Abstraction, Refinement and Decomposition for Systems Engineering

Michael Butler
users.ecs.soton.ac.uk/mjb

Marktoberdorf Summer School 2012
Abstraction, Refinement and Decomposition for Systems Engineering (Using Event-B)

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf Summer School 2012
Lecture 1: Problem Abstraction and Model Refinement - An Overview

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf Summer School 2012
Overview

• Motivation
  – difficulty of discovering errors / cost of fixing errors

• Small pedagogical example (access control)
  – abstraction
  – refinement
  – automated analysis

• Background on Event-B formal method

• Methodological considerations
Cost of fixing requirements errors

“Extra Time Saves Money”
Warren Kuffel
Computer Language
December 1990
Cost of error fixes grows - difficult to change this
Rate of error discovery

Error discovery rate

Time of error discovery

- Reqs
- Spec
- Design
- Impl
- Test & fix
- Accept testing
- Deploy
Invert error identification rate?

![Graph showing time of error discovery and error discovery rate across different stages of software development: Reqs, Spec, Design, Impl, Test & fix, Accept testing, Deploy.]
Why is it difficult to identify errors?

- Lack of precision
  - ambiguities
  - inconsistencies

- Too much complexity
  - complexity of requirements
  - complexity of operating environment
  - complexity of designs
Need for precision and abstraction at early stages (front-loading)

• **Precision through early stage models**
  – Amenable to analysis by tools
  – Identify and fix ambiguities and inconsistencies as early as possible

• **Mastering complexity through abstraction**
  – Focus on *what* a system does (its purpose)
  – Incremental analysis and design
Rational design, by example

- Example: access control system

- Example intended to give a feeling for:
  - problem abstraction
  - modelling language
  - model refinement
  - role of verification and Rodin tool
Important distinction

• Program Abstraction:
  – Automated process based on a formal artifact (program)
  – Purpose is to reduce complexity of automated verification

• Problem Abstraction:
  – Creative process based on informal requirements
  – Purpose is to increase understanding of problem
Access control requirements

1. Users are authorised to engage in activities
2. User authorisation may be added or revoked
3. Activities take place in rooms
4. Users gain access to a room using a one-time token provided they have authority to engage in the room activities
5. Tokens are issued by a central authority
6. Tokens are time stamped
7. A room gateway allows access with a token provided the token is valid
Access control requirements

1. Users are authorised to engage in activities
2. User authorisation may be added or revoked
3. Activities take place in rooms
4. Users gain access to a room using a one-time token provided they have authority to engage in the room activities
5. Tokens are issued by a central authority
6. Tokens are time stamped
7. A room gateway allows access with a token provided the token is valid
Entities and relationships

- **USER**
  - holder
  - room
  - authorise

- **TOKEN**
  - issuer

- **ROOM**
  - location
  - manage
  - read

- **AUTHORITY**
  - manage

- **ACTIVITY**
  - take place

- **GATEWAY**
  - guards
  - trust

Relationships:
- User is authorised to perform activities.
- Token is associated with user, issuer, and room.
- Room is managed by authority.
- Authority manages gateways.
- Gateways handle guards and trust.
This model is unnecessarily complex to specify the main access control policy
Extracting the essence

• **Purpose** of our system is to enforce an access control policy

• **Access Control Policy**: *Users may only be in a room if they are authorised to engage in all activities that may take place in that room*

• To express this we only require **Users, Rooms, Activities and relationships** between them

• **Abstraction**: focus on key entities in the problem domain related to the purpose of the system
Entities and relationships

- USER
- ACTIVITY
- TOKEN
- ROOM
- AUTHORITY
- GATEWAY

Relationships:
- authorised
- location
- room
- authorise
- manage
- read
- guards
- trust
- takeplace
- issuer
Abstract by removing entities

Relationships represented in Event-B

authorised ∈ USER ↔ ACTIVITY  // relation
takeplace ∈ ROOM ↔ ACTIVITY  // relation
location ∈ USER ↦ ROOM      // partial function
Access control invariant

∀u,r . u ∈ dom(location) ∧ location(u) = r ⇒ takeplace[r] ⊆ authorised[u]

if user u is in room r,
then u must be authorised to engaged in all activities that can take place in r
## State snapshot as tables

**authorised**

<table>
<thead>
<tr>
<th>USER</th>
<th>ACTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>a1</td>
</tr>
<tr>
<td>u1</td>
<td>a2</td>
</tr>
<tr>
<td>u2</td>
<td>a1</td>
</tr>
</tbody>
</table>

**takeplace**

<table>
<thead>
<tr>
<th>ROOM</th>
<th>ACTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>a1</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
</tr>
<tr>
<td>r2</td>
<td>a1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USER</th>
<th>ROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>r1</td>
</tr>
<tr>
<td>u2</td>
<td>r2</td>
</tr>
<tr>
<td>u3</td>
<td></td>
</tr>
</tbody>
</table>
Event for entering a room

\[
\text{Enter}(u,r) \triangleq \\
\text{when} \\
\hspace{1cm} \text{grd1} : \ u \in \text{USER} \\
\hspace{1cm} \text{grd2} : \ r \in \text{ROOM} \\
\hspace{1cm} \text{grd3} : \ \text{takeplace}[r] \subseteq \text{authorised}[u] \\
\text{then} \\
\hspace{1cm} \text{act1} : \ \text{location}(u) := r \\
\text{end}
\]

Does this event maintain the access control invariant?
Role of invariants and guards

• **Invariants**: specify properties of model variables that should *always* remain true
  – violation of invariant is undesirable (*safety*)
  – use (automated) proof to verify invariant preservation

• **Guards**: specify *enabling conditions* under which events may occur
  – should be strong enough to ensure invariants are maintained by event actions
  – but not so strong that they prevent desirable behaviour (*liveness*)
Remove authorisation

RemoveAuth(u,a) ≜

when

  grd1 : u ∈ USER
  grd2 : a ∈ ACTIVITY
  grd3 : u ↦ a ∈ authorised

then

  act1 : authorised := authorised \ { u ↦ a }

end

Does this event maintain the access control invariant?
Counter-example from model checking

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>authorised</td>
<td>${(\text{User1} \rightarrow \text{Activity1}), (\text{User2} \rightarrow \text{Activity2})}$</td>
</tr>
<tr>
<td>location</td>
<td>${(\text{User1} \rightarrow \text{Room2})}$</td>
</tr>
<tr>
<td>takeplace</td>
<td>${(\text{Room1} \rightarrow \text{Activity1}), (\text{Room1} \rightarrow \text{Activity2}), (\text{Room2} \rightarrow \text{Activity1}), (\text{Room2} \rightarrow \text{Activity2})}$</td>
</tr>
</tbody>
</table>

invariant violated!
Operations

RemAuth(Activity2,User1)
Enter(Room2,User1)
AddAuth(Activity2,User2)
AddAuth(Activity2,User1)
AddAuth(Activity1,User1)
$initialise_machine({},{},{{z}})
$setup_constants()
(root)
invariant violated!
Failing proof
Strengthen guard of RemAuth
Early stage analysis

• We constructed a simple *abstract* model

• Already using verification technology we were able to *identify errors* in our conceptual model of the desired behaviour
  – we found a solution to these early on
  – verified the “correctness” of the solution

• Now, let’s proceed to another *stage* of analysis...
We construct a new model (refinement)

Guard of abstract Enter event:

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

is replaced by a guard on a token:

\[ \text{grd3b: } t \in \text{valid} \land \text{room}(t) = r \land \text{holder}(t) = u \]
Failing refinement proof

t\in\text{validToks}

r=\text{room}(t)

u=\text{holder}(t)

\text{takeplace}[[\{\text{room}(t)\}]\subseteq\text{authorised}[[\{\text{holder}(t)\}]]
To ensure consistency of the refinement we need invariant:  

inv 6: \( t \in \text{valid} \)  

\[ \Rightarrow \]  

\( \text{takeplace} [ \text{room}(t) ] \subseteq \text{authorised}[ \text{holder}(t) ] \)
Invariant enables PO discharge
But get new failing PO
Strengthen guard of refined `RemAuth`
Requirements revisited

1. Users are authorised to engage in activities
2. User authorisation may be added or revoked
3. Activities take place in rooms
4. ...

Question: was it obvious initially that revocation of authorisation was going to be problematic?
Rational design – what, how, why

• **What** does it achieve?
  
  if user $u$ is in room $r$,
  then $u$ must be authorised to engaged in all activities that can take place in $r$

• **How** does it work?
  
  Check that a user has a valid token

• **Why** does it work?
  
  For any valid token $t$, the holder of $t$ must be authorised to engage in all activities that can take place in the room associated with $t$
What, how, why written in B

• *What* does it achieve?
  
  \[ \text{inv1: } u \in \text{dom(location)} \land \text{location}(u) = r \]
  \[ \Rightarrow \]
  \[ \text{takeplace}[r] \subseteq \text{authorised}[u] \]

• *How* does it work?

  \[ \text{grd3b: } t \in \text{valid} \land r = \text{room}(t) \land u = \text{holder}(t) \]

• *Why* does it work?

  \[ \text{inv2: } t \in \text{valid} \]
  \[ \Rightarrow \]
  \[ \text{takeplace} \left[ \text{room}(t) \right] \subseteq \text{authorised} \left[ \text{holder}(t) \right] \]
B Method (Abrial, from 1990s)

- **Model** using set theory and logic

- **Analyse models** using proof, model checking, animation

- Refinement-based development
  - verify conformance between *higher-level* and *lower-level* models
  - chain of refinements

- Code generation from low-level models

- Commercial tools, :
  - *Atelier-B* (ClearSy, FR) - used mainly in railway industry
  - *B-Toolkit* (B-Core, UK, Ib Sorensen)
B evolves to Event-B (from 2004)

• B Method was designed for *software* development

• Realisation that it is important to reason about *system* behaviour, not just software

• Event-B is intended for modelling and refining system behaviour

• Refinement notion is more flexible than B
  • *Same set theory and logic*

• Rodin tool for Event-B (V1.0 2007)
  – Open source, Eclipse based, open architecture
  – Range of plug-in tools
System level reasoning

• Examples of systems modelled in Event-B:
  – Train signalling system
  – Mechanical press system
  – Access control system
  – Air traffic information system
  – Electronic purse system
  – Distributed database system
  – Cruise control system
  – Processor Instruction Set Architecture
  – ...

• System level reasoning:
  – Involves abstractions of overall system not just software components
Other Lectures

• Verification of Event-B models with Rodin tool
• Structured event decomposition
• Model decomposition
• Towards a method for decomposition
END
Verification and tools in Event-B modelling

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf Summer School 2012
Overview

• Abstraction & refinement
  validation & verification

• Proof obligations in Event-B

• Rodin tool features
Problem Abstraction

• Abstraction can be viewed as a process of simplifying our understanding of a system.

• The simplification should
  – focus on the intended purpose of the system
  – ignore details of how that purpose is achieved.

• The modeller/analyst should make judgements about what they believe to be the key features of the system.
Abstraction (continued)

• If the purpose is to provide some service, then
  – model what a system does from the perspective of the service users
  – ‘users’ might be computing agents as well as humans.

• If the purpose is to control, monitor or protect some phenomenon, then
  – the abstraction should focus on those phenomenon
  – in what way should they be controlled, monitored or protected?
Refinement

• Refinement is a process of enriching or modifying a model in order to
  – augment the functionality being modelled, or
  – explain how some purpose is achieved

• Facilitates abstraction: we can postpone treatment of some system features to later refinement steps

• Event-B provides a notion of consistency of a refinement:
  – Use proof to verify the consistency of a refinement step
  – Failing proof can help us identify inconsistencies
Validation and verification

• Requirements validation:
  – The extent to which (informal) requirements satisfy the needs of the stakeholders

• Model validation:
  – The extent to which (formal) model accurately captures the (informal) requirements

• Model verification:
  – The extent to which a model correctly maintains invariants or refines another (more abstract) model
    • Measured, e.g., by degree of validity of proof obligations
Event-B verification and tools
Event-B modelling components

- **machine** $m$
  - variables $\nu$
  - invariants $\ell$
  - events $e_1, e_2, \ldots$

- **context** $\text{ctx}$
  - sets $s$
  - constants $c$
  - axioms $\chi$

- **machine** $m_1$
  - sees context $c_1$

- **machine** $m_2$
  - sees context $c_2$

- **context** $c_1$
  - refines $m_1$
  - extends $m_2$

- **context** $c_2$
  - sees $m_2$
Event structure

\[ E = \]

any

x1, x2, ...

\[ \text{where} \]

G1

\text{(predicates)}

G2

...

\[ \text{then} \]

v1 := exp1

v2 := exp2

...

end

\\ event name

\\ event parameters

\\ event guards

\\ event actions
Role of Event Parameters

• Most generally, parameters represent nondeterministically chosen values, e.g.,

```plaintext
NonDetInc =

any d where v+d \leq MAX then v:=v+d end
```

• Event parameters can also be used to model input and output values of an event

• Can also have nondeterministic actions:

```plaintext
when v<MAX then v :| v < v' \leq MAX end
```
Refinement for events

• A refined machine has two kinds of events:
  – **Refined** events that refine some event of the abstract machine
  – **New** events that refine *skip*

• Verification of event refinement uses
  – **gluing** invariants linking abstract and concrete variables
  – **witnesses** for abstract parameters
Proof obligations in Event-B

- Well-definedness (WD)
  - e.g., avoid division by zero, partial function application
- Invariant preservation (INV) ***
  - each event maintains invariants
- Guard strengthening (GRD) ***
  - Refined event only possible when abstract event possible
- Simulation (SIM) ***
  - update of abstract variable correctly simulated by update of concrete variable
- Convergence (VAR)
  - Ensure convergence of new events using a variant
Invariant Preservation

• Assume: variables $v$ and invariant $I(v)$

• Deterministic event:
  $Ev = \text{when } P(v) \text{ then } v := \exp(v) \text{ end}$

• To prove $Ev$ preserves $I(v)$:
  
  INV: $P(v), I(v) \vdash I(\exp(v))$

• This is a sequent of the form Hypotheses $\vdash$ Goal

• The sequent is a Proof Obligation (PO) that must be verified
Using Event Parameters

- Event has form:

\[ \text{Ev} = \text{any } x \text{ where } P(x,v) \text{ then } v := \text{exp}(x,v) \text{ end} \]

INV: \[ I(v), P(x,v) \vdash I( E(x,v) ) \]
Example PO from Rodin

\[\forall u, r .
\begin{align*}
u \in \text{dom}(\text{location}) & \land \\
\text{location}(u) &= r
\end{align*}
\Rightarrow
\text{takeplace}[\{r\}] \subseteq \text{authorised}[\{u\}]
\]

\[u \in \text{USER} \setminus \text{dom}(\text{location})
\text{takeplace}[\{r\}] \subseteq \text{authorised}[\{u\}]
\]

\[(\text{locationu}\{u \mapsto r\})(u_0) = r_0\]
\[u_0 \in \text{dom}(\text{locationu}\{u \mapsto r\})
\text{takeplace} = \text{ROOM} \times \text{ACTIVITY}
\]

\[\text{location} = \text{USER} \mapsto \text{ROOM}
\]

\[\text{takeplace}[\{(\text{locationu}\{u \mapsto r\})(u_0)\}] \subseteq \text{authorised}[\{u_0\}]
\]
Simulation: maintaining a gluing relation

\[
\begin{array}{c}
\text{a0} \\
\downarrow \quad \downarrow \\
\text{c0} & \text{a1} \\
\downarrow \quad \downarrow \\
\text{c1}
\end{array}
\]

- abs
- con
- J
- J
New concrete events refine *skip* (stuttering step)
Refining traces

con1 \rightarrow new1 \rightarrow con2 \rightarrow new2 \rightarrow con3
abs1 \rightarrow abs2 \rightarrow abs3

J
Proof method for refinement (deterministic case)

• Suppose event $\text{con}$ refines event $\text{abs}$:
  \[
  \begin{align*}
  \text{abs} & = \text{when } P(a) \text{ then } a := E(a) \text{ end} \\
  \text{con} & = \text{when } Q(c) \text{ then } c := F(c) \text{ end}
  \end{align*}
  \]

• Verification of this refinement gives rise to two Proof Obligations:
  
  \[
  \begin{align*}
  \text{GRD:} & \quad I(a), J(a,c), Q(a) \vdash P(a) \\
  \text{SIM:} & \quad I(a), J(a,c), Q(a) \vdash J( E(a), F(c) )
  \end{align*}
  \]

• See [Abrial 2010] for non-deterministic case of refinement POs using witnesses
Some references

Comprehensive definition of proof obligations (plus much more):


Event- B is strongly influenced by Back’s action system formalism:

State trace refinement:


Event trace refinement:

Rodin Toolset for Event-B

• Extension of Eclipse IDE

• Rodin Builder manages:
  – Well-formedness + type checking
  – Consistency/refinement PO generator
  – Proof manager
  – Propagation of changes

• Extension points to support plug-ins
Rodin Proof Manager (PM)

- PM constructs **proof tree** for each PO
- Automatic and interactive modes
- PM calls **reasoners** to
  - discharge goal, or
  - split goal into subgoals
- Basic **tactic language** to adapt PM
- Collection of reasoners:
  - simplifiers, rule-based, decision procedures
Range of Automated Provers

- **Built-in**: tactic language, simplifiers, decision procedures
- **AtelierB plug-in** for Rodin (ClearSy, FR)
- **SMT plug-in** (Systerel, FR)
- **Isabelle plug-in** (Schmalz, ETHZ)
Supporting model changes

• Models are constantly being changed
  – When a model changes, proof impact of changes should be minimised as much as possible:

• Sufficiency comparison of POs
  – In case of success, provers return list of used hypotheses
  – Proof valid provided the used hypothesis in new version of a PO

• Renaming:
  – Identifier renaming applied to models (avoiding name clash)
  – Corresponding POs and proofs automatically renamed
ProB Model Checker (Leuschel)

• Automated checker
  – search for invariant violations
  – search for deadlocks
  – search for proof obligation violations

• Implementation uses constraint logic programming
  – makes all types finite
  – exploits symmetries in B types
Proof and model checking

• **Model checking:** force the model to be finite state and explore state space looking for invariant violations
  - 😊 completely automatic
  - 😊 powerful debugging tool (counter-examples)
  - 😞 state-space explosion

• **(Semi-)automated proof:** based on deduction rules
  - 😞 not completely automatic
  - 😊 leads to discovery of invariants - deepen understanding
  - 😊 no restrictions on state space
Some references

• Abrial, Butler, Hallerstede, Hoang, Mehta and Voisin
  *Rodin: An Open Toolset for Modelling and Reasoning in Event-B.*

• Leuschel and Butler
  *ProB: An Automated Analysis Toolset for the B Method.*
Rodin Demo

Access Control Example
Rodin Plug-ins

- ProB model checker:
  - consistency and refinement checking
- External provers:
  - AtelierB plug-in for Rodin (ClearSy, FR)
  - SMT plug-in (Systerel, FR)
  - Isabelle plug-in (Schmalz, ETHZ)
- Theory plug-in – user-defined mathematical theories
- UML-B: Linking UML and Event-B
- Graphical model animation
  - ProB, AnimB, B-Motion Studio
- Requirements management (ProR)
- Team-based development
- Decomposition
- Code generation
- ...

www.event-b.org
Contributors to Rodin toolset

Jean-Raymond Abrial
Stefan Hallerstede
Farhad Mehta
Thierry Lecomte
Mathieu Clabaut
Alexei Iliasov
Jens Bendisposto
Dominique Cansell
Renato Silva
Michael Jastram
Issam Maamria
Abdolbaghi Rezazadeh
Carine Pascal
Vitaly Savicks

Laurent Voisin
Thai Son Hoang
Christophe Metayer
Michael Leuschel
Colin Snook
Nicolas Beauger
Kriangsak Damchoom
Cliff Jones
Francois Terrier
Fabian Fritz
Andy Edmunds
Mar Yah Said
Andreas Furst
Thomas Muller

...
END
Abstract program structures for decomposing atomicity

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf 2012
Abstraction and decomposition

• In a refinement based approach it is beneficial to model systems abstractly with little architectural structure and large atomic steps
  – e.g., file transfer, distributed database transaction

• Refinement and decomposition are used to add structure and separate elements of the structure

• Atomicity decomposition
  – Decomposing large atomic steps to more fine-grained steps

• Model decomposition
  – Decomposing models for separate refinement of sub-models
Event-B style refinement

- Refinement
  - one-to-many event refinement
  - new events (refine *skip*)
- Flexible: allows complex relationships between abstract and refined models
- But (perhaps) too much flexibility
  - Need support for adding explicit “algorithmic” structures in refinement steps
simple file store example

machine filestore1

variables file, dsk

invariant
  file ⊆ FILE ∧
  dsk ∈ file → CONT

initialisation
  file := { } || dsk := { }

events

CreateFile ≜ ...

WriteFile ≜ // set contents of f to be c
  any f, c where
  f ∈ file
  c ∈ CONT
  then
  dsk(f) := c
  end

ReadFile ≜ // return contents of f
  any f, c! where
  f ∈ file
  c! = dsk(f)
  end
Sample event traces of file store

\[
\langle \text{CreateFile.f1}, \\
\text{WriteFile.f1.c1}, \\
\text{ReadFile.f1.c1}, \ldots \rangle
\]

\[
\langle \text{CreateFile.f1}, \\
\text{CreateFile.f2}, \\
\text{WriteFile.f2.c4}, \\
\text{WriteFile.f1.c6}, \ldots \rangle
\]

An (infinitely) many more traces.
Refinement of file store

- Structure of file content: \( \text{CONT} = \text{PAGE} \not\rightarrow \text{DATA} \)

- Instead of writing entire contents in one atomic step, each page is written separately:

  - **machine** filestore2  
    **refines** filestore

  - **variables** file, dsk, **writing**, wbuf, tdk

  - **invariant**
    
    \[
    \begin{align*}
    \text{writing} & \subseteq \text{file} \\
    \text{wbuf} & \in \text{writing} \rightarrow \text{CONT} \\
    \text{tdk} & \in \text{writing} \rightarrow \text{CONT} & \text{\(\Rightarrow\)} & \text{// temporary disk}
    \end{align*}
    \]
Refining the *WriteFile* event

- **Abstract:** *WriteFile*

- **Refinement:**
  - *StartWriteFile*
  - *WritePage*
  - *EndWriteFile* (refines *WriteFile*)
Events of refinement

StartWriteFile \( \triangleq \)

\[
\text{any } f, c \text{ where } \\
f \in (\text{file} \setminus \text{writing}) \\
c \in \text{CONT} \\
\text{then} \\
\text{writing} := \text{writing} \cup \{f\} \\
\text{wbuf}(f) := c \\
\text{tdsk}(f) := \{} \\
\text{end}
\]

WritePage \( \triangleq \)

\[
\text{any } f, p, d \text{ where } \\
f \in \text{writing} \\
p \mapsto d \in \text{wbuf}(f) \\
p \mapsto d \notin \text{tdsk}(f) \\
\text{then} \\
\text{tdsk}(f) := \text{tdsk}(f) \cup \{p \mapsto d\} \\
\text{end}
\]
Events of refinement

EndWriteFile

refines WriteFile ≜

any f, c where
f ∈ writing

c = tdsk(f)

dom( tdsk(f) ) =
    dom( wbuf(f) )

then

dsk(f) := tdsk(f)

writing := writing \ { f }

wbuf := wbuf \ { f }

tdsk := tdsk \ { f }

end

AbortWriteFile ≜

any f, c where
f ∈ writing

c = tdsk(f)

then

writing := writing \ { f }

wbuf := wbuf \ { f }

tdsk := tdsk \ { f }

end
Comparing abstract and refined traces

\langle \text{CreateFile.f1}, \\
\text{CreateFile.f2}, \\
\text{WriteFile.f2.c2}, \\
\text{WriteFile.f1.c1} \\
\ldots \rangle

\langle \text{CreateFile.f1}, \\
\text{StartWriteFile.f1.c1}, \\
\text{CreateFile.f2}, \\
\text{WritePage.f1.p2.c1(p2)}, \\
\text{StartWriteFile.f2.c2}, \\
\text{WritePage.f1.p1.c1(p1)}, \\
\text{WritePage.f2.p1.c2(p1)}, \\
\text{WritePage.f2.p2.c2(p2)}, \\
\text{EndWriteFile.f2.c2}, \\
\text{WritePage.f1.p3.c1(p2)}, \\
\text{EndWriteFile.f1.c1} \\
\ldots \rangle
Breaking atomicity

• Abstract *WriteFile* is replaced by
  – new events: *StartWriteFile, WritePage*,
  – refining event: *EndWriteFile*

• Refined events for *different* files may interleave

• Non-interference is dealt with by treating new events as refinements of *skip*
  – new events must maintain gluing invariants

• **But**: not all event relations are explicit
  – insufficient structure
Jackson Structure Diagrams

- Part of Jackson System Development

- Graphical representation of structured programs

- We can exploit the hierarchical nature of JSD diagrams to represent event refinement

- Adapt JSD notation for our needs
WriteFile sequencing in JSD

Sequencing is from left to right

* signifies iteration
Adapting the diagrams

- Attach the iterator to an arc rather than a node to clarify atomicity.
- Events are represented by leaves of the tree.
- Solid line indicates EndWrite refines Write.
- Dashed line indicates new events refining skip.
Nondeterministic forall

- pages may be written after \textit{StartWrite} has occurred
- the writing is complete (\textit{EndWrite}) once all pages have been written
- order of \textit{PageWrite} events is nondeterministic
- this abstract program structure represents atomicity refinement explicitly
Interleaving of multiple instances

- Multiple write “processes” for different files may interleave
  - (sub-)events of $\text{Write}(f_1)$ may interleave with (sub-)events of $\text{Write}(f_2)$
  - (sub-)events of $\text{Write}(f_1)$ may interleave with (sub-)events of $\text{Read}(f_1)$
- Interleaving can be reduced with explicit guards (e.g., write lock)
Hierarchical refinement

- **Write**($f$)
  - **all**($p$)
    - **StartWrite**($f$)
    - **PageWrite**($f,p$)
    - **EndWrite**($f$)
  - **all**($b$)
    - **StartPage**($f,p$)
    - **ByteWrite**($f,p,b$)
    - **EndPage**($f,p$)
Event-B encoding

variable \( B \subseteq S \land \text{finite}(S) \)

Events:

\[ B \triangleq x \in S \setminus B \quad \text{if} \quad B := B \cup \{x\} \]

\[ C \triangleq B = S \land \neg C \]

\[ C := \text{TRUE} \]
SOME program structure

Events:

\[ B \triangleq x \in S \setminus B \quad B := B \cup \{x\} \]

\[ C \triangleq B \neq \emptyset \quad \land \quad \neg C \]

\[ C := \text{TRUE} \]

C can occur provided B(x) occurs for at least one x

B(x’) may occur after C for other x’
Treating failure in file write

- **AbortWrite** may occur if **PageFail(p)** occurs for some page \( p \)
- Weak: **PageFail(p')** may occur for other \( p' \) after **AbortWrite**
Separation of concerns

WriteOk ≜ begin
  disk := file
end

WriteFail ≜ begin
  skip
end

WriteOk xor WriteFail
Layered refinement

• M0: two events - $WriteOk$ and $WriteFail$
• M1: refine atomicity of $WriteOk$
• M2: refine atomicity of $WriteFail$
Search

- **FindOk**: find a point in $S$ satisfying property $P$  \( x \in S \cap P \)
or
- **NoFind**: determine that no point in $S$ satisfies \( S \cap P = {} \)
Invariants for verification

- Pass $\subseteq S \cap P$
- Fail $\subseteq S \setminus P$
Transform to sequential model

StartFind ;
for i in S do
    Fail(i)
    []
    Pass(i) ; exit
od ;
if exit then FindOk else NoFind fi
Alternatively refine to parallel model

- Partition $S$ so that search is farmed out to multiple processors $p \in P$
- This is a simple refinement step in Event-B
Replicated data base

• Abstract model

\[ \text{db} \in \text{object} \rightarrow \text{DATA} \]

\[
\text{Commit} = \begin{array}{l}
/* \text{update a set of objects os} */ \\
\text{any os, update} \\
\text{where} \\
\text{os} \subseteq \text{object} \land \\
\text{update} \in (\text{os} \rightarrow \text{DATA}) \rightarrow (\text{os} \rightarrow \text{DATA}) \\
\text{then} \\
\text{db} := \text{db} \leftrightarrow \text{update(os} \triangleright \text{db})
\end{array}
\]
At abstract level, update transaction is a choice of 2 atomic events:
Refinement by replicated database

\[ \text{ldb} \in \text{site} \rightarrow (\text{object} \rightarrow \text{DATA}) \]

Update is by two phase commit:
- PreCommit followed by Commit
- Global commit if all sites \textit{pre-commit}
- Global abort if at least one site aborts
Event refinement diagram for Commit

Which event refines the abstract Commit?
Event refinement diagram for Commit

Decision to proceed is made by *GlobalCommit*
Event refinement diagram for Abort

Protocol aborts transaction if *some* site aborts
Locking objects

• $\text{PreCommit}(t,s)$ : locks all objects for transaction $t$ at site $s$

• $\text{LocalCommit}(t,s) \; \text{LocalAbort}(t,s)$ : release all objects for transaction $t$ at site $s$
Read transactions

- Abstract read: values read are from single abstract database $db$
- Concrete read: (provided objects are not locked) values read are from copy of database at a site $ldb(s)$

- Key gluing invariant:
  $\forall s, o \cdot o \notin \text{dom}(\text{lock}(s)) \Rightarrow (ldb(s))(o) = db(o)$

- But $(ldb(s))(o) = db(o)$ is broken by GlobalCommit
Global and local commit not synchronised

Commit updates \( db \), but
\textit{GlobalCommit} does not update \( ldb \)

\textit{LocalCommit} updates \( ldb(s) \)

How are \( db(o) \) and \( ldb(s)(o) \) related in between \textit{GlobalCommit} and \textit{LocalCommit}?
Another gluing invariant

t ∈ GlobalCommit ∧
t ↦ s /∈ LocalCommit ∧
os = tos[t] ∧ o ∈ os ∧
U = upd(t) ∧ L = os ⊲ ldb(s)

⇒

db(o) = (U(L))(o)

The abstract value of an object at a site is determined by applying the update associated with the transaction to the database at the local site.
Layered strategy for *Commit*

Layered strategy allowed us to focus on difficult part of the abstraction first, led to simpler invariants, hence simpler proofs.
Concluding

• Abstract program structures add value to existing refinement framework
  – Structures provide explicit representation of atomicity decomposition (with sufficient interleaving)
  – Power of diagrams – rapid understanding

• Not quite transformational approach:
  – abstract programs provide templates for constructing refined models
  – refined models are verified but templates increases likelihood of correctness
End
Model Decomposition for Distributed Design in Event-B

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf 2012
Decomposition

• Beneficial to model systems abstractly with little architectural structure and large atomic steps
  – e.g., file transfer, replicated database transaction

• Refinement and decomposition are used to add structure and then separate elements of the structure

• Atomicity decomposition: Decomposing large atomic steps to more fine-grained steps

• Model decomposition: Decomposing refined models to for (semi-)independent refinement of sub-models

• Towards a method for decomposition
Reminder

Event-B machine consists of

- **Variables** (e.g., *authorised*, *location*,...)

- **Invariants**
  - Predicate logic
  - Also used for type inference

- **Events**
  - Acting on variables, expected to maintain invariants
  - Specified by parameters, guards, actions
Model Decomposition styles

- **Shared Event**
  - Sub-models interact through synchronisation over shared events
  - Shared events can have common parameters

- **Shared Variable**
  - Sub-models interact through shared variables
  - Events are independent

- Both styles supported by a decomposition plug-in
Shared Event Decomposition

Partition the variables
Shared Event Decomposition – by example

A $\triangleq v := v+1$

B $\triangleq$ when $v > 0 \land w < M$ then $v := v-1 \mid \mid w := w+1$ end

C $\triangleq$ when $w > 0$ then $w := w-1$ end
Decompose by partitioning variables

\[ A \triangleq v := v + 1 \]

\[ B \triangleq \text{when } v > 0 \land w < M \text{ then } v := v - 1 \mid \mid w := w + 1 \text{ \text{end}} \]

\[ C \triangleq \text{when } w > 0 \text{ then } w := w - 1 \text{ \text{end}} \]

B event needs to be split into \( v \)-part and \( w \)-part
Parallel Event Split

B is split into two parallel events operating on independent variables:

B1 $\triangleq$ when $v > 0$ then $v := v - 1$ end

B2 $\triangleq$ when $w < M$ then $w := w + 1$ end
Synchronised events with parameter passing

\[ B \triangleq \text{any } x \text{ where } 0 < x \leq v \]

\[ \text{then } \ v := v - x \ | \ | \ w := w + x \ \text{end} \]

*B can be split into 2 events that have \( x \) in common:

\[ B1 \triangleq \text{any } x \text{ where } 0 < x \leq v \ \text{then } v := v - x \ \text{end} \]

\[ B2 \triangleq \text{any } x \text{ where } x \in \mathbb{Z} \ \text{then } w := w + x \ \text{end} \]

*B1 constrains the value for \( x \) by \( 0 < x \leq v \) (output)*

*B2 just constrains the value of \( x \) to a type (input)*
Partitioning events

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

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

\[ Ey = \]
\[ \text{any } p \text{ where } \]
\[ G2( y, p ) \]
\[ \text{then} \]
\[ y := H2( y, p ) \]
\[ \text{end} \]
Pre-partitioning

\[
E = \text{any } p \text{ where } \begin{align*}
&G1( x, p, f(y) ) \\
&G2( y, p )
\end{align*}
\]
then
\[
\begin{align*}
x &:= H1( x, p, f(y) ) \\
y &:= H2( y, p )
\end{align*}
\]
end

\[
E = \text{any } p, q \text{ where } \begin{align*}
&q = f(y) \\
&G1( x, p, q ) \\
&G2( y, p )
\end{align*}
\]
then
\[
\begin{align*}
x &:= H1( x, p, q ) \\
y &:= H2( y, p )
\end{align*}
\]
end

Transform E to make it easier to split into x-part and y-part
Composition and Decomposition

• Decomposition: from M, decomposition plug-in generates:
  – machines L, P
  – composed machine M'

• M' is a wrapper for L || P

• Consistency of decomposition:
  – prove M' refines M

composed machine M'
refines M
Includes L, P
events
  A = L.A
  B = L.B || P .B
  C = P.C
end
Shared event composition operator

• Shared event composition operator for Event-B machines is syntactically simple
  – combine guards and combine actions of events to be synchronised
  – no shared state variables
  – common event parameters represent values to be agreed by both parties on synchronisation

• Corresponds to parallel composition in CSP
  – processes interact via synchronised channels
  – monotonic: subsystems can be refined independently
Shared Variable Decomposition

Partition the events
Refinement after decomposition

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

• **Shared variable:** can refine sub-model provided
  • External events are not refined (rely condition)
  • Private events in M1 that affect shared variables must refine some external event of M2, e.g., E3 refines E3’
  • Shared variables are not refined.
  • Invariants used in refinement are preserved by external events
Observation on Decomposition

• The decomposition itself is straightforward
  – Essentially a syntactic partitioning of events

• The more challenging part is refining the abstract model to a sufficiently detailed model to allow the syntactic decomposition to take place
Asynchronous distributed system

For distributed systems, agents do not interact directly. Instead they interact via some middleware, e.g., the Internet.
Some references


• http://www.ecs.soton.ac.uk/people/mjb/publications
END
Towards a **Method** for Decomposition

Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf 2012
Decomposition

• Beneficial to model systems abstractly with little architectural structure and large atomic steps
  – e.g., file transfer, replicated database transaction

• Refinement and decomposition are used to add structure and then separate elements of the structure

• Atomicity decomposition: Decomposing large atomic steps to more fine-grained steps

• Model decomposition: Decomposing refined models to for (semi-)independent refinement of sub-models

• Towards a method for decomposition
Partition the variables
Asynchronous distributed system

For distributed systems, agents do not interact directly.
Instead they interact via some middleware, e.g., the Internet
Atomicity and machine decomposition of ATM
Abstract Events for Cash Withdrawal

An ATM transaction results in one of three outcomes:

Distributed implementation with ATM and Bank server:
- *LowCash* only affects the ATM
- *LowBal* and *Withdraw* affect ATM and Bank
**LowCash**: separate user request from ATM response

![Diagram](image)

- Transaction
  - xor
    - LowCash
    - LowBal
    - Withdraw
  - Req Cash
  - Low Cash
LowBal: introduce protocol steps
Withdraw: separate cash delivery and balance reduction
Withdraw: protocol steps

Withdraw

Deliv Cash

Reduce Balance

Req Cash
Query Bal
Resp OK
Deliv Cash
Conf
Reduce Bal
Separate sending and receiving for protocol steps

Which are **ATM** events and which are **Bank** events?
Distinguish ATM and Bank events
Extract ATM behaviour
Extract Bank behaviour

Bank withdraw

Recv Query
Send Resp
Recv Conf
Reduce Bal
What about communication between ATM and Bank?

Diagram:
- Withdraw
  - Deliv Cash
  - Reduce Balance
    - Conf
      - Reduce Bal
    - Send Conf
  - Deliv Resp
    - Send Resp
  - Send Query
  - Recv Query
  - Recv Resp
  - Resp OK
  - Query Bal
  - Req Cash
Identify need for asynchronous communication

Buffer is required whenever there is a transition from red to green or green to red
Decompose model into ATM, Bank and Buffers

ATM

Local Events
- ReqCash
- DelivCash
- LowCash
- LowBal

Variables
- cash

Shared Events
- SendQuery
- RecvResp
- SendConf

Buffer

Bank

Local Events
- ReduceBal

Variables
- balance

Shared Events
- SendResp
- RecvConf
- RecvQuery
Decomposition of replicated database
Abstraction of Distributed Database

Abstract model:

\[ \text{db} \in \text{object} \rightarrow \text{DATA} \]
Refinement by replicated database

\[ \text{ldb} \in \text{site} \rightarrow (\text{object} \rightarrow \text{DATA}) \]

- Decompose atomicity of Commit and Abort following 2-phase commit protocol
Structured refinement of *Commit*

```
Commit(t)
```

- **Start(t)**
- **PreCommit(t,s)**
- **Global Commit(t)**
- **Local Commit(t,s)**
Structured refinement of *Abort*

- **Abort(t)**
  - **Start(t)**
  - **Refuse(t,s)**
  - **Global Abort(t)**
  - **Local Abort(t,s)**

- **some s in SITE**
  - **all s in PreCommit[{t}]**
Towards a distributed system

1. Start with *atomic* model of transaction, independent of architecture/roles

2. Introduce *separate steps* of a transaction
   – independent transactions can run concurrently

3. Introduce explicit *message send/receive*
   – this will allow us to separate the coordinator and worker roles
Introducing messaging

Commit(t)

Start(t)

PreCommit(t, s)

Global Commit(t)

Broadcast Start(t)

RcvStart(s, t)

Pre Cmt(t, s)

Send Pre Cmt(t, s)

Recv Pre Commit(t, s)

all s
Separate **coordinator** and **worker** events

- **Commit(t)**
  - **Start(t)**
  - **PreCommit(t,s)**
    - **all s**
    - **Global Commit(t)**
      - **Broadcast Start(t)**
      - **RcvStart(s,t)**
      - **Pre Cmt(t,s)**
      - **Send Pre Cmt(t,s)**
      - **Recv Pre Commit(t,s)**
Identify communications buffers

Start(t) → PreCommit(t,s) → all s → Global Commit(t)

Broadcast Start(t) → RcvStart(s,t) → Pre Cmt(t,s) → Send Pre Cmt(t,s) → Recv Pre Commit(t,s)

Commit(t)
Coordinator abstract program

Coordinator(t)

Broadcast Start(t)

Recv Pre Commit(t,s)

Global Commit(t)

all s
Worker behaviour

NonCoordinator(s,t)

PreCommit(s,t)

RcvStart(s,t)  Pre Cmt(t,s)  Send Pre Cmt(t,s)
Other case studies

- Multimedia protocol (Asieh Salehi)
- Data manipulation in satellite (Asieh Salehi)
- Railway network (Renato Silva)
- Automotive control (Sanaz Yeganefard)
A TeleCommand (TC) is received by the Core from Earth.
The syntax of the received TC is checked in the core.
Further semantic checking has to be carried out either in the core or devices based on the type of TCs.
For all received TCs, a control TeleMessage (TM) is generated and sent back to Earth.
For some particular types of TC, one or more data TMs are generated and sent back to Earth.
Space Craft Development

Model Decomposition

Refinements
Before Decomposition

Refinements
After Decomposition

M0
M1
M2
M3

Device
Core

M4
M5
Event refinement structure

BepiColombo\(tc\)

- ReceiveTC\(tc\)
- TC_Validation_Ok\(tc\)
- TCValid_GenerateData\(tc\)
- TCValid_ReplyDataTM\(tc\)
- TC_Check_Ok\(tc\)
- TC_Execute_Ok\(tc\)
- TCExecOk_ReplyCtrlTM\(tc\)
- XOR
  - TC_Core_Execute_Ok\(tc\)
  - TC_Device_Execute_Ok\(tc\)
- SendTC_Core_to_Device\(tc\)
- CheckTC_in_Device_Ok\(tc\)
- SendOkTC_Device_to_Core\(tc\)
- TC_GenerateData_in_Device\(tc, d\)
- TC_TransferData_Device_to_Core\(tc\)
- ALL (tm)

164
Railway System Decomposition

- Decomposition for Railway
- 3 refinement levels: Railway_M0 to Railway_M2

- Decompose Railway_M2
Some references


• http://www.ecs.soton.ac.uk/people/mjb/publications
Code Generation from Event-B

Ada-Europe 2012
Background

• Typical embedded systems
  – Several concurrent tasks
  – Tasks may be aperiodic or periodic
  – Some sharing of variables
  – Task and data structures usually static

• Event-B supports modelling of concurrency
  – Model atomic steps in concurrent computation
  – Refinement allows atomicity to be refined with interleaving of (sub-)atomic steps
  – Events and machines are the basic structuring mechanisms
Tasking Event-B

• Tasking Machine (Event-B machine +explicit control flow term)
  – system may have several parallel tasking machines
  – add structured control flow to machine: ; / If / While
  – atomic steps in a task correspond to atomic events

• Environment Machine
  – Similar to tasking machine but only intended for simulation of controller environment

• Shared-data Machine (standard Event-B machine)
  – tasking machine interact indirectly via shared data machine

• Interaction between tasks and shared data represented by shared-event composition (synchronisation)
Proof and generation

• **Proof:** control flow structures are encoded as Event-B

• **Code generation:**
  – *Internal intermediate* language based on Ada subset (IL1)
  – Synchronisation implemented by synchronised call (monitor)
  – Back-end to textual Ada/C via simple rules

• **Data types:**
  – Data types are defined as reusable theories
  – Rewrite rules define back-end translation to Ada or C
Heating Controller Block Diagram

Main Functions

- Adjusting Target Temperature
- Sensing temperature
- Displaying current and target temperatures
- Activating/Deactivating Alarms
- Change target temperature
- Power on/off Heater
- Sensing heater status
Decomposition to tasks

Decomposing the Controller from its Environment

Controller

Environment

Decomposition of the Controller into Tasks and a shared Object

Controller

Shared Object

Temperature Ctrl Task

Display Task

Update Task

Monitor Task

Heater Task
Event-B Development for the Heating Controller

Specification Level
Refinement

First Level Decomposition

Environment

Second Level Decomposition

Shared Object

Task Bodies

Temperature Ctrl Task

Shared Object1

Temperature Ctrl Task1

Display Update Task

Heater Monitor Task

Heater Monitor Task1

Display Update Task1

Environ1
Not (yet) supporting...

- Dynamic task structures
- Fine-grained locking of shared variables
- Reasoning about timing properties of tasks
- ...

174
Wrap-up
Important Messages

- **Role of formal modelling / problem abstraction:**
  - increase understanding of problem
  - decrease errors

- **Role of refinement and decomposition:**
  - manage complexity through multiple levels of abstraction and architecture

- **Role of verification:**
  - improve quality of models (consistency, invariants)

- **Role of tools:**
  - make verification as automatic as possible, pin-pointing errors and even suggesting improvements

- Event-B can and should be linked with complementary methods
Challenges

- More powerful proof automation
- Richer modelling and refinement patterns
  - General and domain specific
  - Automated application of patterns
- Code generation:
  - Support much broader program structures
- Linking systematic requirements analysis with problem abstraction
  - General and domain-specific
  - Problem structure versus solution structure
- More experimental validation of methods and tools in realistic industrial settings
- Education/training
- ...
Keep up to date / contribute

• www.event-b.org

• wiki.event-b.org
  – share your Event-B models
  – share your plug-in plans
  – suggest plug-in ideas