UNIVERSITY OF Southampton

An Introduction to The Formal Development and Verification of Software with Event-B/ RODIN

Mike Poppleton
users.ecs.soton.ac.uk/mrp

Slides adapted from Prof. Michael Butler, Marktoberdorf Summer School 2012

UNIVERSITY OF Southampton

Session 1: Problem Abstraction and Model Refinement - An Overview

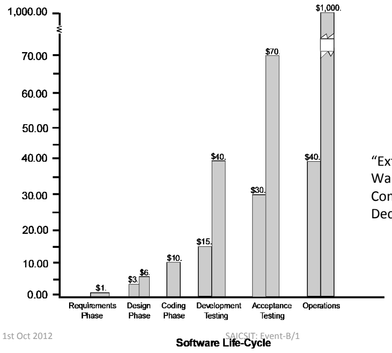
This afternoon:

- Session 2: Verification and tools in Event-B modelling
- Session 3: Case study: the cardiac pacemaker

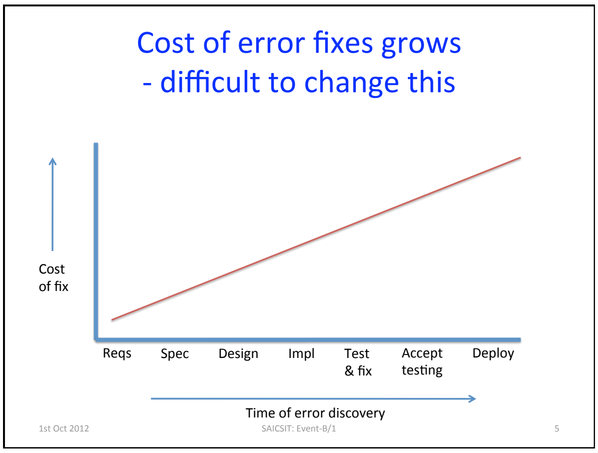
Overview

- 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

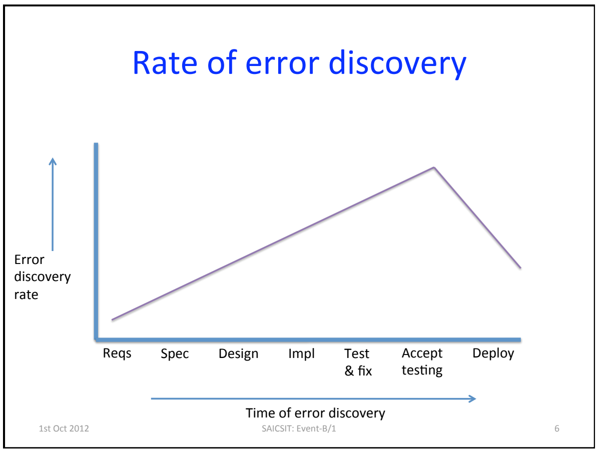


"Extra Time Saves Money"
Warren Kuffel
Computer Language
December 1990



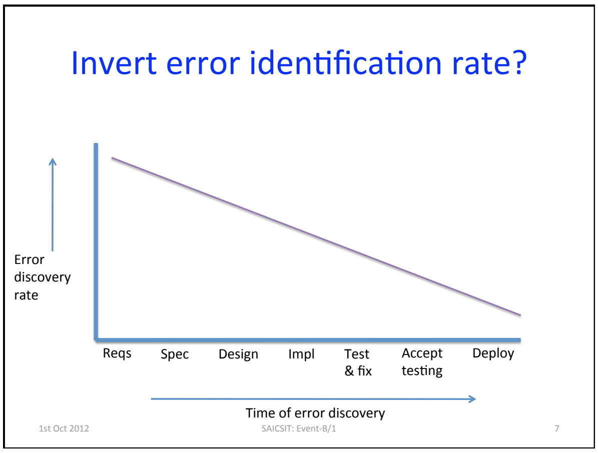
Cost of error fixes grows - difficult to change this

1st Oct 2012 SAICSIT: Event-B/1 5



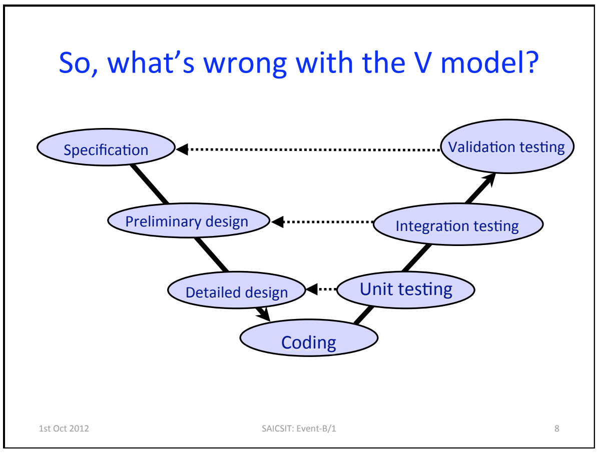
Rate of error discovery

1st Oct 2012 SAICSIT: Event-B/1 6



Invert error identification rate?

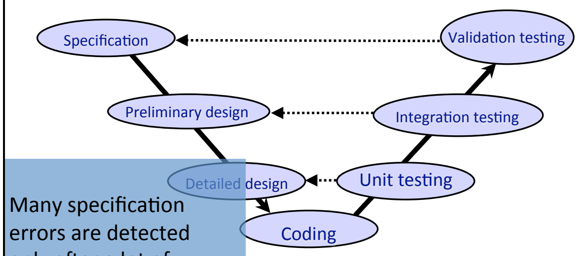
1st Oct 2012 SAICSIT: Event-B/1 7



So, what’s wrong with the V model?

1st Oct 2012 SAICSIT: Event-B/1 8

So, what's wrong with the V model?



Many specification errors are detected only after a lot of development has been undertaken

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 precise models/blueprints

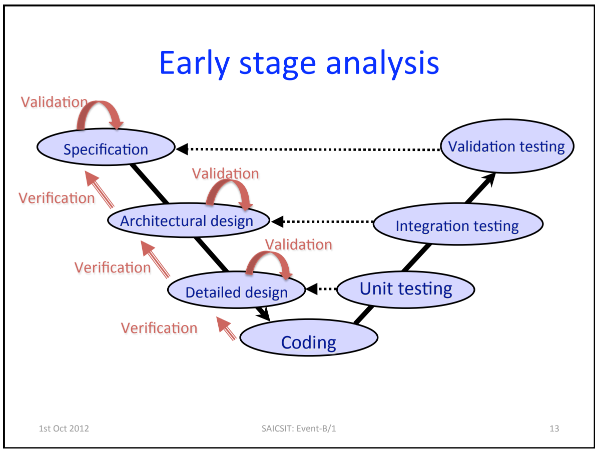
- Early stage analysis
  - Precise descriptions of intent
  - Amenable to analysis by tools
  - Identify and fix ambiguities and inconsistencies as early as possible
- Mastering complexity
  - Encourage abstraction
  - Focus on *what* a system does
  - Early focus on *key / critical* features
  - Incremental analysis and design: separation of concerns

Correctness-by-construction using Formal Methods

- Mathematical techniques for formulation and analysis of systems
- Formal methods facilitate:
  - Clear specifications (contract)
  - Rigorous *validation* and *verification*

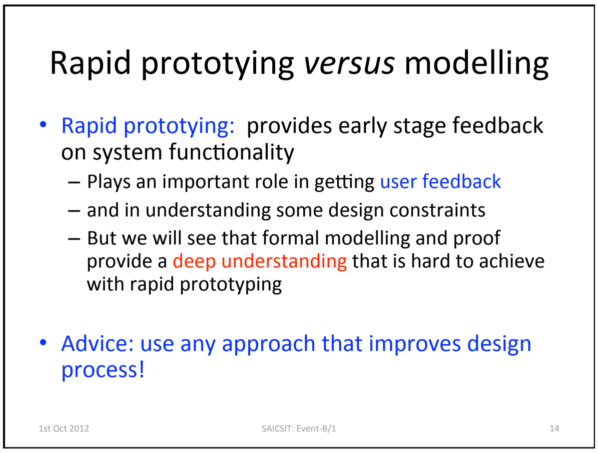
Validation: does the contract specify the right system?
- answered through judgement

Verification: does the finished product satisfy the contract?
- can be answered formally



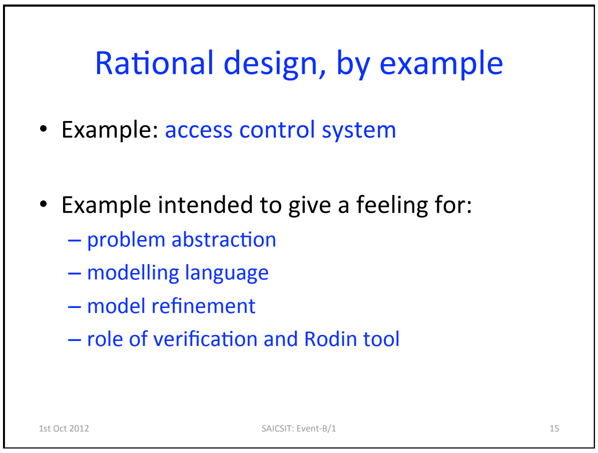
Early stage analysis

Validation
Specification
Validation testing
Verification
Validation
Architectural design
Integration testing
Validation
Verification
Detailed design
Unit testing
Verification
Coding



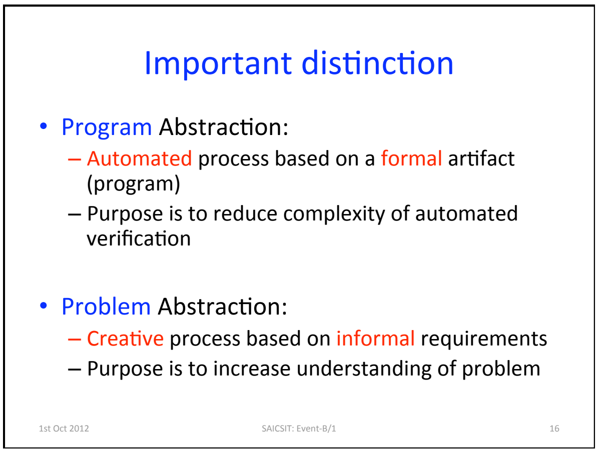
Rapid prototyping *versus* modelling

- 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!



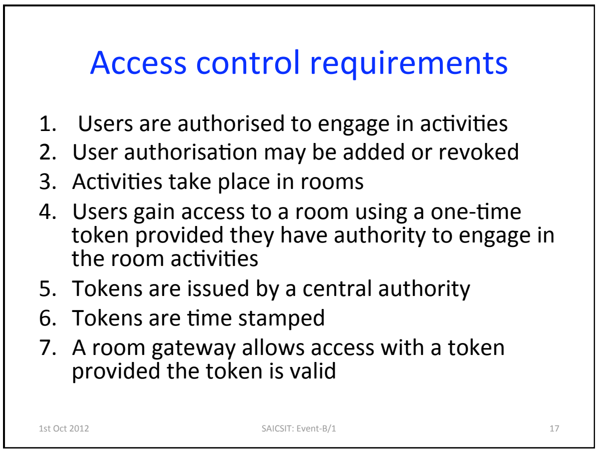
Rational design, by example

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



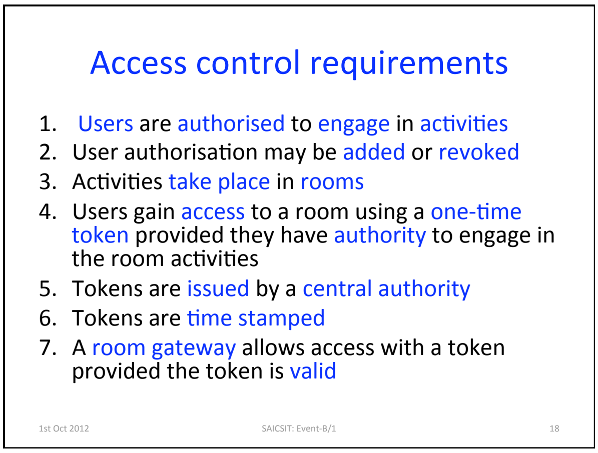
Important distinction

- 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



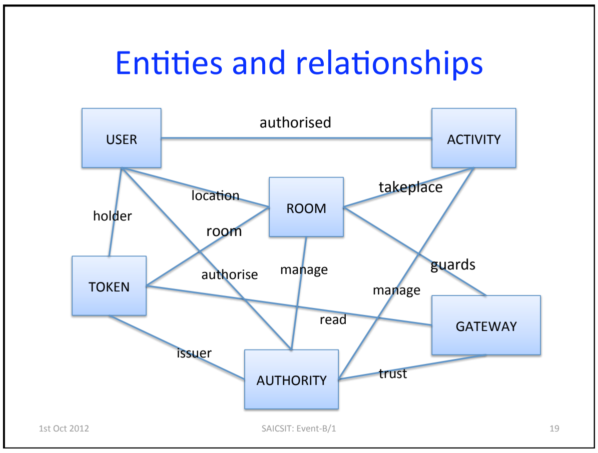
Access control requirements

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

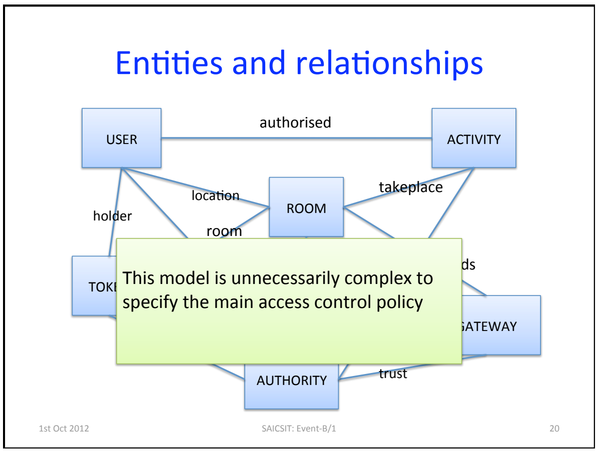


Access control requirements

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

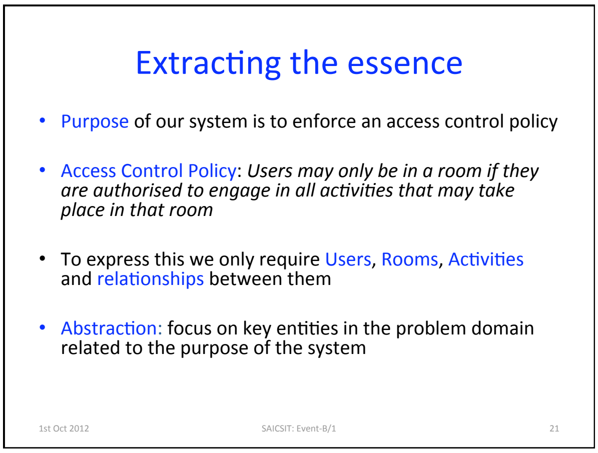


Entities and relationships



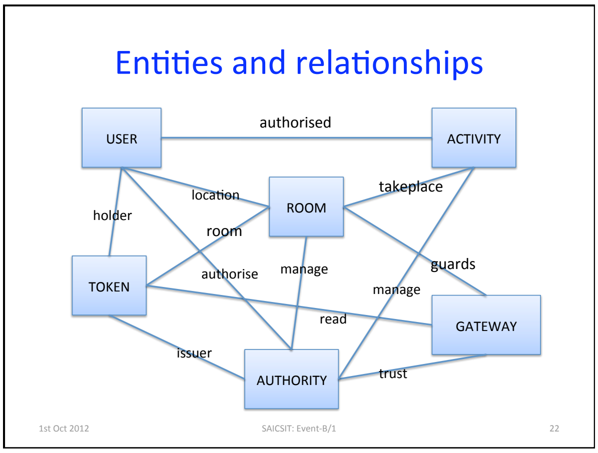
Entities and relationships

This model is unnecessarily complex to specify the main access control policy

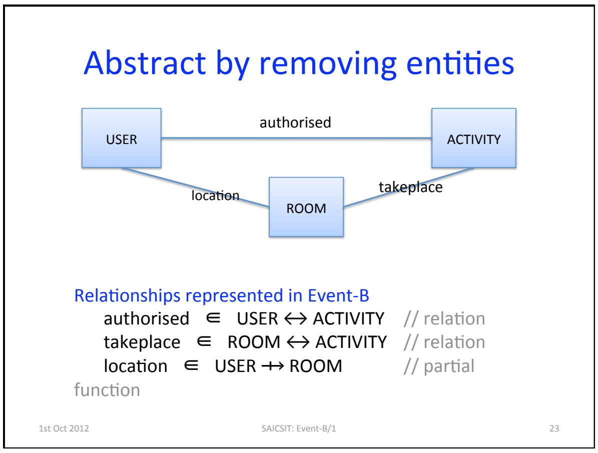


Extracting the essence

- 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



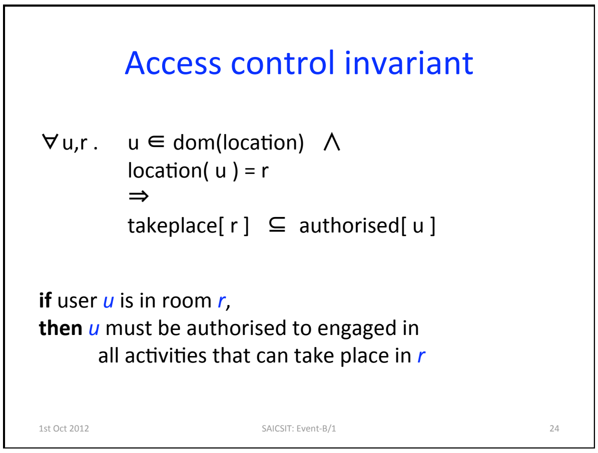
Entities and relationships



Abstract by removing entities

Relationships represented in Event-B

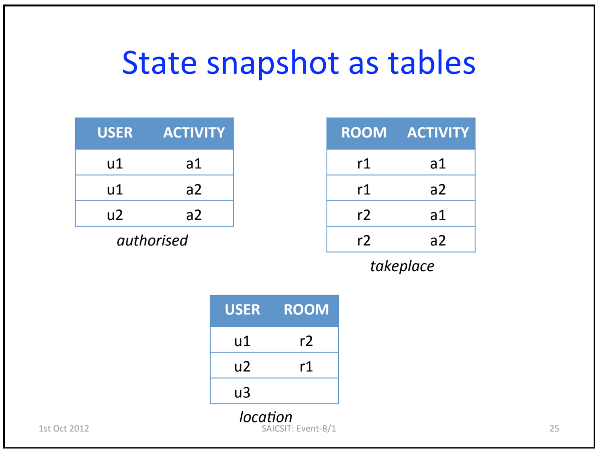
authorised ∈ USER ↔ ACTIVITY // relation
takeplace ∈ ROOM ↔ ACTIVITY // relation
location ∈ USER ⇸ ROOM // partial function



Access control invariant

$$\begin{aligned} \forall u,r .\ & u \in dom(location) \ \wedge \\ & location(\ u\ ) = r \\ & \Rightarrow \\ & takeplace[\ r\ ] \ \subseteq\ authorised[\ u\ ] \end{aligned}$$

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

USER	ACTIVITY
u1	a1
u1	a2
u2	a2

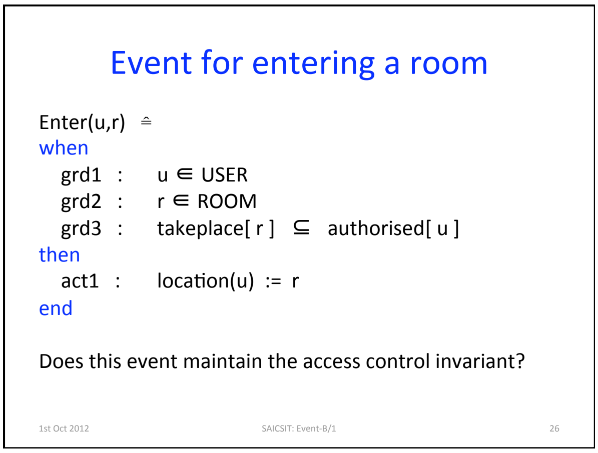
authorised

ROOM	ACTIVITY
r1	a1
r1	a2
r2	a1
r2	a2

takeplace

USER	ROOM
u1	r2
u2	r1
u3	

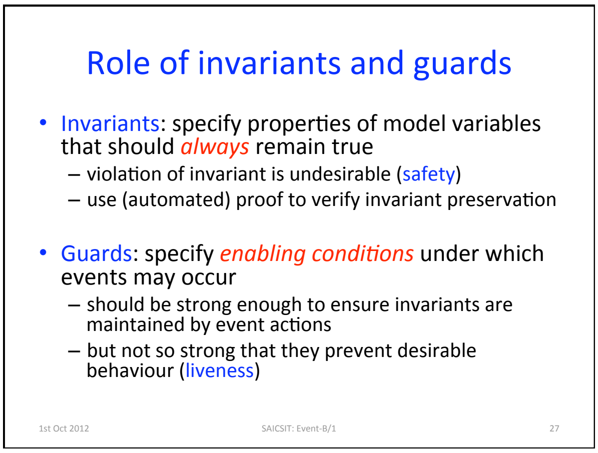
location



Event for entering a room

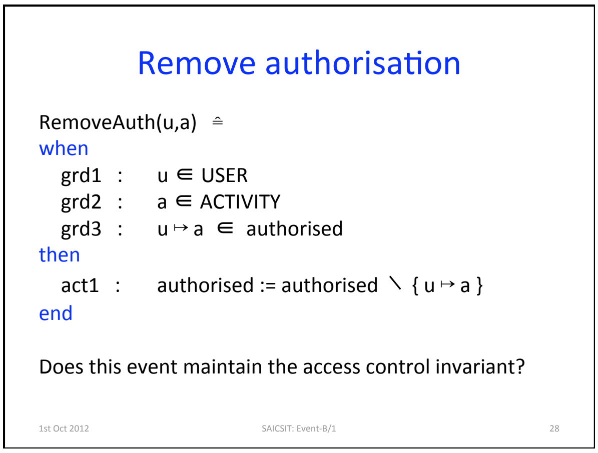
```
Enter(u,r)  ≙
when
   grd1  :    u ∈ USER
   grd2  :    r ∈ ROOM
   grd3  :    takeplace[ r ]  ⊆  authorised[ u ]
then
   act1  :    location(u)  :=  r
end
```

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)

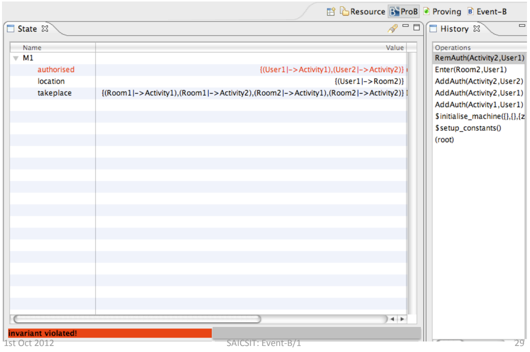


Remove authorisation

```
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

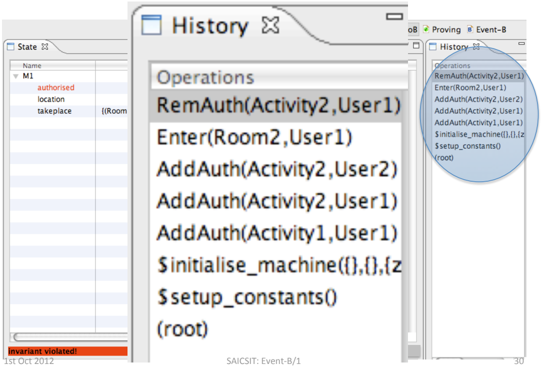


Name	Value
M1	
authorised	{(User1|->Activity1),(User2|->Activity2)}
location	{(User1|->Room2)}
takeplace	{(Room1|->Activity1),(Room1|->Activity2),(Room2|->Activity1),(Room2|->Activity2)}

```
RemAuth(Activity2,User1)
Enter(Room2,User1)
AddAuth(Activity2,User2)
AddAuth(Activity2,User1)
AddAuth(Activity1,User1)
$initialise_machine({},{},{z
$setup_constants()
(root)
```

invariant violated!

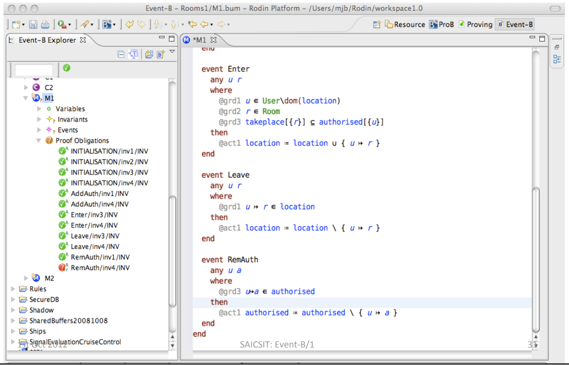
1st Oct 2012 SAICSIT: Event-B/1 29



```
RemAuth(Activity2,User1)
Enter(Room2,User1)
AddAuth(Activity2,User2)
AddAuth(Activity2,User1)
AddAuth(Activity1,User1)
$initialise_machine({},{},{z
$setup_constants()
(root)
```

1st Oct 2012 SAICSIT: Event-B/1 30

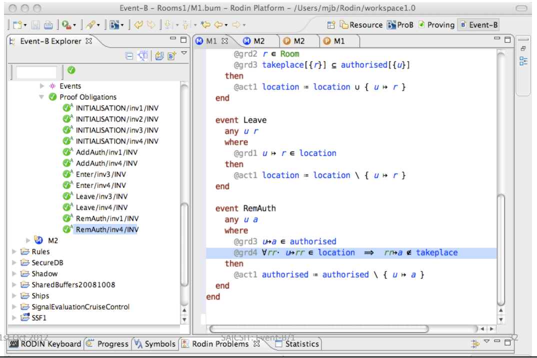
Failing proof



```
end

event Enter
  any u r
  where
    @grd1 u ∈ User\dom(location)
    @grd2 r ∈ Room
    @grd3 takeplace[{r}] ⊆ authorised[{u}]
  then
    @act1 location ≔ location ∪ { u ↦ r }
end

event Leave
  any u r
  where
    @grd1 u ↦ r ∈ location
  then
    @act1 location ≔ location \ { u ↦ r }
end

event RemAuth
  any u a
  where
    @grd3 u↦a ∈ authorised
  then
    @act1 authorised ≔ authorised \ { u ↦ a }
end
end
```

1st Oct 2012 SAICSIT: Event-B/1 31

Strengthen guard of *RemAuth*



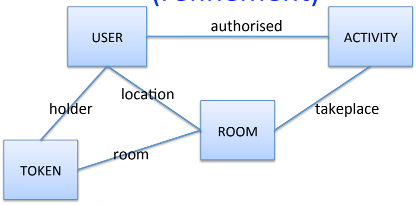
```
    @grd2 r ∈ Room
    @grd3 takeplace[{r}] ⊆ authorised[{u}]
  then
    @act1 location ≔ location ∪ { u ↦ r }
end

event Leave
  any u r
  where
    @grd1 u ↦ r ∈ location
  then
    @act1 location ≔ location \ { u ↦ r }
end

event RemAuth
  any u a
  where
    @grd3 u↦a ∈ authorised
    @grd4 ∀rr· u↦rr ∈ location  ⇒  rr↦a ∉ takeplace
  then
    @act1 authorised ≔ authorised \ { u ↦ a }
end
end
```

1st Oct 2012 SAICSIT: Event-B/1 32

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, lets proceed to another stage of analysis…

We construct a new model (refinement)



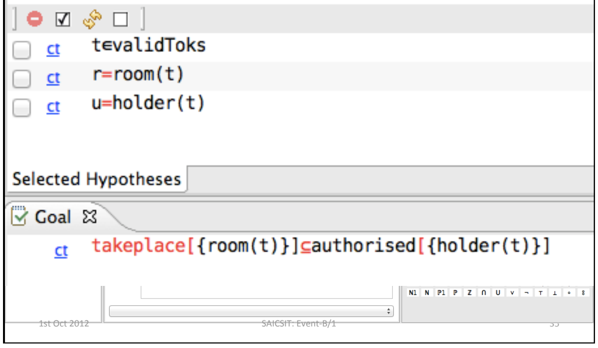
Guard of abstract Enter event:

grd3: takeplace[r] ⊆ authorised[u]

is replaced by a guard on a token:

grd3b: t ∈ valid ∧ room(t) = r ∧ holder(t) = u

Failing refinement proof



□ ct t∈validToks

□ ct r=room(t)

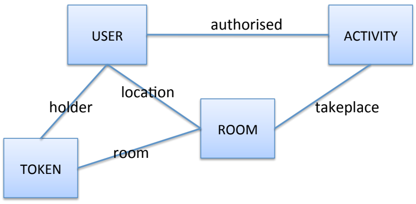
□ ct u=holder(t)

Selected Hypotheses

Goal ☒

ct takeplace[{room(t)}]⊆authorised[{holder(t)}]

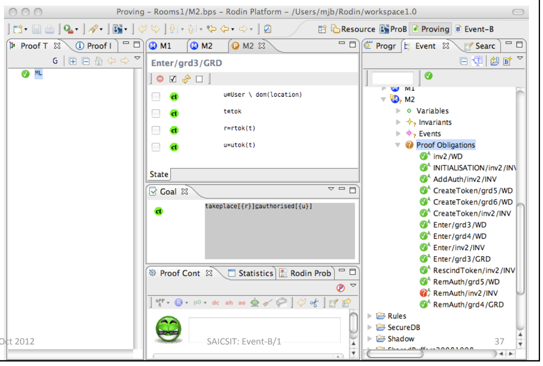
Gluing invariant



To ensure consistency of the refinement we need invariant:

inv 6: t ∈ valid
⇒
takeplace [room(t)] ⊆ authorised[holder(t)]

Invariant enables PO discharge



Enter/grd3/GRD

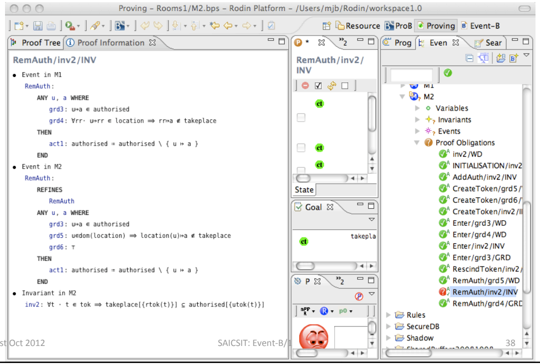
```
u∈User \ dom(location)
t∈tok
r=rtok(t)
u=utok(t)
```

Goal:

```
takeplace[{r}]⊆authorised[{u}]
```

1st Oct 2012 SAICSIT: Event-B/1 37

But get new failing PO

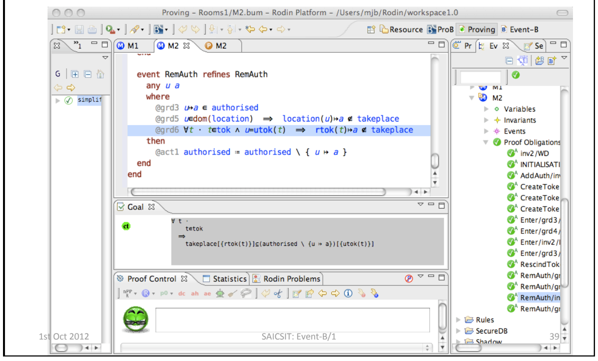


RemAuth/inv2/INV

```
• Event in M1
  RemAuth:
    ANY u, a WHERE
      grd3: u↦a ∈ authorised
      grd4: ∀rr· u↦rr ∈ location ⇒ rr↦a ∉ takeplace
    THEN
      act1: authorised ≔ authorised \ { u ↦ a }
    END
• Event in M2
  RemAuth:
    REFINES
      RemAuth
    ANY u, a WHERE
      grd3: u↦a ∈ authorised
      grd5: u∈dom(location) ⇒ location(u)↦a ∉ takeplace
      grd6: ⊤
    THEN
      act1: authorised ≔ authorised \ { u ↦ a }
    END
• Invariant in M2
  inv2: ∀t · t ∈ tok ⇒ takeplace[{rtok(t)}] ⊆ authorised[{utok(t)}]
```

1st Oct 2012 SAICSIT: Event-B/1 38

Strengthen guard of refined *RemAuth*



```
event RemAuth refines RemAuth
  any u a
  where
    @grd3 u↦a ∈ authorised
    @grd5 u∈dom(location)  ⇒  location(u)↦a ∉ takeplace
    @grd6 ∀t · t∈tok ∧ u=utok(t)  ⇒  rtok(t)↦a ∉ takeplace
  then
    @act1 authorised ≔ authorised \ { u ↦ a }
  end
end
```

Goal:

```
∀ t ·
  t∈tok
 ⇒
  takeplace[{rtok(t)}]⊆(authorised \ {u ↦ a})[{utok(t)}]
```

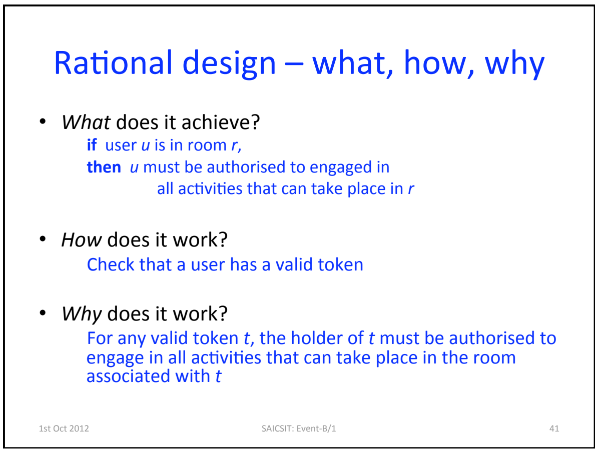
1st Oct 2012 SAICSIT: Event-B/1 39

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?

1st Oct 2012 SAICSIT: Event-B/1 40

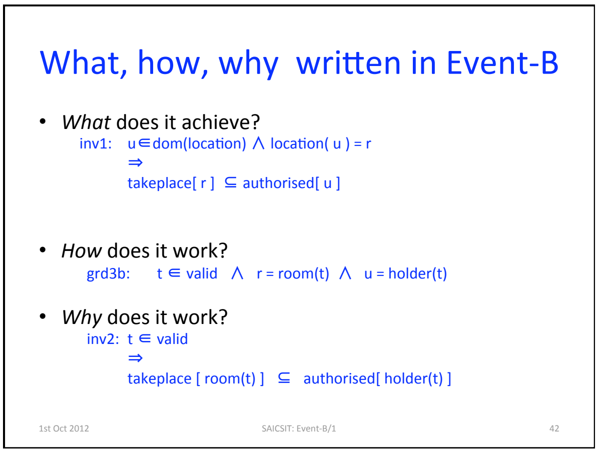


Rational design – what, how, why

- *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*

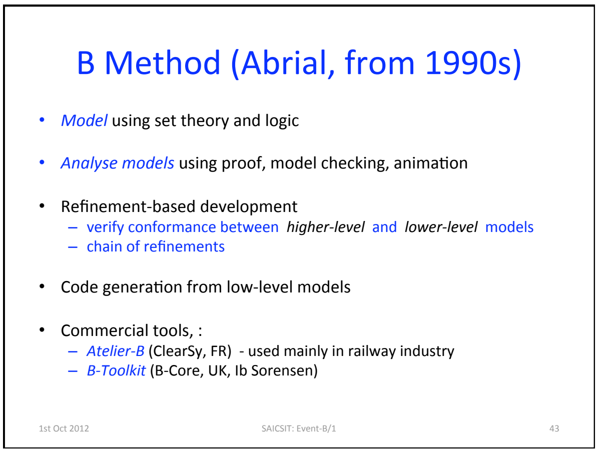


What, how, why written in Event-B

- *What* does it achieve?
  inv1: $u \in dom(location) \wedge location(u) = r$
  $\Rightarrow$
  $takeplace[r] \subseteq authorised[u]$

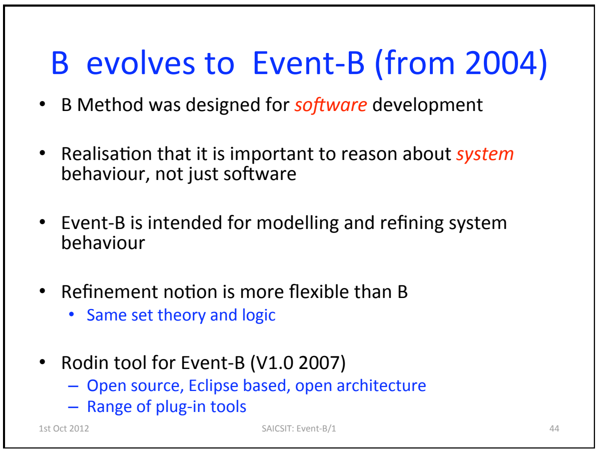
- *How* does it work?
  grd3b: $t \in valid \wedge r = room(t) \wedge u = holder(t)$

- *Why* does it work?
  inv2: $t \in valid$
  $\Rightarrow$
  $takeplace[room(t)] \subseteq authorised[holder(t)]$



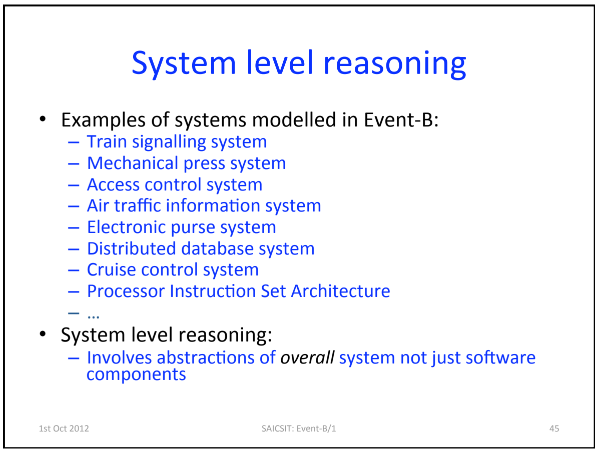
B Method (Abrial, from 1990s)

- *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)



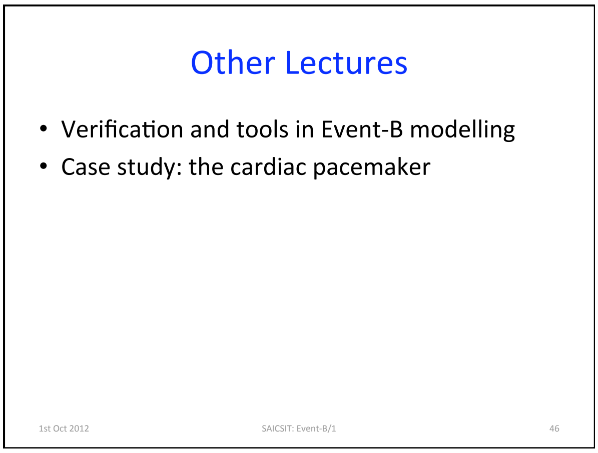
B evolves to Event-B (from 2004)

- 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



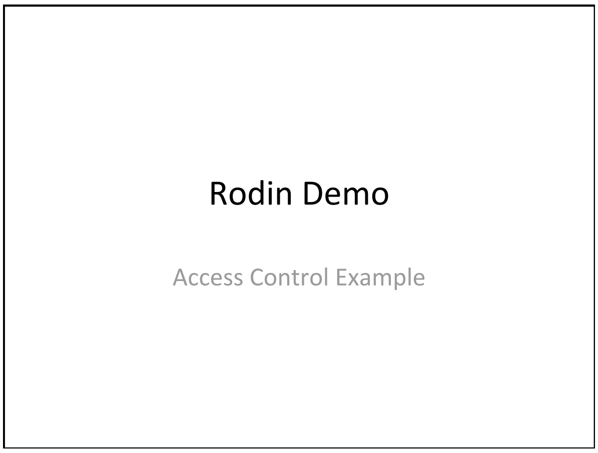
System level reasoning

- 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



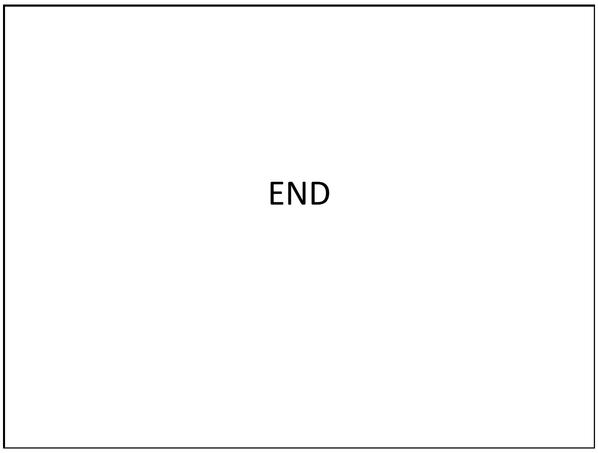
Other Lectures

- Verification and tools in Event-B modelling
- Case study: the cardiac pacemaker



Rodin Demo

Access Control Example



END