System-level modelling with Event-B

B Dissemination Day, GRACE, Tokyo

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

School of Electronics and Computer Science
University of Southampton, UK
Contents

• Motivation
• Event-B overview
• *Rational design* with Event-B:
  – abstraction
  – refinement
  – proof and mechanical analysis
• Decomposition structures
• Take-home messages
System level

• Examples of systems:
  – Train signalling system
  – Mechanical press system
  – Access control system
  – Air traffic information system
  – Electronic purse system
  – Distributed database system
  – Cruise control system
  – …

• System level reasoning:
  – Involves abstractions of *overall* system not just software components
What’s wrong with the V model?

Many errors are introduced early but detected late – such errors are expensive to fix.
Why is it difficult to detect errors?

• Lack of precision
  – ambiguities
  – inconsistencies

• Too much **complexity:**
  – complexity of requirements
  – complexity of operating environment
  – complexity of designs
Need for precise models/blueprints

• Precision from early stages with models
  – Precise descriptions of intent
  – Amenable to analysis by tools
  – Identify and fix ambiguities and inconsistencies as early as possible

• Mastering complexity
  – Encourage abstraction
  – Focus on what a system does
  – Early focus on key/critical features
  – Incremental analysis and design
Formal Methods

• Mathematical techniques for formulation and analysis of systems

• Formal methods facilitate:
  – Clear specifications (contract)
  – Rigorous validation and verification

Validation: does the contract specify the right system?
  – answered informally

Verification: does the finished product satisfy the contract?
  – can be answered formally
Early stage analysis

Specification

Architectural design

Detailed design

Coding

Validation testing

Integration testing

Unit testing

Verification

Validation
Rapid prototyping versus modelling

• **Rapid prototyping:** provides early stage feedback on system functionality
  – Plays an important role in getting **user feedback**
  – and in understanding some design constraints
  – But we will see that formal modelling and proof provide a **deep understanding** that is hard to achieve with rapid prototyping

• **Advice:** use any approach that improves design process!
Event-B (Abrial)

• State-transition model (like ASM, B, VDM, Z)
  – set theory as mathematical language

• Refinement (based on action systems by Back)
  – data refinement
  – one-to-many event refinement
  – new events (stuttering steps)

• Proof method
  – Refinement proof obligations (POs) generated from models
  – Automated and interactive provers for POs
Rational design, by example

• Example: access control system

• Example intended to give a feeling for:
  – modelling language
  – abstraction and refinement
  – role of verification
Access control system

- Users are authorised to engage in activities
- User authorisation may be added or revoked
- Activities take place in rooms
- Users gain access to a room using a one-time token provided they have authority to engage in the room activities
- Tokens are issued by a central authority
- Tokens are time stamped
- A room gateway allows access with a token provided the token is valid
Entity-relationship diagram

- USER
- ACTIVITY
- TOKEN
- ROOM
- AUTHORITY
- GATEKEEPER

Relationships:
- authorised
- location
- room
- authorise
- manage
- read
- issuer
- trust
- guards
- takeplace
This model is unnecessarily complex to specify the main access control policy which concerns users, rooms and activities.
Simplify / abstract

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 $r$

$$\text{location}(u) = r \implies \text{takeplace}[r] \subseteq \text{authorised}[u]$$

Abstraction: focus on key entities in the problem domain
Enter a room

\[ \text{Enter} \triangleq \]
\[ \text{when} \]
\[ \text{grd1} : \quad u \in \text{User} \]
\[ \text{grd2} : \quad r \in \text{Room} \]
\[ \text{grd3} : \quad \text{takeplace}[r] \subseteq \text{authorised}[u] \]
\[ \text{then} \]
\[ \text{act1} : \quad \text{location}(u) := r \]
\[ \text{end} \]

Does this operation maintain the security invariant?
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 operation maintain the security invariant?
Counterexample from model checking with ProB plug-in for Rodin
Failing proof with Rodin
Strengthen guard of **RemAuth**
Now refine

Abstract condition on a user and room for entering:
\[ \text{takeplace}[ r ] \subseteq \text{authorised}[ u ] \]

is replaced by a condition on a token:
\[ t \in \text{valid} \land \text{room}(t) = r \land \text{holder}(t) = u \]
Failing refinement proof
Gluing invariant

To ensure consistency of the refinement we need invariant:

\[ t \in \text{valid} \]

\[ \Rightarrow \]

\[ \text{takeplace} \left[ \text{room}(t) \right] \subseteq \text{authorised}[\text{holder}(t)] \]
Invariant enables PO discharge
But get new failing PO
Source of failing PO
Strengthen guard of refined *RemAuth*
Rational design – what, how, why

• **What** does it achieve?
  
  \textit{if} user \( u \) is in room \( r \),

  \textit{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 that room
What, how, why written in B

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

• **How** does it work?
  
  \[
  t \in \text{valid} \land r = \text{room}(t) \land u = \text{holder}(t)
  \]

• **Why** does it work?
  
  \[
  t \in \text{valid} \Rightarrow \\
  \text{takeplace}[\text{room}(t)] \subseteq \text{authorised}[\text{holder}(t)]
  \]
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
Simple file store example

sets FILE, PAGE, DATA
CONT = PAGE $\rightarrow$ DATA

machine filestore
variables file, dsk

\[\text{initialisation} \]
\[\text{file} := \{\} \quad | \quad \text{dsk} := \{\}\]

\[\text{events} \]
\[\text{CreateFile} \triangleq ...\]

\[\text{WriteFile} \triangleq \quad \text{// set contents of } f \text{ to be } c \quad \text{any } f, c\quad \text{where} \]
\[f \in \text{file} \quad c \in \text{CONT} \quad \text{then} \]
\[\text{dsk}(f) := c \quad \text{end} \]

\[\text{ReadFile} \triangleq \quad \text{// return data in page } p \text{ of } f \quad \text{any } f, p, d! \quad \text{where} \]
\[f \in \text{file} \quad p \in \text{dom}(\text{dsk}(f)) \quad d! = \text{dsk}(f)(p) \quad \text{end} \]
Refinement of file store

• Instead of writing entire contents of a file in one atomic step, each page is written separately

\textbf{machine} filestore2  \\
\textbf{refines} filestore  \\
\textbf{variables} file,dsk,writing,wbuf, sdsk

\textbf{invariant}

\texttt{writing} \subseteq \texttt{file}  \\
\texttt{wbuf} \in \texttt{writing} \rightarrow \texttt{CONT}  \\
\texttt{sdsk} \in \texttt{writing} \rightarrow \texttt{CONT}  \\
\texttt{// shadow disk}
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:** refinement rule does not reflect the connection between then new events and the abstract event
Event refinement diagrams

• Based on diagrammatic notation of *Jackson System Development* (JSD)

• Graphical representation of how abstract atomic events are refined

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

• Adapt JSD notation for our needs
• Diagram represents **atomicity refinement** explicitly and
• Diagram specifies **sequencing constraints** on events
Hierarchical refinement

- **Write(f)**
  - **all(p)**
    - **StartWrite(f)**
    - **PageWrite(f,p)**
      - **all(b)**
        - **StartPage(f,p)**
        - **ByteWrite(f,p,b)**
      - **EndPage(f,p)**
    - **EndWrite(f)**
Replicated data base

- Abstract model
  \[ db \in \text{object} \to \text{DATA} \]

Commit = /* update a set of objects os */
any os, update
where
os \subseteq \text{object} \wedge
update \in (os \to \text{DATA}) \to (os \to \text{DATA})
then
db := db \leftrightarrow \text{update}(os \triangleleft db)
end
Refinement by replicated database

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

Update is by two phase commit:

PreCommit followed by Commit

Global commit if all sites pre-commit
Global abort if at least one site aborts
Mutual Exclusion

At abstract level, update transaction is a choice of 2 atomic events:

Update transaction will commit or abort *but not both*
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**

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

Protocol aborts transaction if some site aborts
Commit and abort affect object locking

• *PreCommit*(t,s) : locks all objects for transaction *t* at site *s*

• *LocalCommit*(t,s)  *LocalAbort*(t,s): release all objects for transaction *t* at site *s*
Introducing messaging

- **Commit**($t$)
  - **Start**($t$)
    - **Broadcast** Start($t$)
    - **Recv** Start($s,t$)
  - **PreCommit**($t,s$)
    - **Send** PreCommit($t,s$)
    - **Recv** PreCommit($t,s$)

All $s$
Where are we going?

• Start with *system-level* model of transaction, independent of architecture/roles

• Then introduced *stages* of a transaction
  — *separation* of normal and error behaviour

• Next we introduce explicit message send/receive
  — this will allow us later to *separate* the requester/responder roles

• Hierarchical diagrams help us to *identify* and *manage* these steps
Architectural/role decomposition

• Explicit message/receive allows to separate requester/responder roles

• We do this by slicing the diagrams
Coordinator behaviour for database

- Commit(t)
- Start(t)
- PreCommit(t,s)
- Broadcast: Start(t)
- Recv: PreCommit(t,s)
- all s
Non-coordinator behaviour for database

Commit(t,s)

Start(t)  PreCommit(t,s)

RecvStart(s,t)  Send PreCommit(t,s)
Important Messages

• Formal modelling can be applied to *systems*
• Role of **formal modelling**:  
  – increase understanding  
  – decrease errors  
• 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  
• **Methods needed**:  
  – stronger guidelines for abstraction, refinement and decomposition needed  
  – good *structures* help to ease their application  
• In practice, refinement is **not** top-down!