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

Abstract Event-B Development

Intermediate Specification

Implementation Source Code

Implementation Refinement

‘implements’

refines

translate

translate

‘observational equivalence’
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}
   }
   }
Non-atomic Operations

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

- Example non-atomic operation

\[
\text{op } \triangleq \text{ label1: } x := y \ ; \text{ label2: } y := z
\]

- 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, \( pc = \{ \text{label1, label2, term} \} \)
  ( ‘term’ is the terminating counter of a process )

- The clause \( \text{op} \triangleq \text{label1: } x := y \; ; \text{label2: } y := z \) has Event-B semantics,

\[
\begin{align*}
\text{op}_\text{l1} & \triangleq \text{WHEN } pc = \text{label1 } \text{THEN } x := y \; || \; pc := \text{label2 } \text{END} \\
\text{op}_\text{l2} & \triangleq \text{WHEN } pc = \text{label2 } \text{THEN } y := z \; || \; pc := \text{term } \text{END}
\end{align*}
\]
A Quick Look at Non-atomic Operation Syntax

Our formal definition uses the Guarded Command Language

\[
\text{NonAtomic} ::= \\
\text{NonAtomic} ; \text{NonAtomic} \\
| \text{NonAtomic} \ [ ; \text{NonAtomic} \ ] \text{ od} \\
| \text{Atomic} \\
\]

\[
\text{Atomic} ::= \text{Label: } \triangleleft \text{Guard} \rightarrow \text{Body} \triangleright \\
\]

omitted from the specification when true.

\[
\text{Body} ::= \text{Assignments} | \text{Call}
\]
- Translation Function

\[
\text{TNA} \in \text{NonAtomic} \times \text{Label} \times \text{PName} \rightarrow \mathcal{P}(\text{Events})
\]

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

- Translation rule for a sequential clause

\[
\langle \text{na}1; \text{na}2, l2, P \rangle^\text{TNA} \\
\cong \\
\langle \text{na}1, l1, P \rangle^\text{TNA} \cup \langle \text{na}2, l2, P \rangle^\text{TNA}
\]

We find the exit label for \text{na}1 using a function sLabel(\text{na}2)

We define: \text{sLabel} \in \text{NonAtomic} \rightarrow \text{Label}

and, \text{sLabel}(\text{na}2) = l1
Mapping to Event-B (Labelled Atomic)

-Translation function TLA for actions (Base case)

\[ \text{TLA} \in \text{Atomic} \times \text{Label} \times \text{PName} \rightarrow \mathbb{I}^p(\text{Event}) \]

- Translation rule for a guarded atomic action

\[ < l_1: \langle g \rightarrow a \rangle, l_2, P >^{\text{TLA}} \]
\[ \triangleq \]
\[ l_1_p = \text{WHEN } P_{pc} = l_1 \land g \]
\[ \text{THEN } a \parallel P_{pc} := l_2 \]
\[ \text{END} \]

where \( P_{pc} \) is the program counter of Process \( P \)
The Branching Non-atomic Clause

- A simple branching construct

\[
\textit{l1: if} \ (g_1) \ \textit{then} \ a_1 [ \ \textit{andthen} \ na_1 ] \ \textit{endif}
\]
\[
\textit{else} \ a_2 [ \ \textit{andthen} \ na_2 ] \ \textit{elseif}
\]

syntactic sugar for:

\[
\textit{l1: } \langle \ g_1 \rightarrow a_1 \ \rangle [ \ ; \ na_1 ]
\]
\[
\textit{else } \langle \ \neg g_1 \rightarrow a_2 \ \rangle [ \ ; \ na_2 ]
\]

- Translation rule for a branching clause

\[
\langle na_1 \ [ ] \ na_2, l_2, P \rangle^{TNA}
\]
\[
\triangleq
\]
\[
\langle na_1, l_2, P \rangle^{TNA} \cup \langle na_2, l_2, P \rangle^{TNA}
\]

this results in an event per branch e.g. l1_true, l1_else
- The non-atomic loop construct (interleaving allowed after each iteration)

\[ l1: \text{while}(\ g\ ) \text{ do } a \text{ endwhile} \]

syntactic sugar for:

\[ \textbf{do } l1: \triangleleft g \rightarrow a \triangleright \textbf{od} \]

- Translation rule for a looping clause

\[ < \textbf{do } l1: \triangleleft g \rightarrow a \triangleright \textbf{od}, l2, P >^{TNA} \]
\[ \triangleq \]
\[ < l1: \triangleleft g \rightarrow a \triangleright, l1, P >^{TLA} \cup < l1: \triangleleft \neg g \rightarrow \text{skip} \triangleright, l2, P >^{TLA} \]

- Also we have,

\[ l1: \textbf{while}(\ g\ ) \text{ do } a \text{ andthen } na \text{ endwhile} \]
- 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,

\[
pn( fp_1, \ldots, fp_k ) \{ \leftarrow g_p \rightarrow a \rightarrow \} : T
\]

with formal parameters \( fp_1, \ldots, fp_k \).

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

\[
\text{when}( g_p ) \{ a \} \equiv \leftarrow g_p \rightarrow a \rightarrow
\]
Procedures

- A call is written

\[ \text{[ } v := ] \ m. \ pn( \ ap_1, \ldots, \ ap_k ) \text{] } \]

- The translation rule for TLA is defined as:

\[ < l1: \triangleleft \ g_c \rightarrow [ \ v := ] \ m. \ pn( \ ap_1, \ldots, \ ap_k ) \triangleright, \ l2, \ P >^\text{TLA} \]
\[ \triangleq \]
\[ l1_p = \text{WHEN } P_{pc} = l1 \land g_p[ fp_1, \ldots, fp_k \setminus ap_1, \ldots, ap_k ] \land g_c \]
\[ \text{THEN } a[ fp_1, \ldots, fp_k \setminus ap_1, \ldots, ap_k ][ \text{ return } \triangleright v ] \triangleright \parallel P_{pc} := l2 \]
\[ \text{END} \]
Adding O-O Features (OCB)

- ProcessClass and MonitorClass Specification
  User invokes \textit{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()

```java
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 ];
}
```

**Procedure** remove()

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