Relating problem structure and solution structure in Event-B refinement

Michael Butler

WG 2.3 Meeting, Winchester
Problem versus Solution

**Problem decomposition:**
- In order to understand a complex problem, break it into sub-problems and master each sub-problem
- Sub-problem should have a degree of ‘unity’ 
  *(Michael Jackson)*

**Solution decomposition:**
- Structure an implementation of a system as an assembly of interacting components
- Follow architectural principles
- Problem structure may be orthogonal to solution structure
Example: Microproc ISA (John Colley, PhD thesis 2010)

- Problem decomposition
  - Arithmetic instructions
  - Memory access instructions
  - Branching instructions

- Solution decomposition (pipeline arch.)
  - Stage 1: Inst fetch
  - Stage 2: Inst decode
  - Stage 3: Inst execute
  - Stage 4: Memory access
  - Stage 5: Writeback (to registers)

- These structures are orthogonal
Outline

- Event-B
  - Modelling
  - Refinement
  - Rodin tool
  - Decomposition
- Problem decomposition and reconciliation
- Transforming to solution structure
Event-B (Abrial)

- **State-based** (like ASM, B, VDM, Z)
  - events: guarded atomic assignments
  - set theory as mathematical language

- **Refinement:**
  - refinement at level of events
  - gluing invariants
  - based on Action Systems (Back)

- **Proof method** supported by Rodin tool
  - Refinement proof obligations (POs) generated from models
  - Automated and interactive theorem provers for POs
Access control example

Variables of Event-B model

@inv1 authorised ∈ User ↔ Activity // relation
@inv2 takeplace ∈ Room ↔ Activity // relation
@inv3 location ∈ User → Room // partial function
Variables and invariants of Event-B model

Variables of Event-B model

@inv1 authorised ∈ User ↔ Activity // relation
@inv2 takeplace ∈ Room ↔ Activity // relation
@inv3 location ∈ User → Room // partial function

Access control invariant:
if user $u$ is in room $r$,
then $u$ must be authorised to engaged in all activities that can take place in $r$

@inv4 $\forall u,r . u \in \text{dom}(\text{location}) \land \text{location}(u) = r \Rightarrow \text{takeplace}[r] \subseteq \text{authorised}[u]$
Rodin demo
Rodin Open Tool Platform

Extension of Eclipse IDE
Repository of structured modelling elements
Rodin Eclipse Builder manages:
  - Well-formedness + type checker
  - Consistency/refinement Proof Obligation generator
  - Proof manager
  - Propagation of changes

Extension points
Open source

www.event-b.org
Refinement

• Refinement: process of enriching or transforming a model in order to
  1. augment the functionality being modelled, or
  2. explain how some purpose is achieved

• Consistency of a refinement:
  • We use proof to verify the consistency of a refinement step
  • Failing proof can help us identify inconsistencies in a refinement step
We construct a new model (refinement)

Abstract guard on a user and room for entering

grd3: \( \text{takeplace}[r] \subseteq \text{authorised}[u] \)

is replaced by a guard on a token

grd3b: \( t \in \text{valid} \land \text{room}(t) = r \land \text{holder}(t) = u \)
Feature augmentation: layered specification of Flash-based filestore

**ML0**: Tree properties and basic operations affecting tree structure

**ML1**: Partition *objects* into *files* and *directories*

**ML2**: Introduces file content

**ML3**: Introduces permissions

**ML4**: Introduces other missing properties such as *name*, *date* of creation and last modification

*Damchoom, K., Butler, M. and Abrial, J. R.*

*Modelling and proof of a Tree-structured File System in Event-B and Rodin.*

*ICFEM 2008*
Decomposition in Event-B

- **Shared variable decomposition**
  - strong coupling between machines (weak encapsulation)
  - data refinement of shared variables is difficult

- **Shared event decomposition**
  - embodies strong encapsulation
  - data refinement of encapsulated variables is easy
Shared Variable Decomposition

Partition the events
Shared Event Decomposition

Partition the variables
Refinement after decomposition

• Shared event: can refine sub-model provided
  • Common parameters of shared events are maintained

• Shared variable: can refine sub-model provided
  • External events are not refined (rely condition)
  • Shared variables are not refined.
  • Invariants used in refinement are preserved by external events
Partitioning events for sharing

\[ E = \]
\[ \text{any } p \text{ where } G1(x, p) \text{ and } G2(y, p) \]
\[ \text{then } \]
\[ x := H1(x, p) \]
\[ y := H2(y, p) \]
\[ \text{end} \]

\[ Ex = \]
\[ \text{any } p \text{ where } G1(x, p) \]
\[ \text{then } \]
\[ x := H1(x, p) \]
\[ \text{end} \]

\[ Ey = \]
\[ \text{any } p \text{ where } G2(y, p) \]
\[ \text{then } \]
\[ y := H2(y, p) \]
\[ \text{end} \]
Simple value transfer example

variable x, y

invariant x + y = N

init x:=0 || y:= N

event TRANSFER

any a where
  a ∈ ℤ
  y ≥ a
then
  x := x + a
  y := y - a
end
Decompose

<table>
<thead>
<tr>
<th>X1</th>
<th>Y1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{variable} \ x</td>
<td>\textbf{variable} \ y</td>
</tr>
<tr>
<td>\textbf{invariant} \ x \in \mathbb{Z}</td>
<td>\textbf{invariant} \ y \in \mathbb{Z}</td>
</tr>
<tr>
<td>\textbf{init} \ x:=0</td>
<td>\textbf{init} \ y:=N</td>
</tr>
<tr>
<td>\textbf{Event} \ \text{INC} \ any \ a \ \text{where} \ a \in \mathbb{Z} \ \text{then} \ x := x + a \ \text{end}</td>
<td>\textbf{Event} \ \text{DEC} \ any \ a \ \text{where} \ a \in \mathbb{Z} \ \text{and} \ y \geq a \ \text{then} \ y := y-a \ \text{end}</td>
</tr>
</tbody>
</table>
Pre-partitioning

E =

any p where
  G1( x, p, f(y) )
  G2( y, p )
then
  x := H1( x, p, f(y) )
  y := H2( y, p )
end

E =

any p, q where
  q = f(y)
  G1( x, p, q )
  G2( y, p )
then
  x := H1( x, p, q )
  y := H2( y, p )
end
Refinement and decomposition of filestore

Solution decomposition

Feature augmentation

Structural refinement

Further refinements focusing on flash spec
Structural refinement of FFS

ML5: Decomposes event *write* into

\[ w_{\text{start}}, w_{\text{step}}, w_{\text{end}} (\text{ok, fail}) \]

ML6: Decomposes event *read* into

\[ r_{\text{start}}, r_{\text{step}}, r_{\text{end}} (\text{ok, fail}) \]

ML7: Links the FS to the flash specification by introducing flash properties
Solution Decomposition

Partitions the machine level 7 into two machines representing the file system layer (FS) and the flash interface layer (FL).

Diagram of the machine decomposition

```plaintext
File System

fat, fat_tmp, wbbuffer, writing, rbuffer, ...

page
write

page
read

flash, programmed_pages, obsolete_pages

Flash Interface
```
Supporting sub-problem structuring

- Hypothesis:
  - Use *shared variable* for problem composition
  - Use *shared event* for solution decomposition
Sub-problems as Event-B machines

• Problem decomposition takes place prior to formalisation
  • sub-problems are identified informally
• Each sub-problem is then formalised
• Can the sub-problems be refined separately?

Diagram:
- E1 connected to v1
- E2 connected to v2
- E3 connected to v2
- E4 connected to v3
Sub-problems & separate refinement

Diagram showing a network of nodes labeled E1, E2, E3, E4, v1, v2, v3.
Sub-problem ‘reconciliation’
Problem decomposition and reconciliation

• In order to be able to refine sub-problems we ‘reconcile’ the sub-problem models
  • through addition of appropriate external events
  • reconciliation is formal

• Problem decomposition is informal

• Reconciliation of sub-models is formal
From problems to solution (staged)

Factorisation
Encapsulation means v2 can be data refined independently

Factorisation should be delayed until refinement of subproblems no longer relies on v2
Exchange

\[ P_1 + P_2 \]

\[ (Q_1 \cdot R_1) + (Q_2 \cdot R_2) \]

\[ (Q_1 + Q_2) \cdot (R_1 + R_2) \]

Q1 and Q2 share vars

Q1 and R2 do not share vars

R1 and R2 share vars

Q2 and R1 do not share vars
(Q1 \cdot R1) + (Q2 \cdot R2)
\[(Q1 + Q2) \cdot (R1 + R2)\]
Exchange in pipeline architecture

A + M

(A1 ⋅ R1) + (A2 ⋅ M2)

(A1 + M2) ⋅ (A2 + M2)
Shared event decomposition (with external variables and events)
Structure refactoring

Factorisation:

\[ P_1 + P_2 = (P'_1 + P'_2) \cdot S_1 \]

Exchange:

\[ (Q_1 \cdot R_1) + (Q_2 \cdot R_2) = (Q_1 + Q_2) \cdot (R_1 + R_2) \]
What’s needed in tooling?

• Explicit representation of mixed + and \( \cdot \) compositions
  • This allows factorisation and exchange to be made explicit, e.g.,
    \[
    \text{Sys1} = M1 + M2 \\
    \text{Sys2} = (M1' + M2') \cdot M3
    \]
  • This means we keep refining a ‘global plan’ as well as refining the individual machines - at least until we reach a top-level product.

• Shared event decomposition should cater for partitioning of external variables and events
Concluding

• Distinguishing problem structure from solution structure feels right
  • Automotive features, distributed services, multicore operating systems, network services,…

• Working hypothesis:
  • Shared variables for problem composition
  • Shared events for solution composition
  But alternatives should be explored too

• Factorisation and exchange support stepwise refactoring of structure from problem to solution

• Abstract algorithmic structures:
  • Need for rules for refactoring these