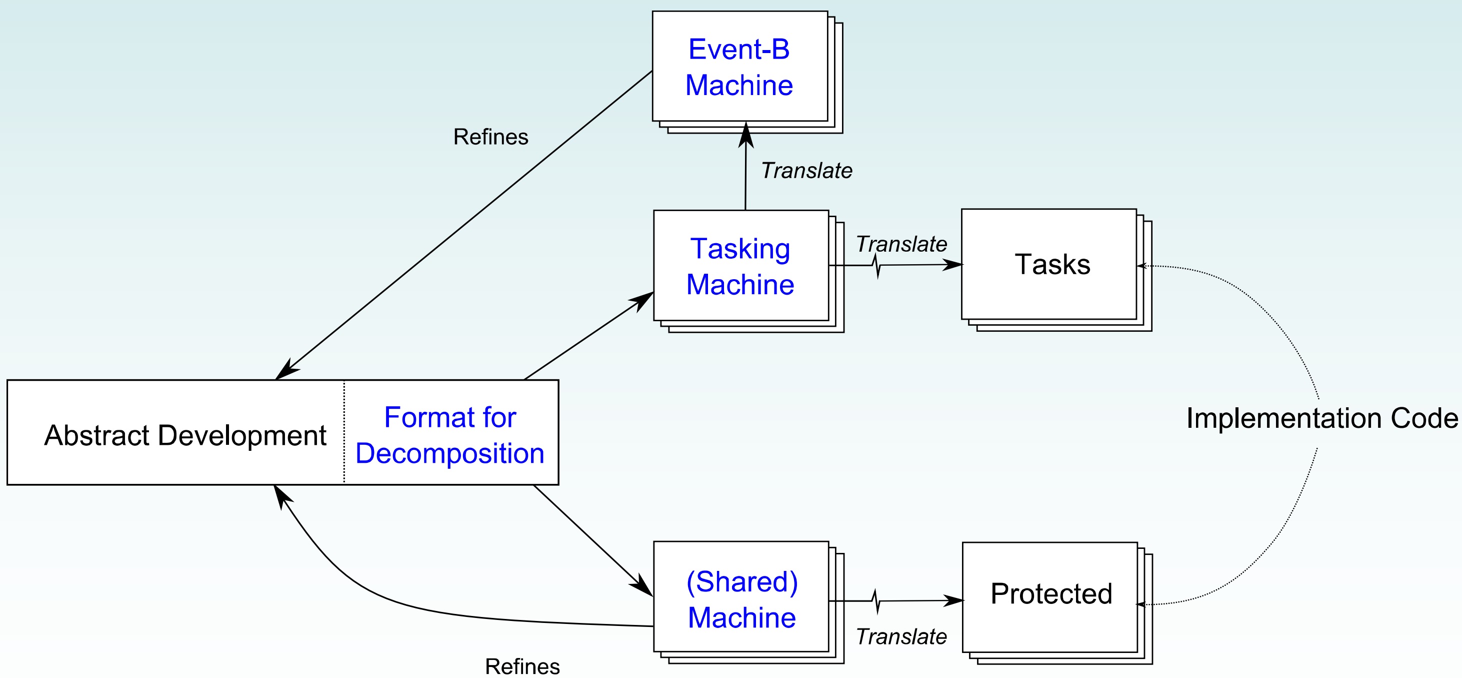


Translation

5. Copy, or reference, decomposed machines for use in the tasking model.

6. Add Tasking Constructs to create Tasking and Shared Machines.
   e.g. synch, loop, branch, sequence, priority, etc.

7. Automatic Translation to Code and Event-B



UNIVERSITY OF Southampton
School of Electronics and Computer Science

Summing Up

This approach,

- extends Event-B with Implementation Constructs.
- uses small steps which are easy to reason about.
- makes use of decomposition.
- generates code.

Need:

- to work on Documentation/Guidelines.
- a better user interface.
- more automation.