University of Southampton School of Electronics and Computer Science

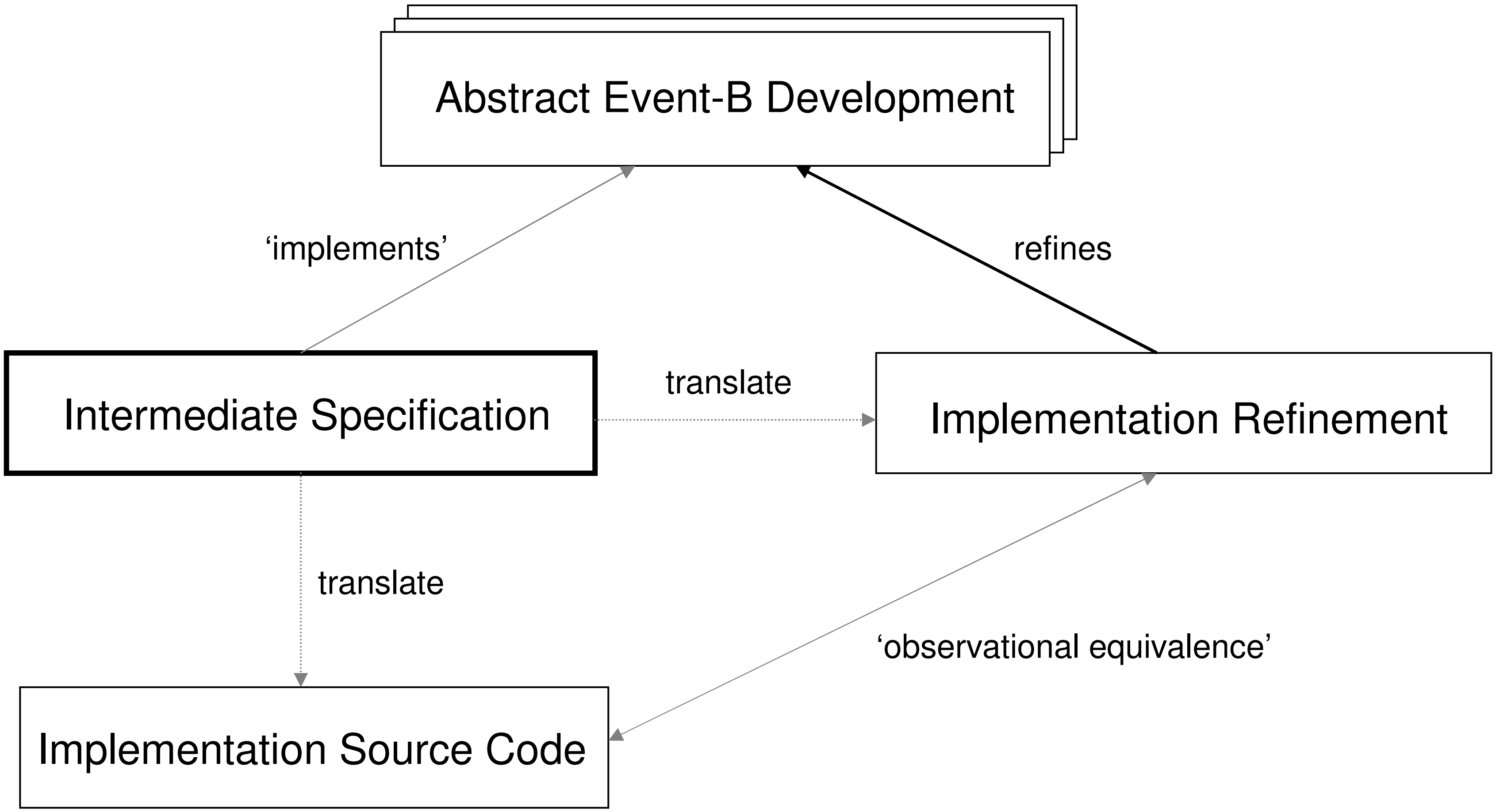
Rodin User and Developer Workshop

Generating Code from Event-B Using an Intermediate Specification Notation

Andy Edmunds - ae02@ecs.soton.ac.uk

Michael Butler - mjb@ecs.soton.ac.uk

Between Abstract Development and Code



Concurrent Processes, Sharing Data

- We can specify processes with a non-atomic operation, for implementing:
  - Shared memory systems
  - Thread-like behaviour
  - Interleaved atomic executions
  - A 'main' process which can provide an execution entry point
- We share data between processes with 'monitor-like' constructs
  - Atomic procedure calls (implementation provides mutex access)
- Can incorporate object-oriented features

An Example Process Specification

```
ProcessClass Proc {
 // encapsulated attributes
 Buffer buff, Boolean isWriter, Channel c, Integer id, Integer tmpBuffSz, Integer tmpDat

 // initialisations
 Procedure create(Integer pid, Buffer bff, Boolean isWritr, Channel ch){
   id:=pid || buff:=bff || isWriter:=isWritr || c:= ch || tmpBuffSz:=-1 || tmpDat:=-1
 }

 // The process behaviour
 Operation run(){
   p1:   if(isWriter=TRUE) then
         tmpBuffSz:=buff.getSize() andthen
         p2: c.getWChan(id, tmpBuffSz);
         p3: while(tmpBuffSz>0) do tmpDat:=buff.remove() andthen
             p4: c.add(tmpDat);
             p5: tmpBuffSz:=tmpBuffSz-1 endwhile ;
         p6: c.freeWChan() endif
   else c.getRChan(id) andthen
         p7: tmpBuffSz:=c.getWriteSize();
         p8: while(tmpBuffSz>0) do tmpDat:=c.remove() andthen
             p9: buff.add(tmpDat);
             p10: tmpBuffSz:=tmpBuffSz-1 endwhile ;
         p11: c.freeRChan() endelse
 }}
```

An Example Monitor Specification

```
MonitorClass Channel{
 // encapsulated attributes
  Integer capacity, Integer[5] buff, Integer head, Integer tail, Integer size, Integer rPID,
  Integer wPID, Integer writeSize

  // initialisations
  Procedure create(){
   head:= 0 || tail:= 0 || size:= 0 || capacity:= 5 || rPID:= -1 || wPID:= -1 || writeSize:= -1 }

// add a value to the tail
  Procedure add(Integer val){
     when(size<capacity){ buff[ tail ]:= val || tail:= (tail+1) mod capacity || size:= size+1} }

// remove and return the value from the buffer head
  Procedure remove(){
   when(size>0){ return:= buff[ head ] || size:= size-1 ||
       head:= (head+1) mod capacity}
  }: Integer
  ….
}
```

Non-atomic Operations

- Allow specification of sequences of interleaving atomic clauses
  using ';' operator to define points that allows interleaving

- Example non-atomic operation

$$op \triangleq label1: \; x := y \;\; ; label2: \; y := z$$

- A Non-atomic clause:

  Can have one or more labelled atomic clauses
  (each clause requires a unique label)

  Does not use synchronization constructs in the specification

Labels as Program Counters

- Program Counters for a process, pc = { label1, label2, term }
  ('term' is the terminating counter of a process)

- The clause op ≙ label1: x := y ; label2: y := z
  has Event-B semantics,

op_l1 ≙ **WHEN** pc = label1 **THEN** x := y || pc := label2 **END**

op_l2 ≙ **WHEN** pc = label2 **THEN** y := z || pc := term **END**

A Quick Look at Non-atomic Operation Syntax

Our formal definition uses the Guarded Command Language

NonAtomic ::=
NonAtomic ; *NonAtomic*
| *NonAtomic* ▯ *NonAtomic*
| **do** *Atomic* [; *NonAtomic*] **od**
| *Atomic*

Atomic ::= *Label*: ◁ *Guard* → *Body* ▷

omitted from the specification when true.

Body ::= Assignments | Call

Mapping to Event-B (Sequence)

- Translation Function

$$TNA \in NonAtomic \times Label \times PName \rightarrow \mathbb{P}(Events)$$

(where PName distinguishes the process by name, and Label is the exit label)

- Translation rule for a sequential clause

$$< na1 \; ; \; na2, \; l2, \; P >^{TNA}$$
$$\triangleq$$
$$< na1, \; l1, \; P >^{TNA} \cup < na2, \; l2, \; P >^{TNA}$$

We find the exit label for *na1* using a function sLabel(*na2*)

We define: sLabel $\in$ NonAtomic $\rightarrow$ Label

and, sLabel(*na2*) = *l1*

Mapping to Event-B (Labelled Atomic)

-Translation function TLA for actions (Base case)

$$\text{TLA} \in \text{Atomic} \times \text{Label} \times \text{PName} \rightarrow \mathbb{P}(\text{Event})$$

- Translation rule for a guarded atomic action

$$< \text{l1}: \triangleleft\, g \rightarrow a \,\triangleright , \text{l2}, \text{P} >^{\text{TLA}}$$

$$\triangleq$$

$$l1_p = \textbf{WHEN} \;\; P_{pc} = l1 \wedge g$$
$$\textbf{THEN} \;\; a \,||\, P_{pc} := l2$$
$$\textbf{END}$$

where P_{pc} is the program counter of Process P

The Branching Non-atomic Clause

- A simple branching construct

  l1: **if**(g_1) **then** a_1 [**andthen** na_1] **endif**
  else a_2 [**andthen** na_2] **endelse**

syntactic sugar for:

$l1: \lhd \; g_1 \rightarrow a_1 \; \rhd [\; ; na_1 \;]$
$\square \; l1: \lhd \; \neg g_1 \rightarrow a_2 \; \rhd [\; ; na_2 \;]$

- Translation rule for a branching clause

$$< na1 \;\square\; na2, l2, P >^{TNA}$$
$$\widehat{=}$$
$$< na1, l2, P >^{TNA} \cup < na2, l2, P >^{TNA}$$

this results in an event per branch e.g. l1_true, l1_else

UNIVERSITY OF Southampton School of Electronics and Computer Science

The Looping Non-atomic Clause

- The non-atomic loop construct (interleaving allowed after each iteration)

  l1: **while**(*g*) **do** *a* **endwhile**

  syntactic sugar for:

  do *l1*: $\lhd\ g \rightarrow a \rhd$ **od**

- Translation rule for a looping clause

$$< \mathbf{do}\ l1: \lhd\ g \rightarrow a\ \rhd\ \mathbf{od},\ l2,\ P >^{TNA}$$
$$\triangleq$$
$$< l1: \lhd\ g \rightarrow a\ \rhd,\ l1,\ P >^{TLA} \cup < l1: \lhd\ \neg g \rightarrow skip\ \rhd,\ l2,\ P >^{TLA}$$

- Also we have,

  l1: **while**(*g*) **do** *a* **andthen** *na* **endwhile**

Procedures

- A procedure definition:

$$\text{Procedure} = \text{LVar} \times \text{Guard} \times \text{Action} \times \text{T}$$

where LVar is a list of local variables (including formal params),
and T is the return type if applicable

- A procedure definition of Monitor *m* with name *pn* can be written,

$$pn(\ fp_1, \ldots, fp_k\ )\ \{ \lhd\ g_p \rightarrow a \rhd \} : T$$

with formal parameters $fp_1, \ldots, fp_k$

- For use above we have a sugared form of conditional waiting construct,

$$\textbf{when}(\ g_p\ )\ \{\ \text{a}\ \} \quad \triangleq \quad \lhd\ g_p \rightarrow a \rhd$$

Procedures

- A call is written

$$[\ v := \ ]\ \ m.\ pn(\ ap_1,\ \ldots,\ ap_k\ )$$

- The translation rule for TLA is defined as:

$$< l1 \colon \lhd\ g_c \rightarrow [\ v := \ ]\ m.pn(\ ap_1,\ \ldots,\ ap_k\ ) \rhd,\ l2,\ P >^{TLA}$$

$$\triangleq$$

$l1_p$ = **WHEN** $P_{pc} = l1 \wedge g_p[\ fp_1,\ldots,fp_k \setminus ap_1,\ldots,ap_k\ ] \wedge g_c$
THEN $a[\ fp_1,\ldots,fp_k \setminus ap_1,\ldots,ap_k\ ][$ **return** $\setminus v\ ] \mid\mid P_{pc} := l2$
END

Adding O-O Features (OCB)

- ProcessClass and MonitorClass Specification
  - User invokes *create* method to instantiate classes
  - Ease of mapping to OO code (Java in our case)
  - Potential to link with UML-B

- In the Event-B mapping:
  - Model instantiation, similar to UML-B
  - Variable renaming avoids name-clashes

Mapping to Java 1.4

-Many restrictions on OCB

To ensure mutual exclusion

To avoid deadlock due to nested monitors, resource contention

- Parallel to sequential semantics

- Conditional waiting using the **when** construct

```
Procedure remove(){
  when(size>0){
    return:= buff[ head ] || size:= size-1 ||
    head:= (head+1) mod capacity}
}: Integer
```

maps to

```
public synchronized int remove() {
   int initial_head = head;
   try{
       while (!(size > 0)){
          wait();
          initial head = head;
      } catch (InterruptedException e) { …. }
   size = size - 1;
   head = (initial_head + 1) % capacity;
   notifyAll();
   return buff[ initial head ];
}
```

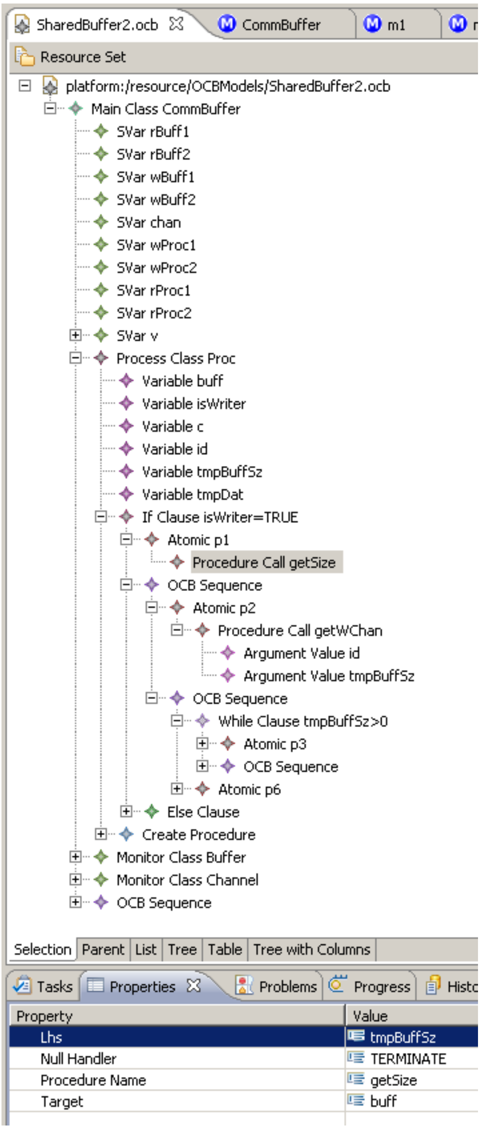
Mapping to Java 1.5

- Transactional OCB – relaxes restrictions
  - Access multiple shared objects in an atomic clause
  - Direct access to shared objects,
    or multiple procedure calls to shared objects, in a clause
  - Use of lock manager to acquire locks

- Add Event-B features:
  - Atomic constructs for implementation level
    - Sequence operator for actions
    - Atomic Branch and Loop

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Future Work



- Develop tools further

  Prototype tool has limited functionality:

  Improve Rodin integration

  link between abstract development and implementation refinement

  Improve static checking

- Map to other languages, SparkAda etc.

- Add text editor

- Integrate with UML-B

- Handle large Event-B implementation refinements