UNIVERSITY OF Southampton

Abstraction, Refinement and Decomposition for Systems Engineering

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

Marktoberdorf Summer School 2012

UNIVERSITY OF Southampton

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

UNIVERSITY OF Southampton

Lecture 1: Problem Abstraction and Model Refinement - An Overview

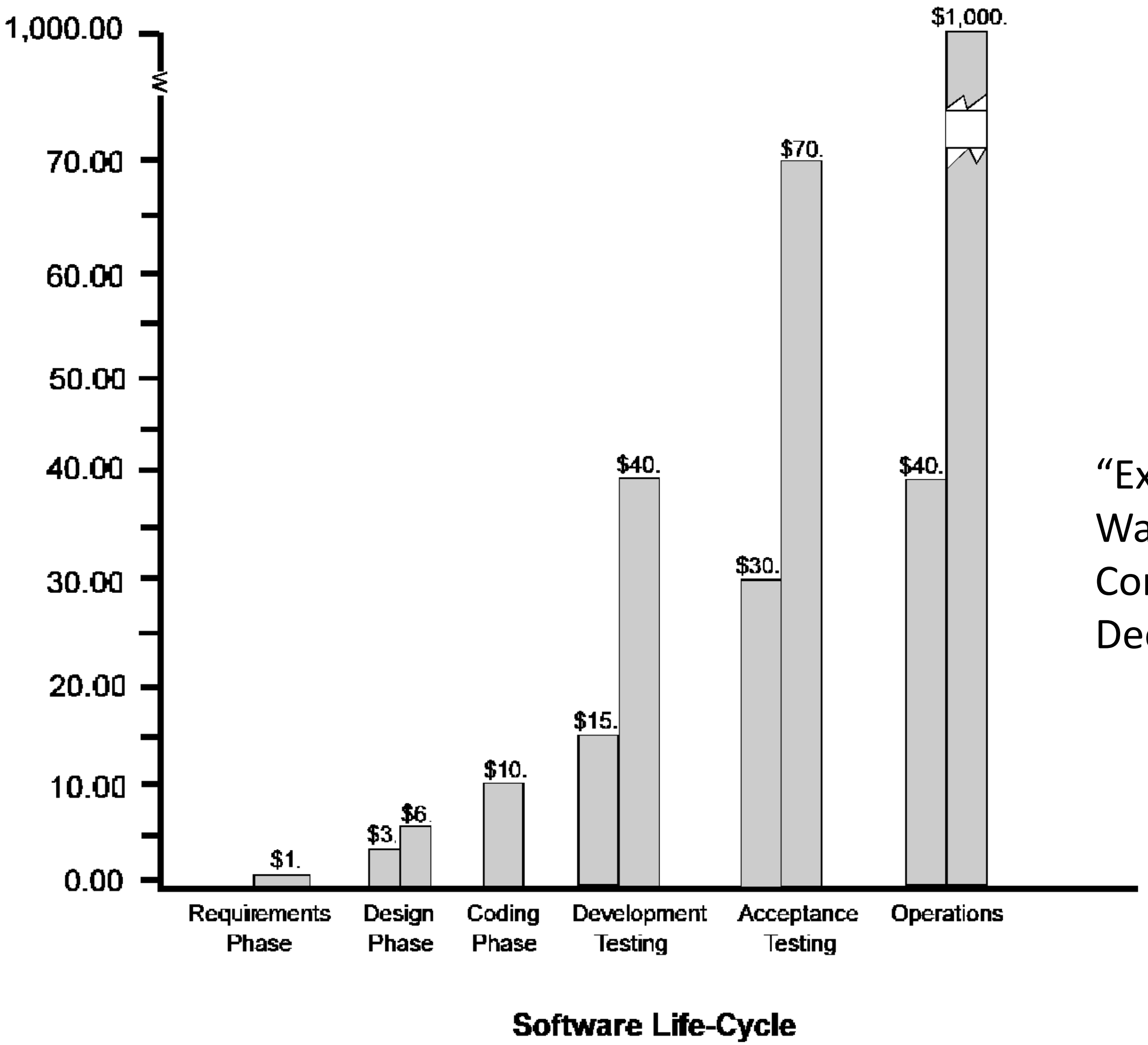
Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf Summer School 2012

Overview

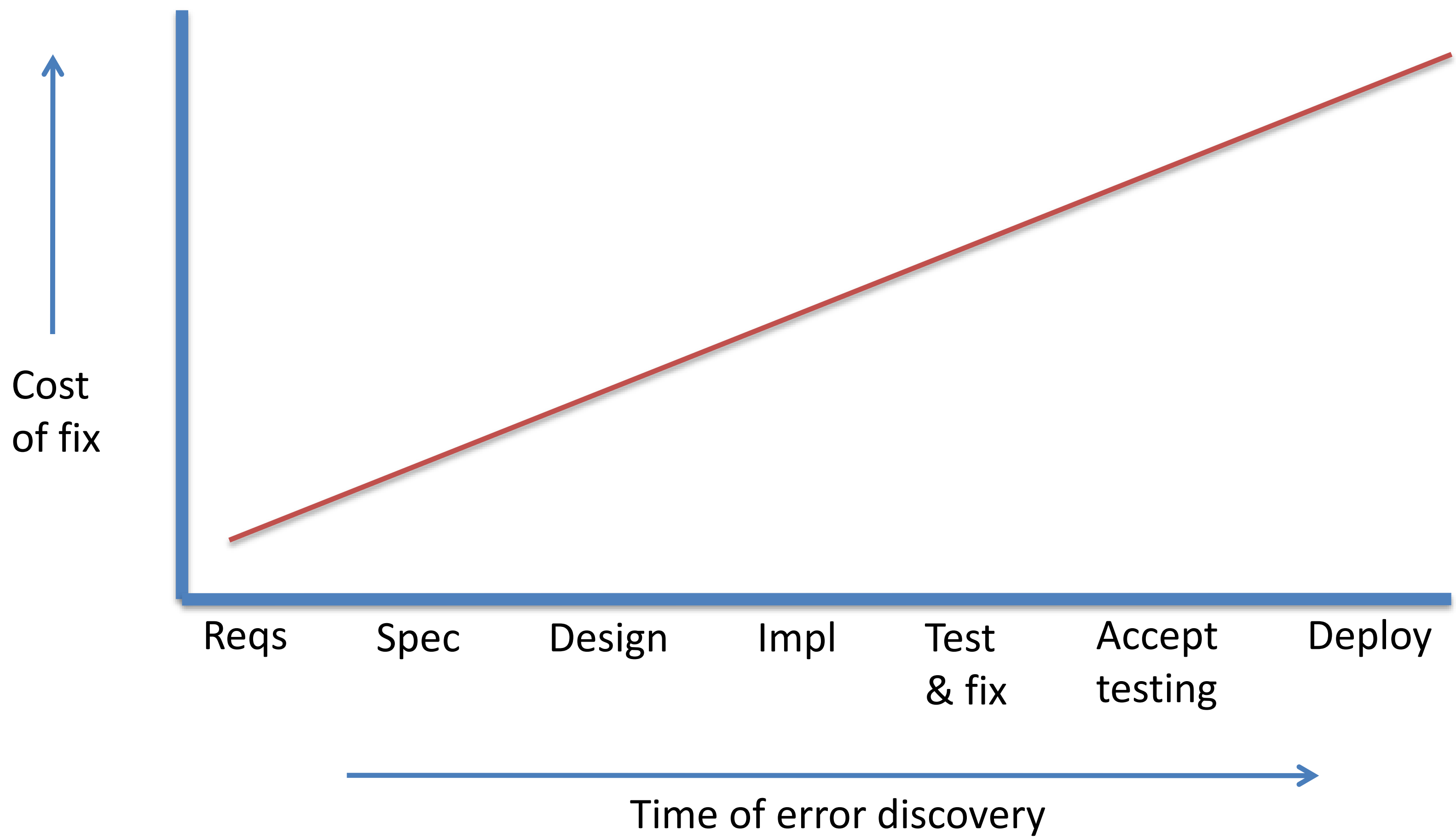
- Motivation
  - difficulty of discovering errors / cost of fixing errors
- Small pedagogical example (access control)
  - abstraction
  - refinement
  - automated analysis
- Background on Event-B formal method
- Methodological considerations

Cost of fixing requirements errors

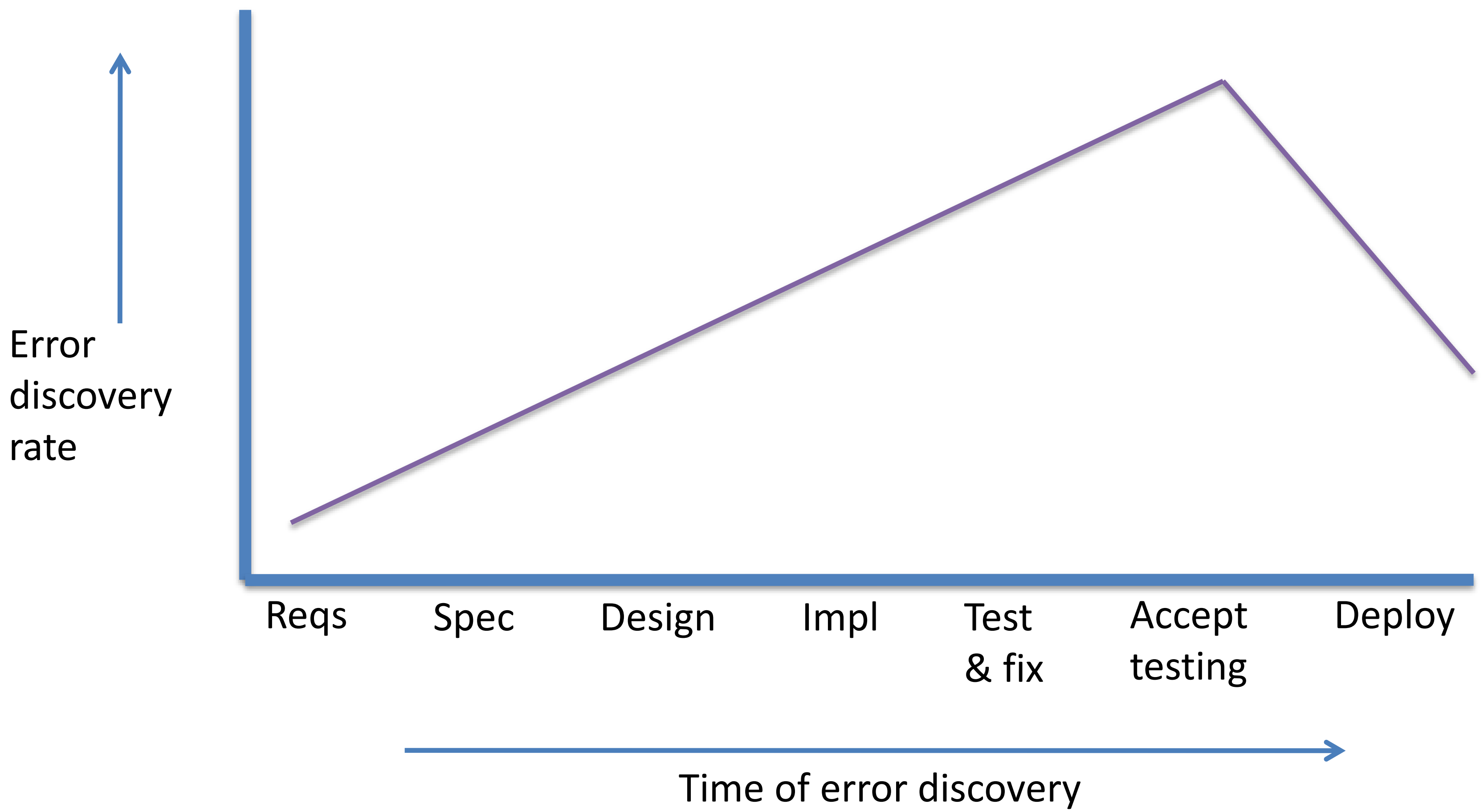


"Extra Time Saves Money"
Warren Kuffel
Computer Language
December 1990

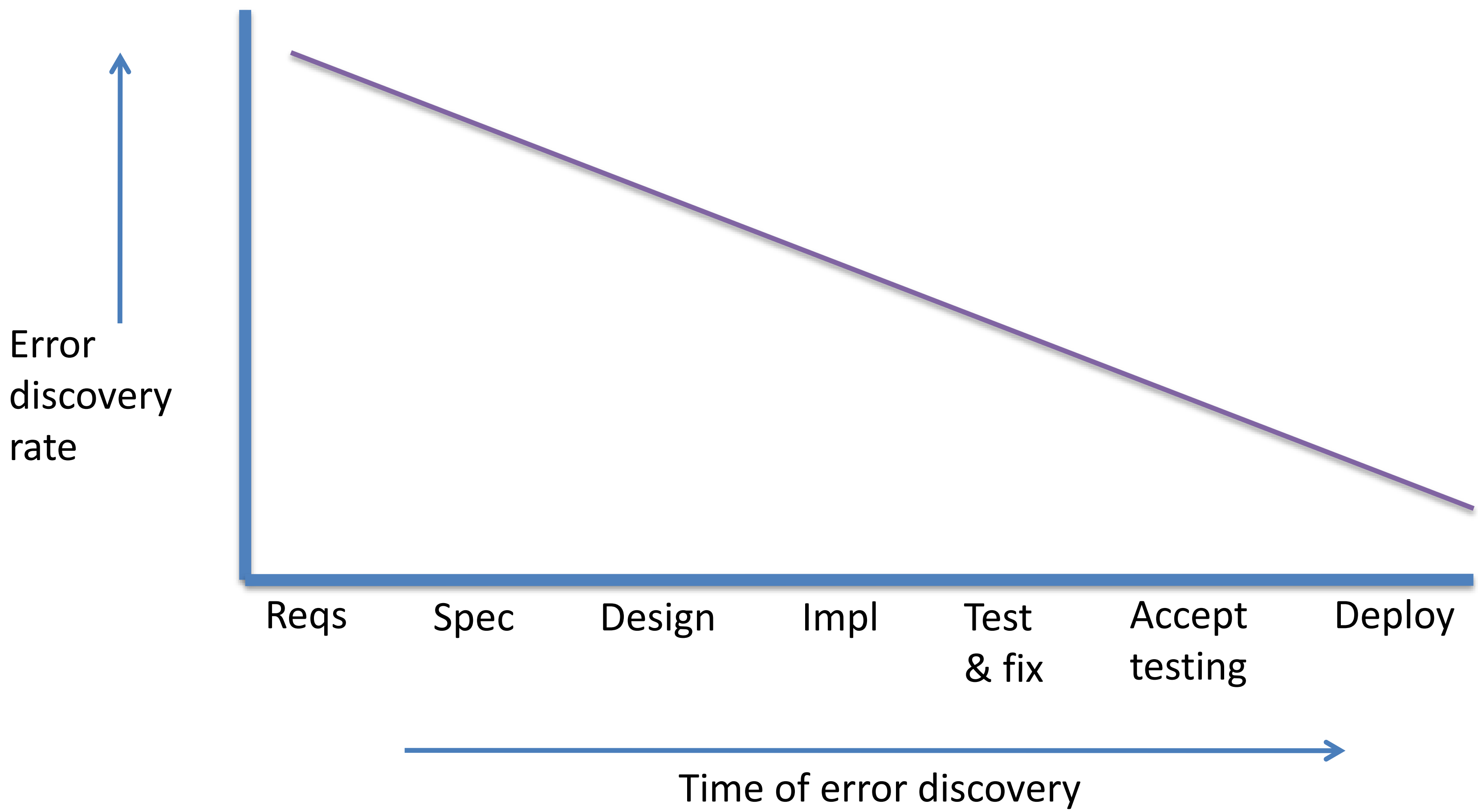
Cost of error fixes grows
- difficult to change this



Rate of error discovery



Invert error identification rate?



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

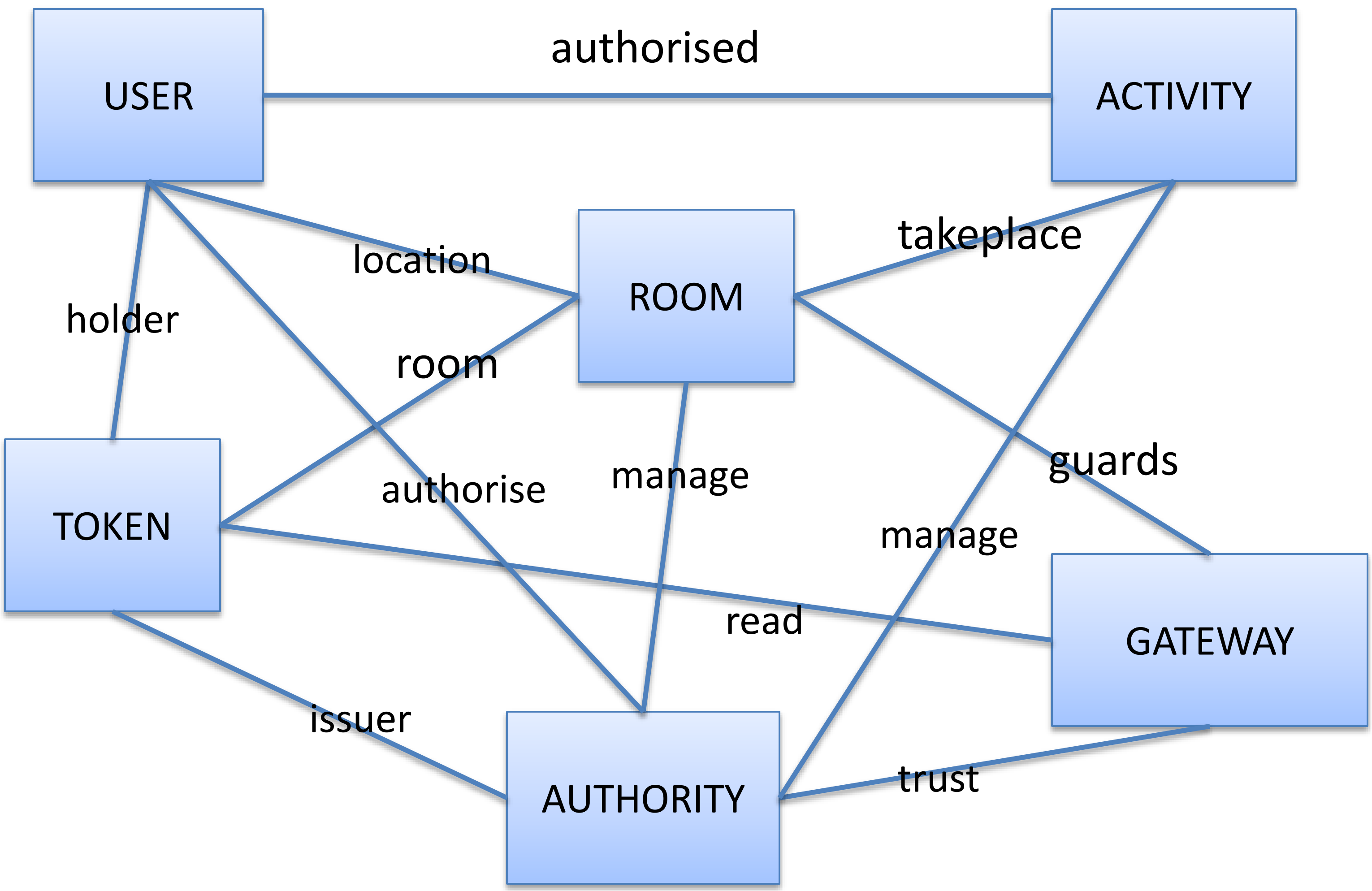
Access control requirements

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

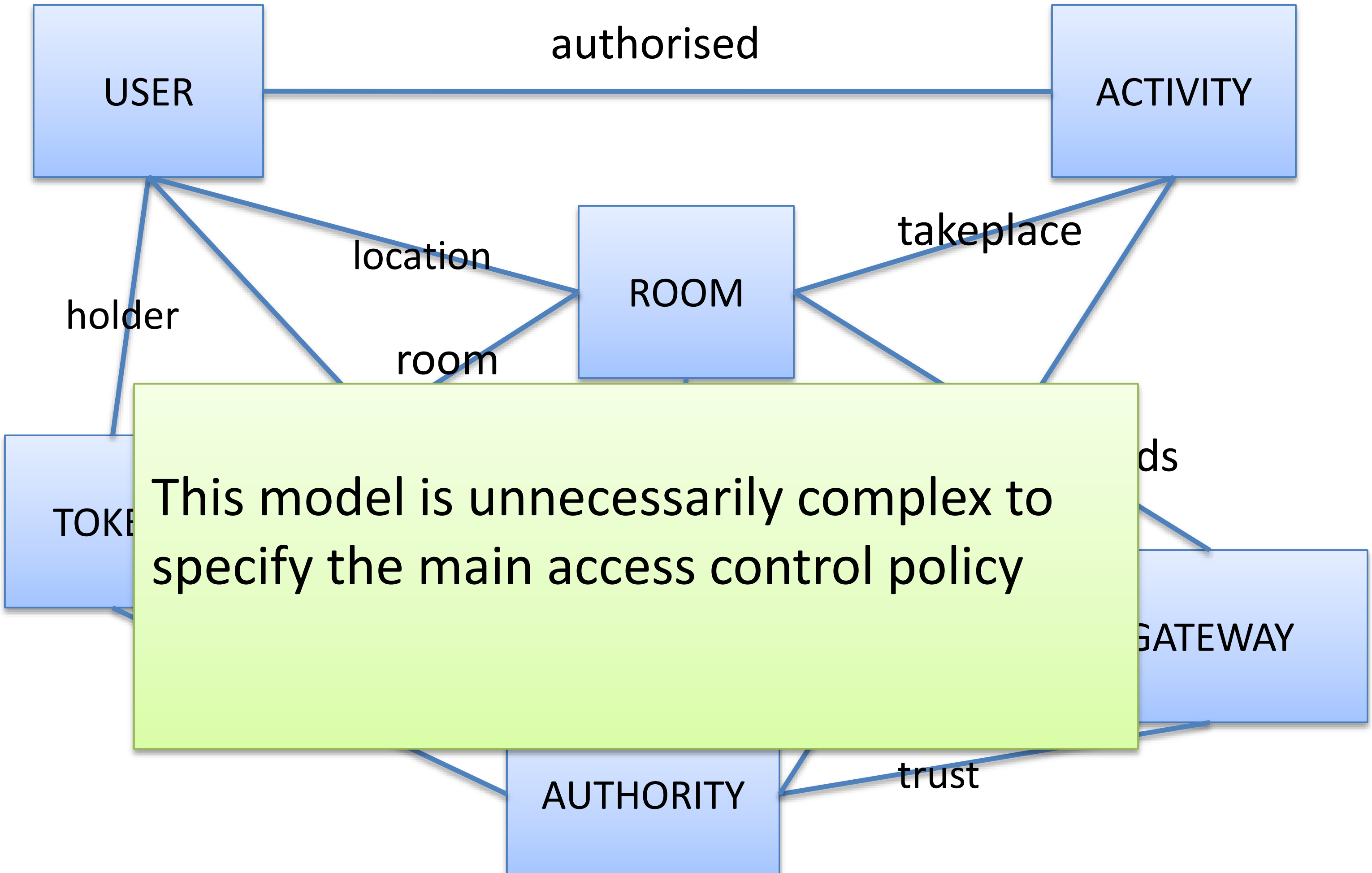
Access control requirements

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



Entities and relationships

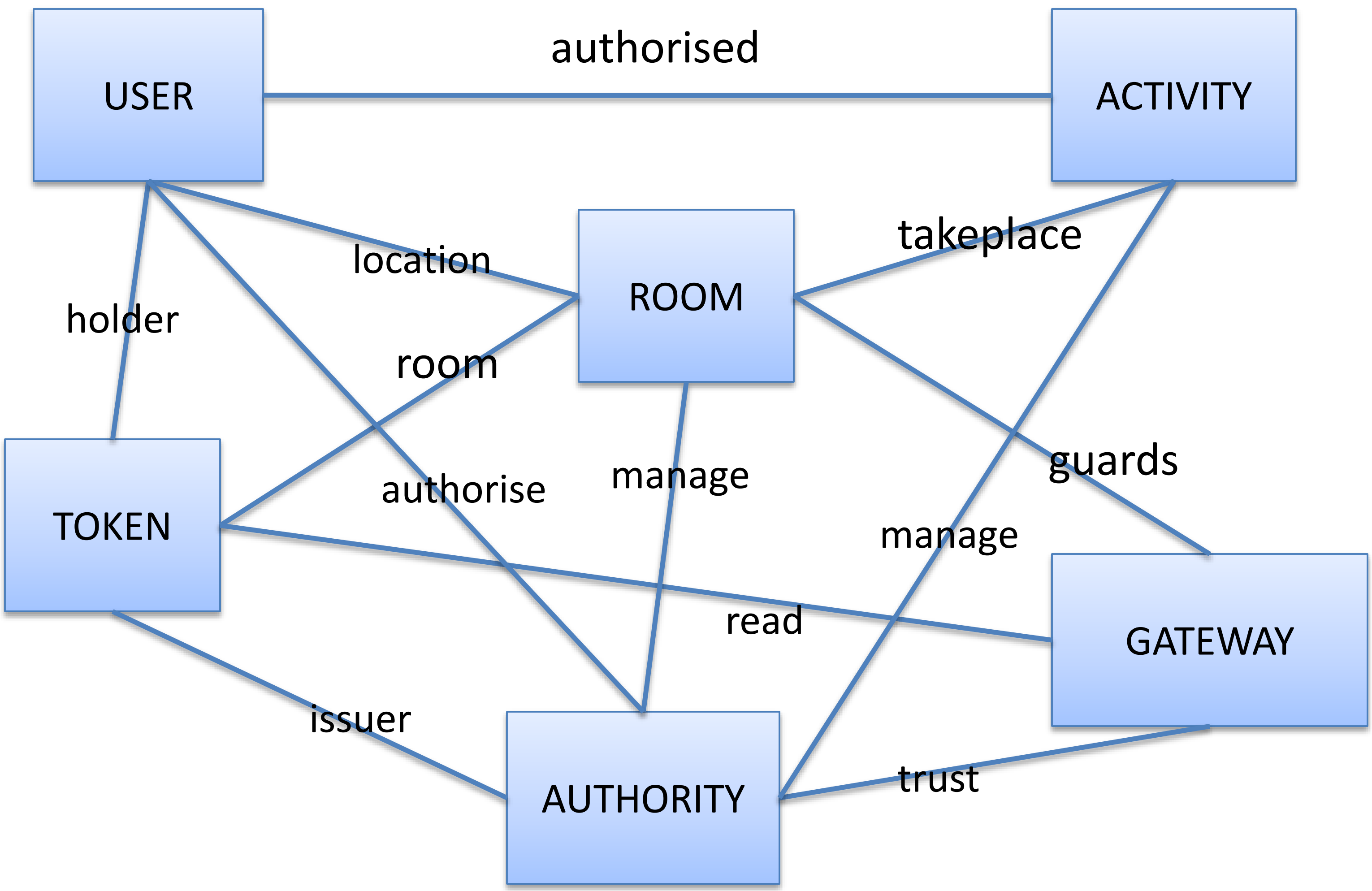


This model is unnecessarily complex to specify the main access control policy

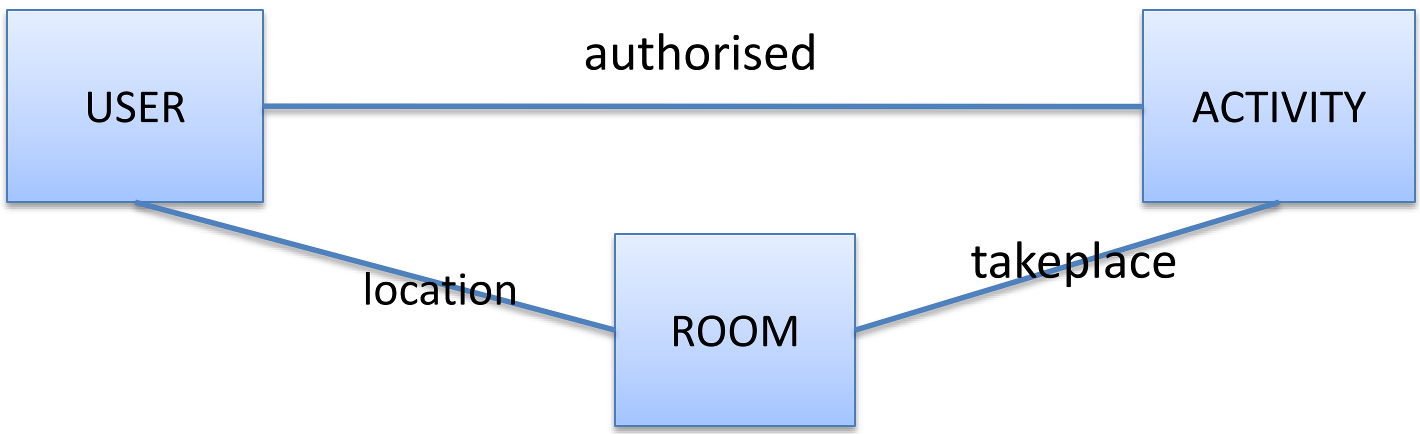
Extracting the essence

- 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

$authorised \in USER \leftrightarrow ACTIVITY$ // relation

$takeplace \in ROOM \leftrightarrow ACTIVITY$ // relation

$location \in USER \pfun ROOM$ // partial function

Access control invariant

$$\forall u,r \,.\; u \in \mathrm{dom}(location) \;\wedge$$
$$location(\,u\,) = r$$
$$\Rightarrow$$
$$takeplace[\,r\,] \;\subseteq\; authorised[\,u\,]$$

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

USER	ACTIVITY
u1	a1
u1	a2
u2	a1

authorised

ROOM	ACTIVITY
r1	a1
r1	a2
r2	a1

takeplace

USER	ROOM
u1	r1
u2	r2
u3	

location

Event for entering a room

Enter(u,r) $\triangleq$

when

grd1 : $u \in USER$

grd2 : $r \in ROOM$

grd3 : $takeplace[\ r\ ] \subseteq authorised[\ u\ ]$

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) $\triangleq$

when

grd1 : $u \in USER$

grd2 : $a \in ACTIVITY$

grd3 : $u \mapsto a \in authorised$

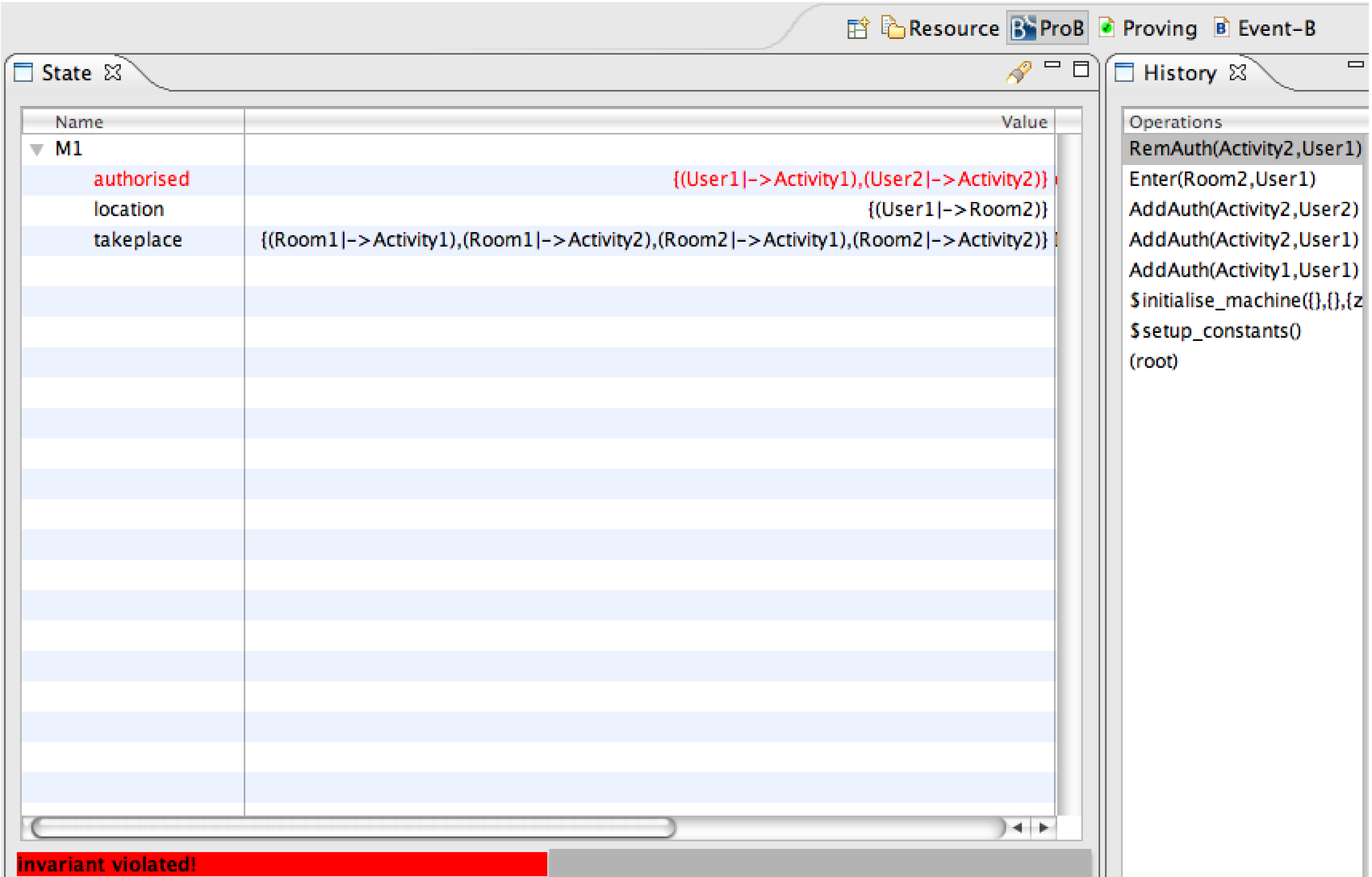
then

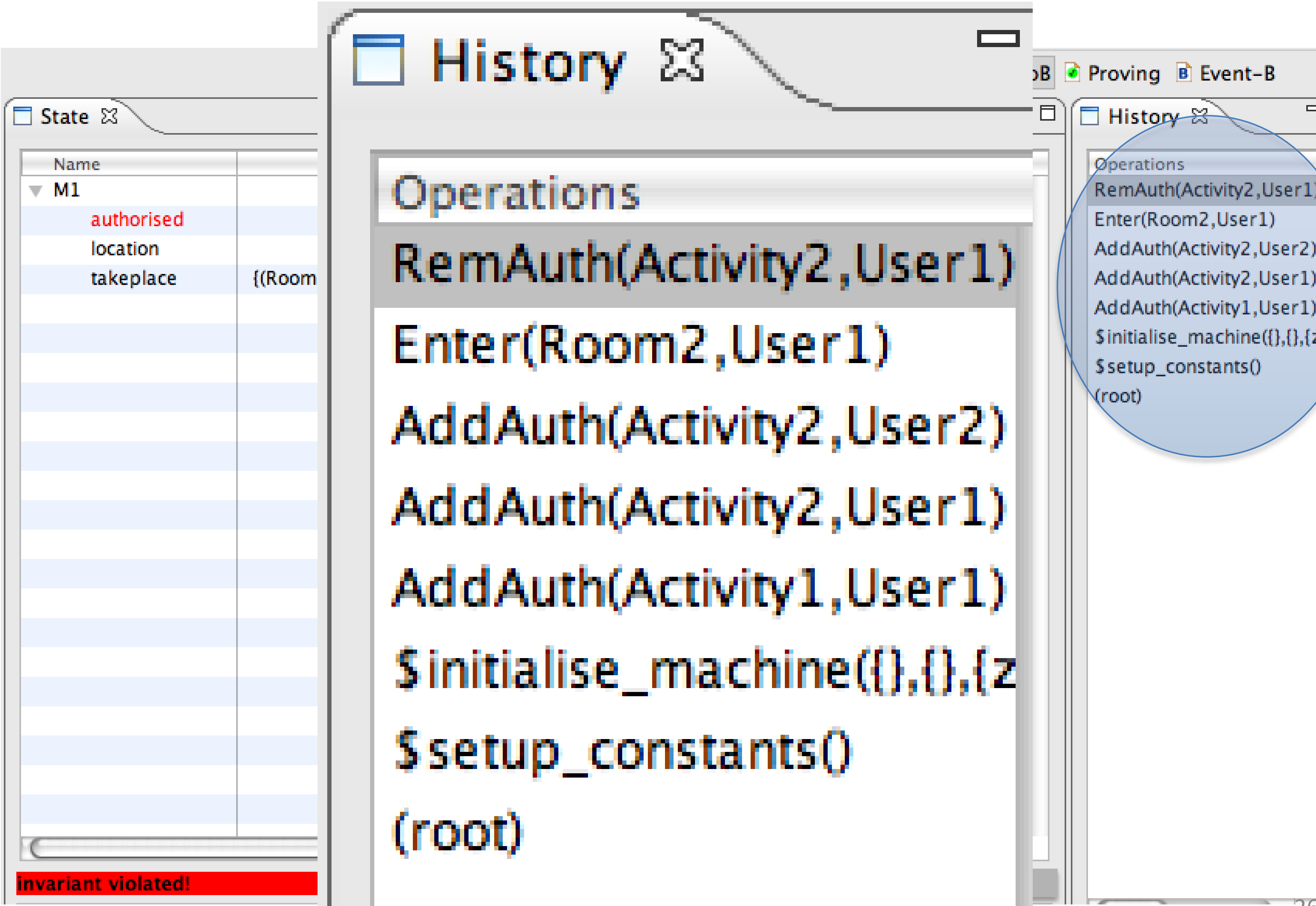
act1 : $authorised := authorised \setminus \{ u \mapsto a \}$

end

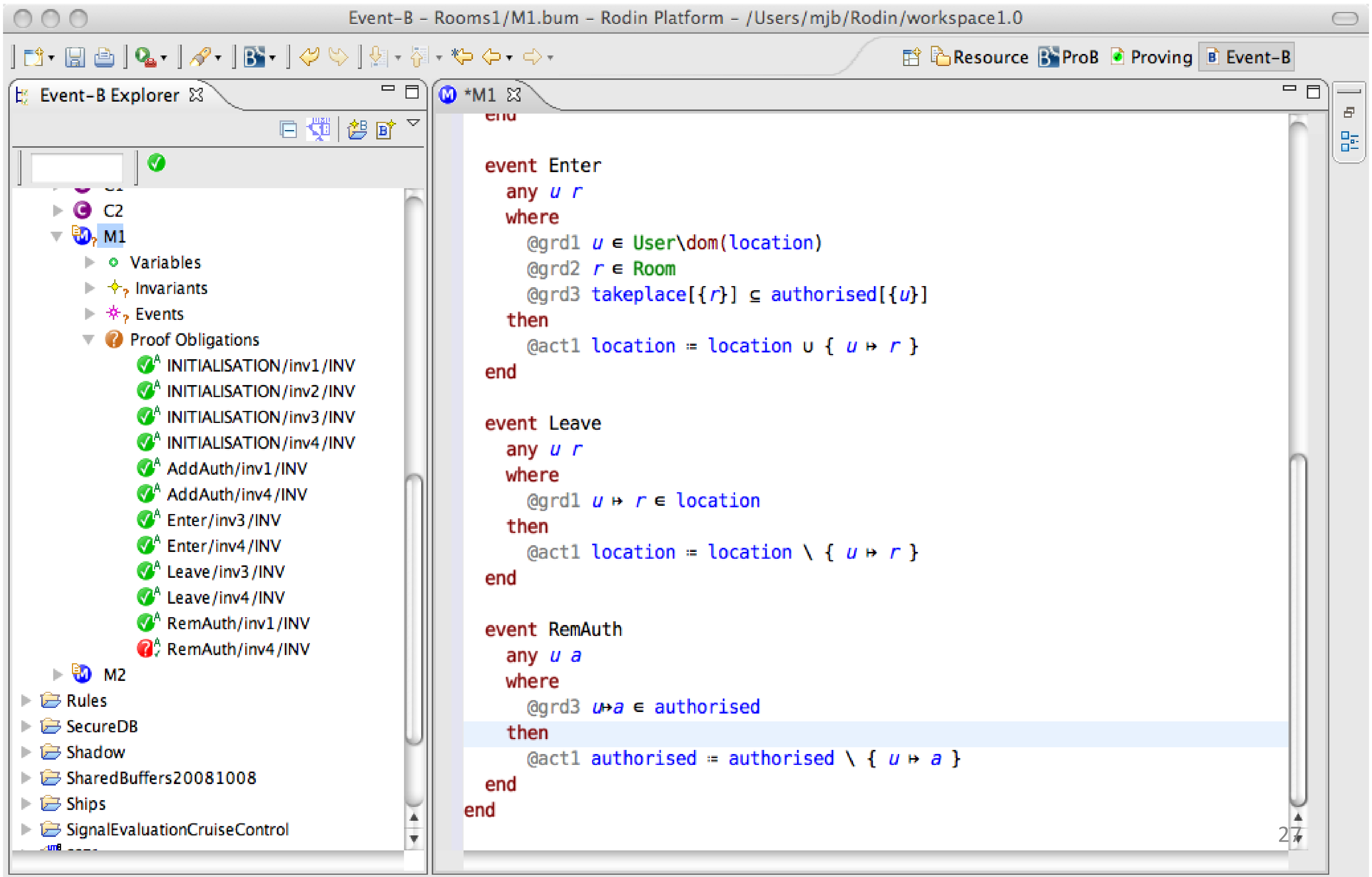
Does this event maintain the access control invariant?

Counter-example from model checking

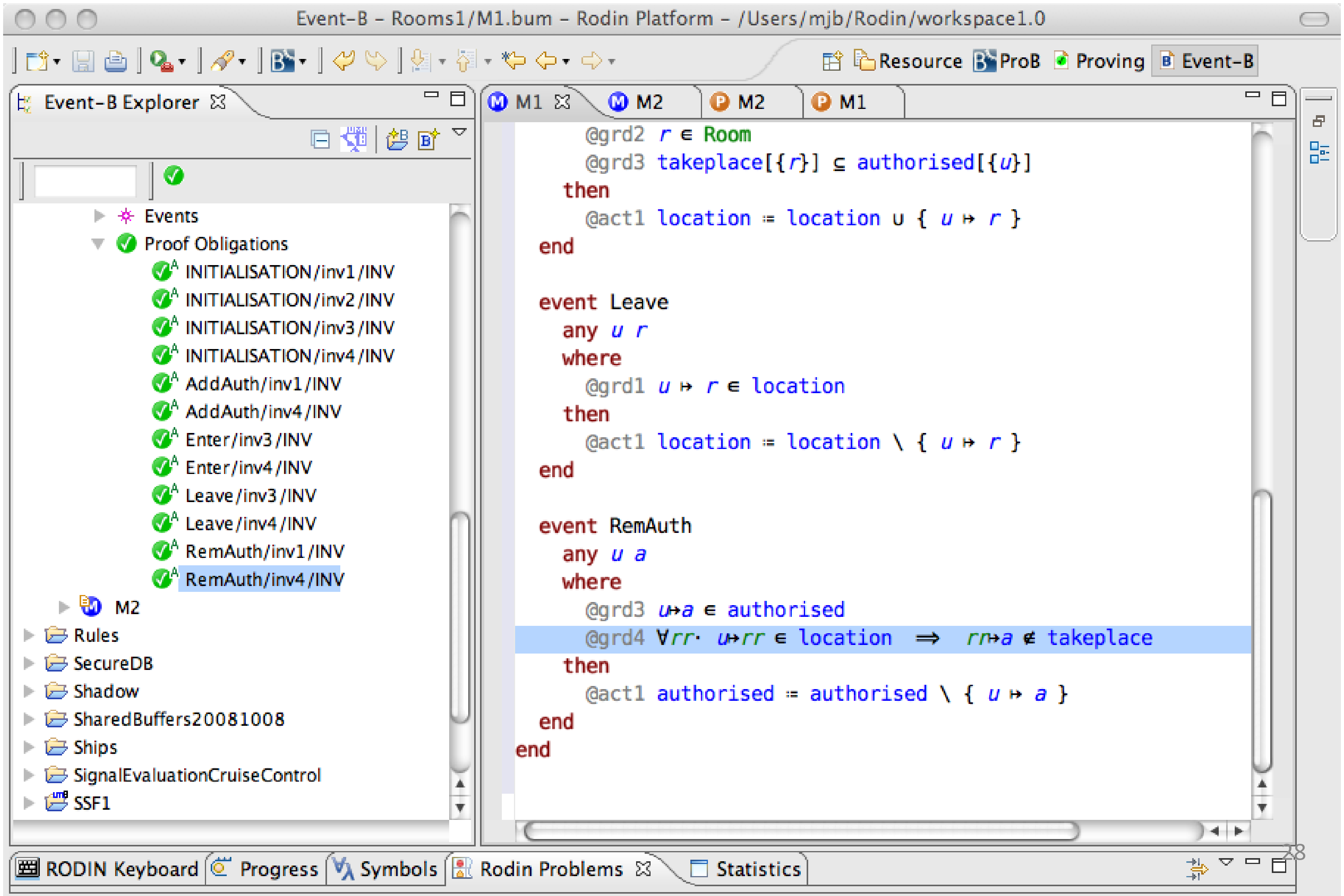



State
Name
M1
authorised
location
takeplace
{(Room
invariant violated!
History
Operations
RemAuth(Activity2,User1)
Enter(Room2,User1)
AddAuth(Activity2,User2)
AddAuth(Activity2,User1)
AddAuth(Activity1,User1)
$initialise_machine({},{},{z
$setup_constants()
(root)
Proving
Event-B

Failing proof



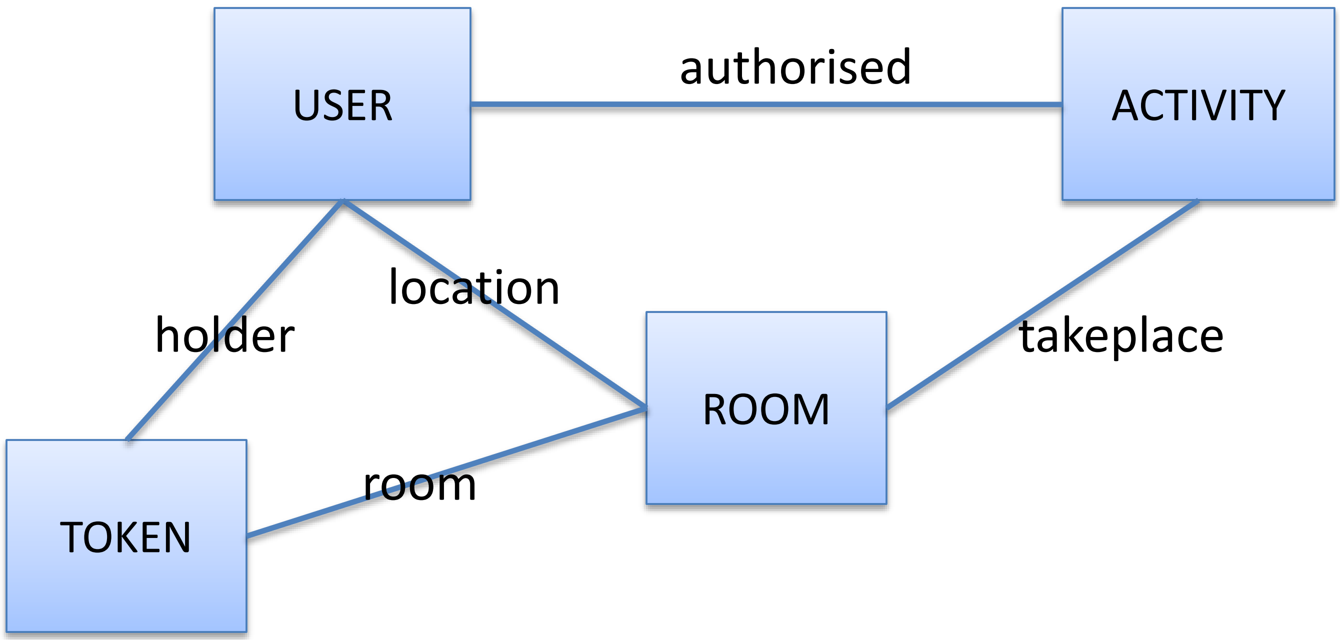
Strengthen guard of *RemAuth*



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, lets proceed to another stage of analysis…

We construct a new model (refinement)



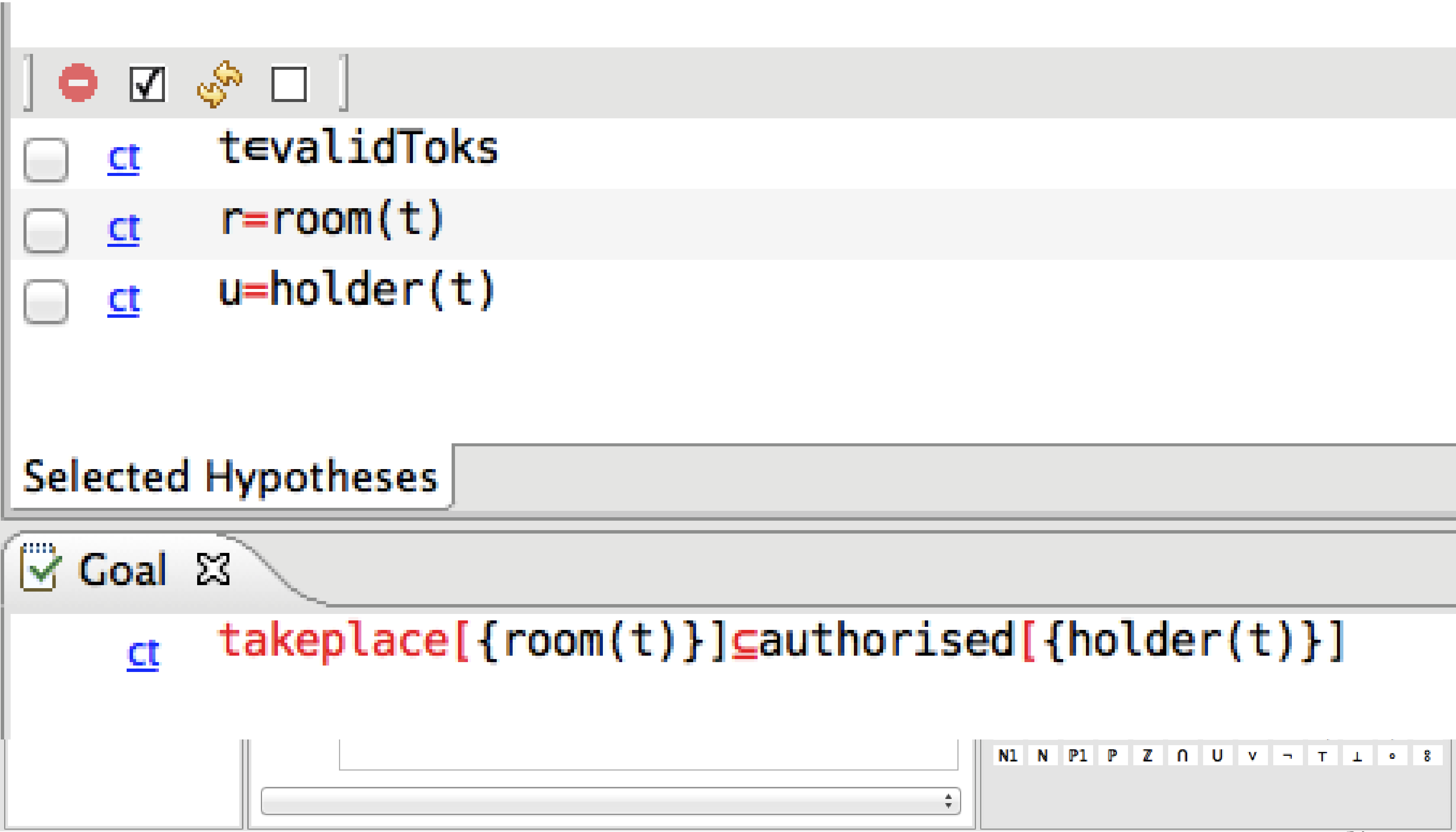
Guard of abstract Enter event:

grd3: takeplace[r] ⊆ authorised[u]

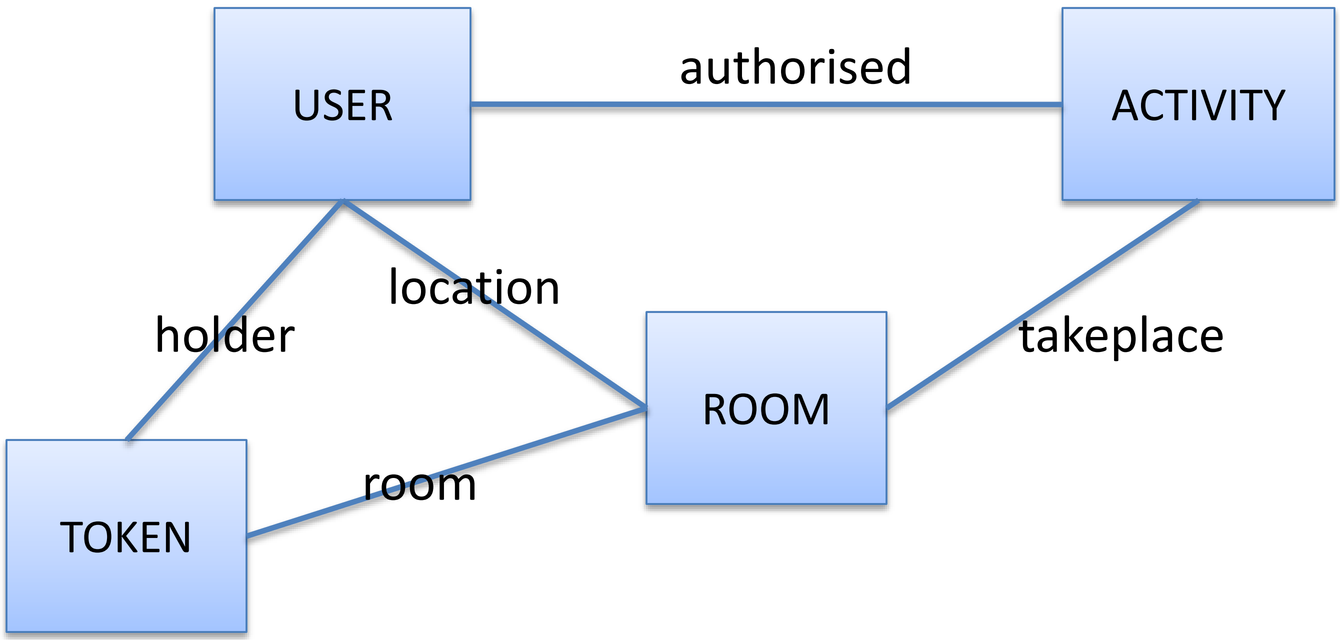
is replaced by a guard on a token:

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

Failing refinement proof



Gluing invariant



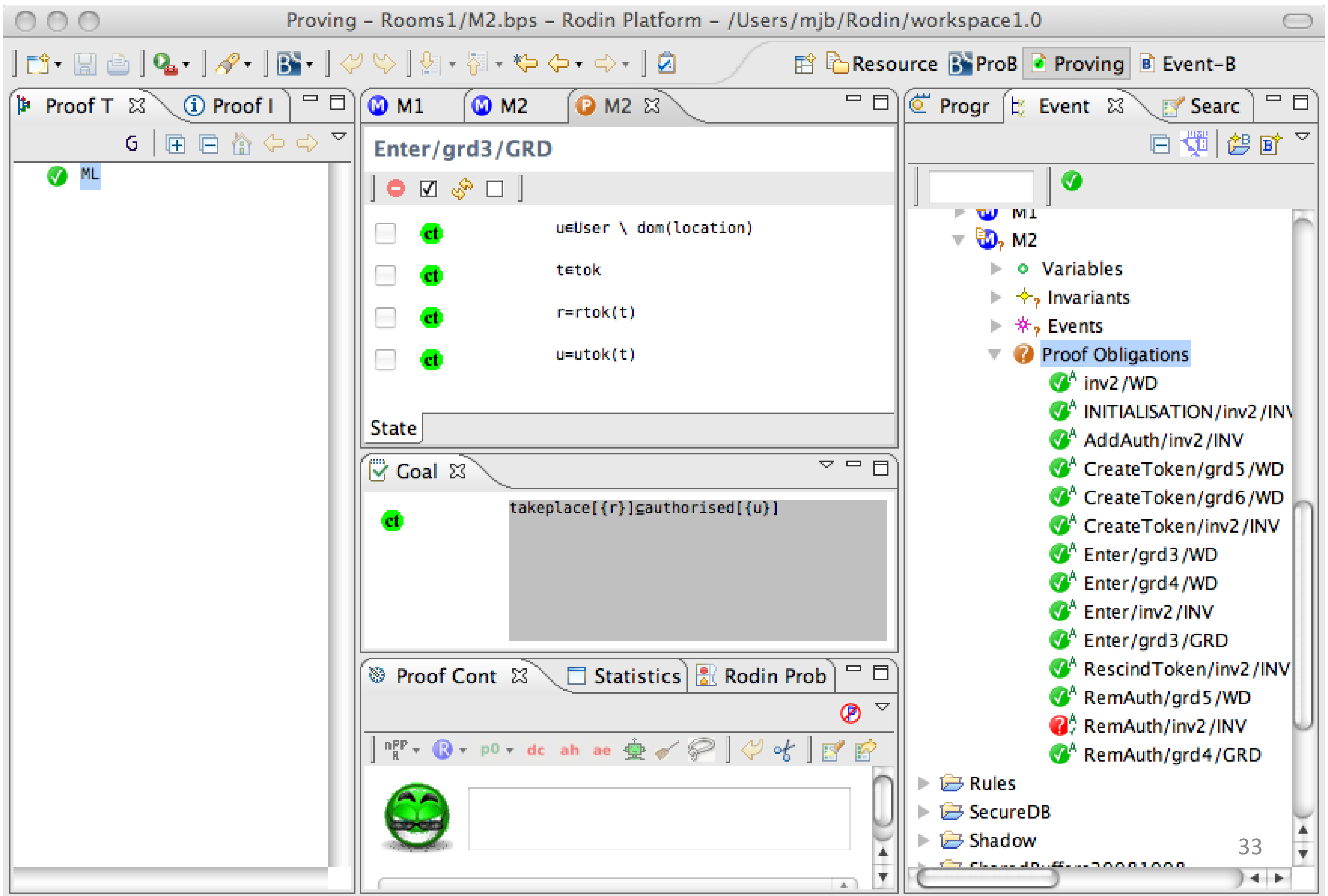
To ensure consistency of the refinement we need invariant:

inv 6: $t \in valid$

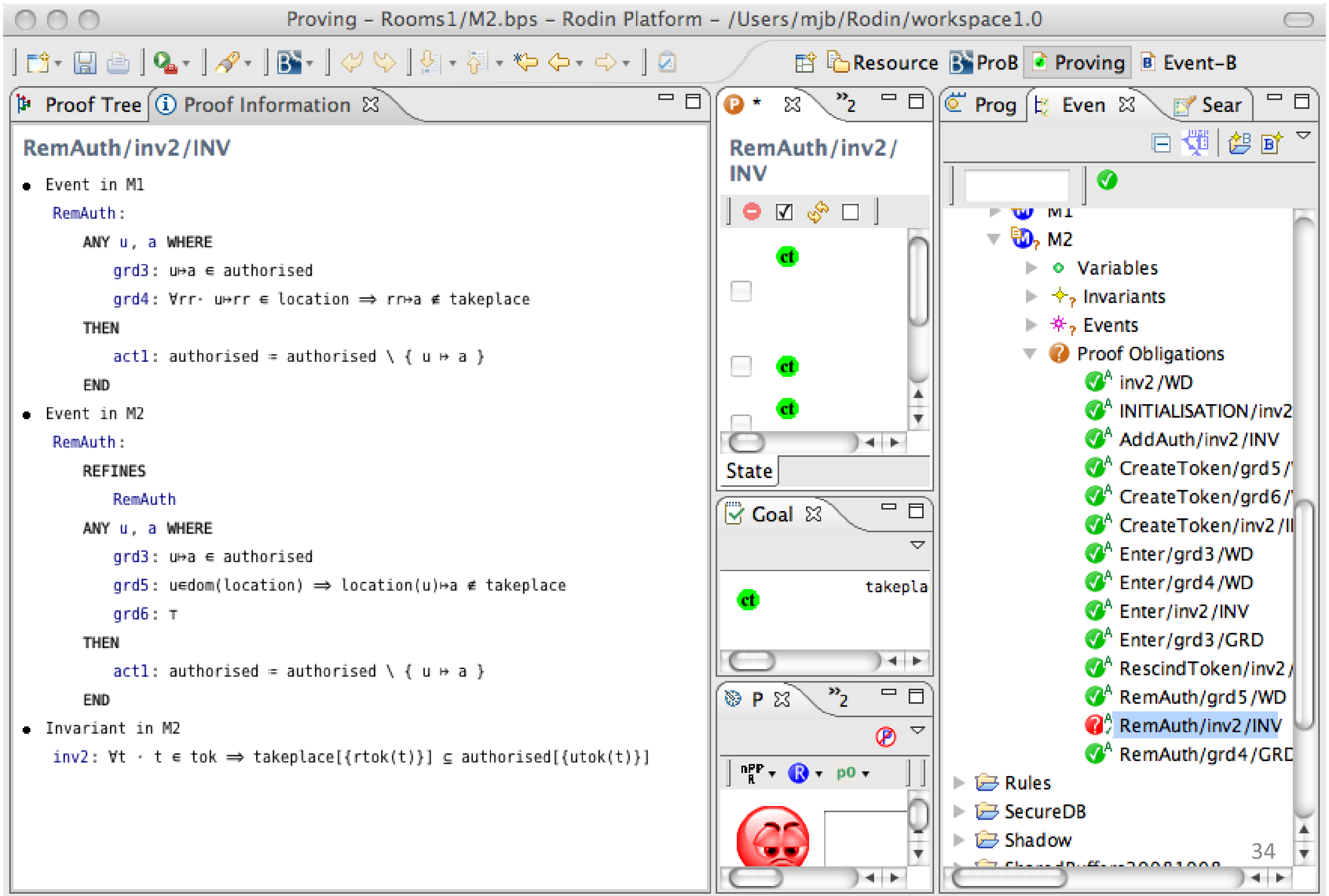
$\Rightarrow$

$takeplace\,[\,room(t)\,] \subseteq authorised[\,holder(t)\,]$

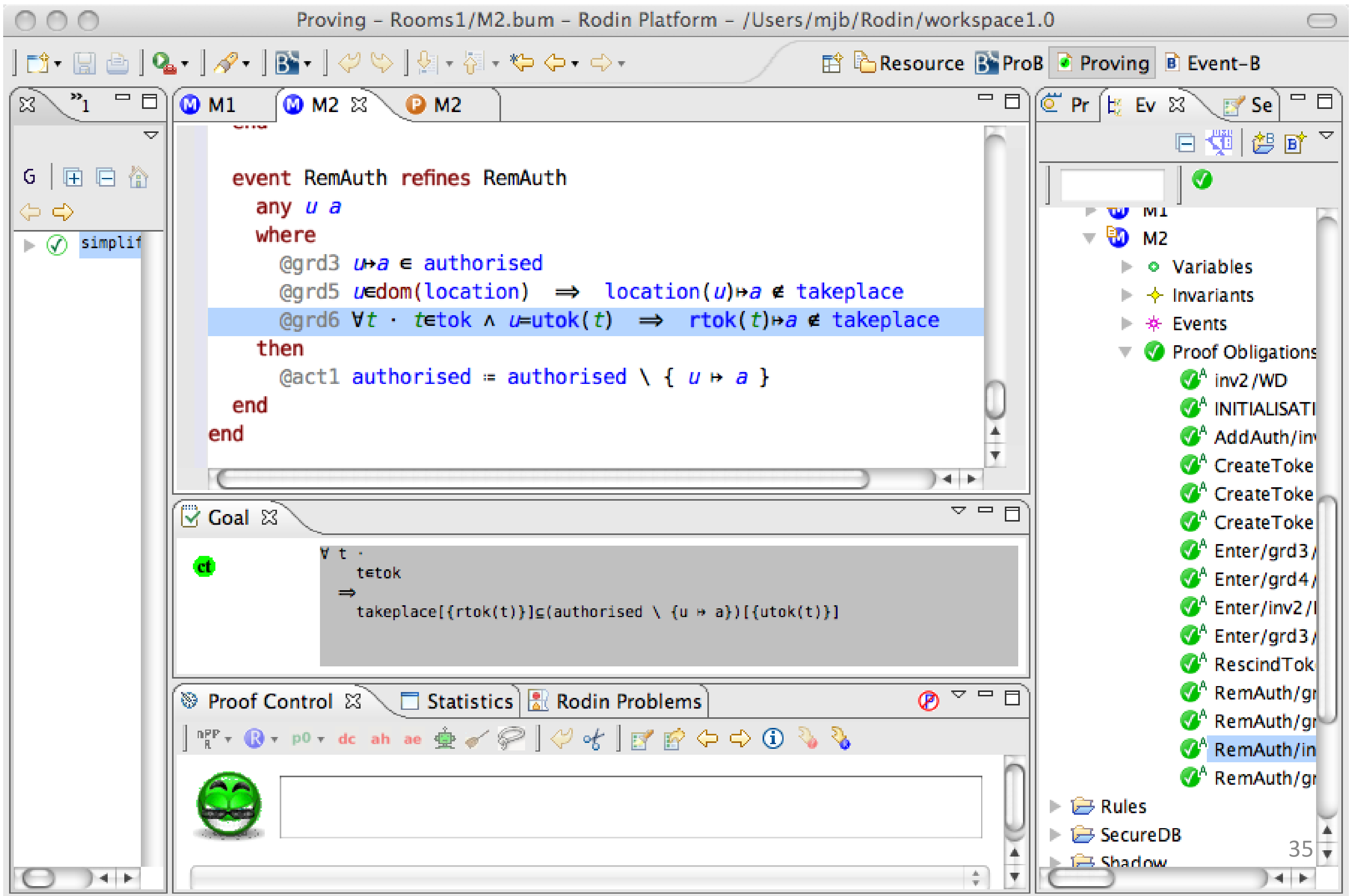
Invariant enables PO discharge



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?

  inv1: $u \in dom(location) \land location(u) = r$
  $\Rightarrow$
  $takeplace[r] \subseteq authorised[u]$

- *How* does it work?

  grd3b: $t \in valid \land r = room(t) \land u = holder(t)$

- *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 of Event-B models with Rodin tool
- Structured event decomposition
- Model decomposition
- Towards a method for decomposition

END

UNIVERSITY OF Southampton

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

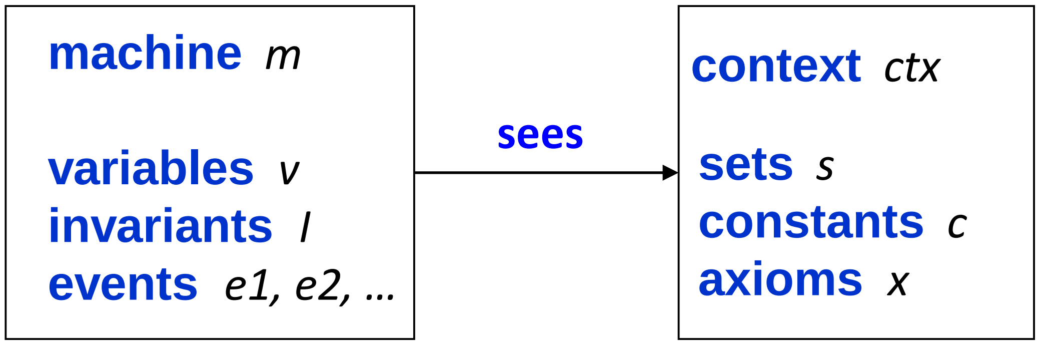
- Event-B provides a notion of consistency of a refinement:
  - Use proof to verify the consistency of a refinement step
  - Failing proof can help us identify inconsistencies

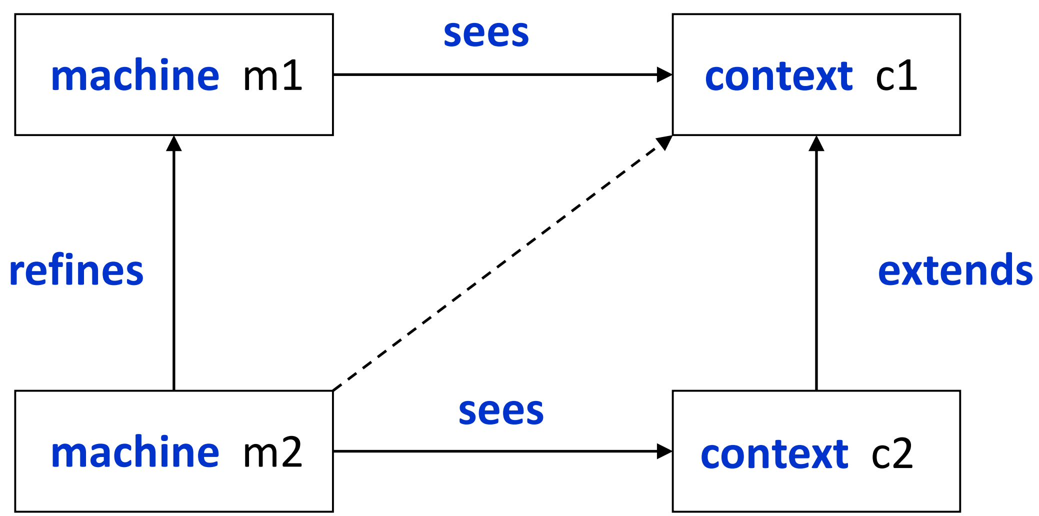
Validation and verification

- 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





Event structure

```
E =                         \\  event name
    any
        x1, x2, ...         \\  event parameters
    where
        G1                  \\   event guards
(predicates)
        G2
        ...
    then
        v1 := exp1          \\ event actions
        v2 := exp2
        ...
    end
```

Role of Event Parameters

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

  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:

  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 = **when** P(v) **then** v := exp(v) **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:

  Ev = **any** x **where** P(x,v) **then** v := exp(x,v) **end**

INV: $I(v),\ P(x,v) \ \vdash\ I(\,E(x,v)\,)$

Example PO from Rodin

Enter/inv3/INV

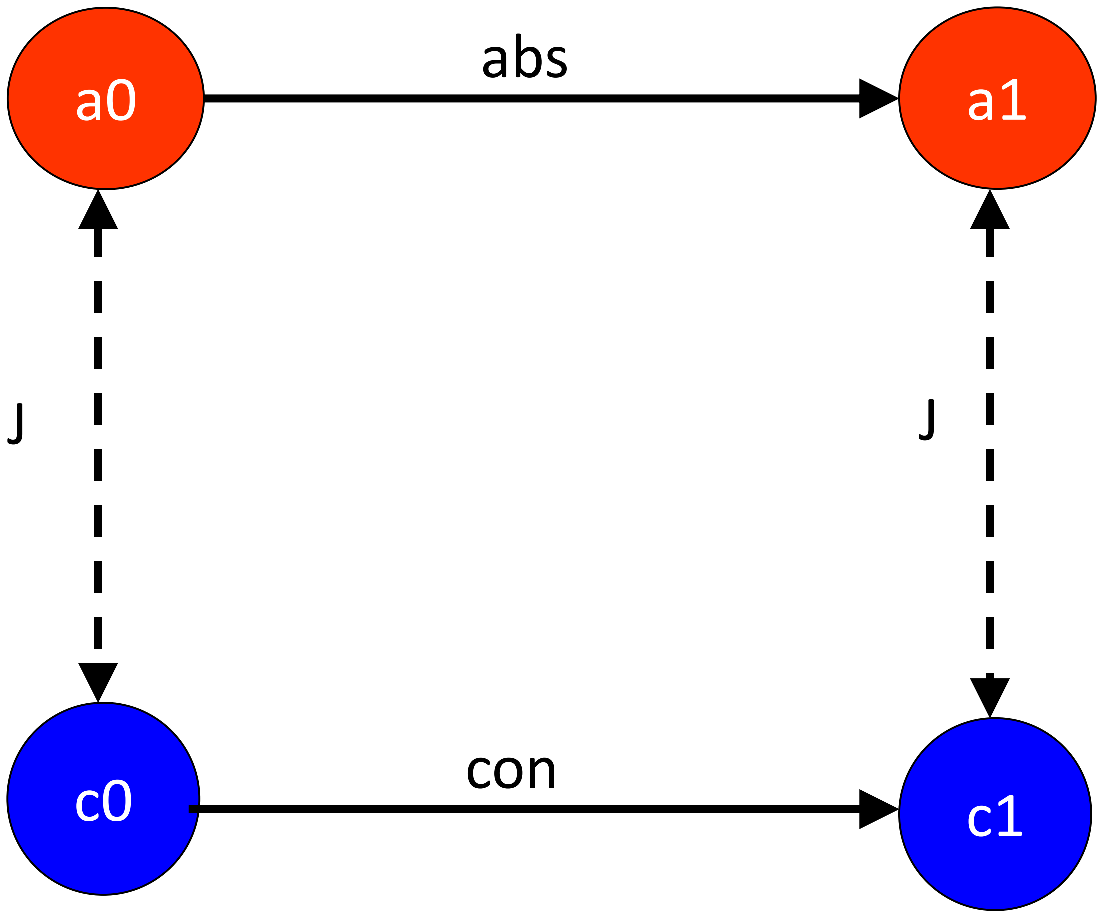
```
∀ u, r ·
    u∈dom(location) ∧
    location(u)=r
  ⇒
    takeplace[{r}]⊆authorised[{u}]
u∈USER \ dom(location)
takeplace[{r}]⊆authorised[{u}]
(location<+{u ↦ r})(u0)=r0
u0∈dom(location<+{u ↦ r})
takeplace=ROOM × ACTIVITY
location∈USER ⇸ ROOM
```

Selected Hypotheses

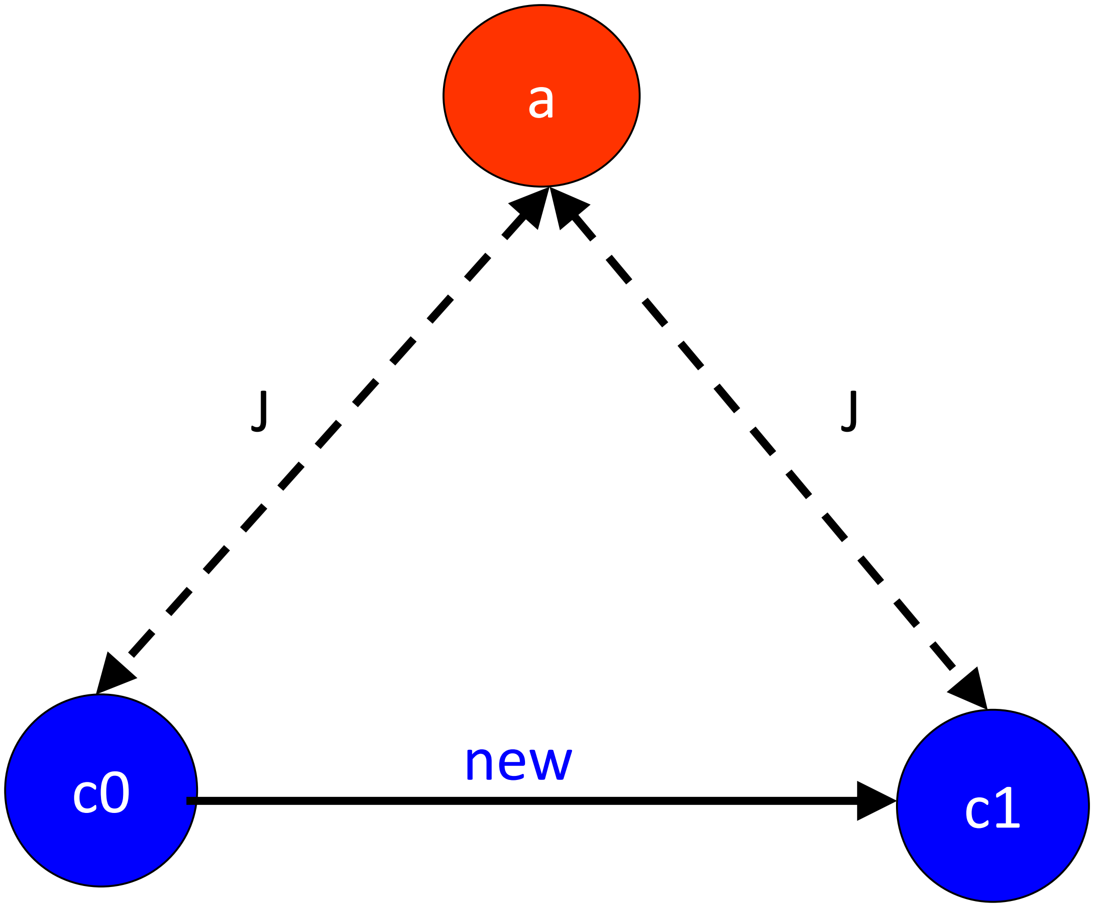
Goal

```
takeplace[{(location<+{u ↦ r})(u0)}]⊆authorised[{u0}]
```

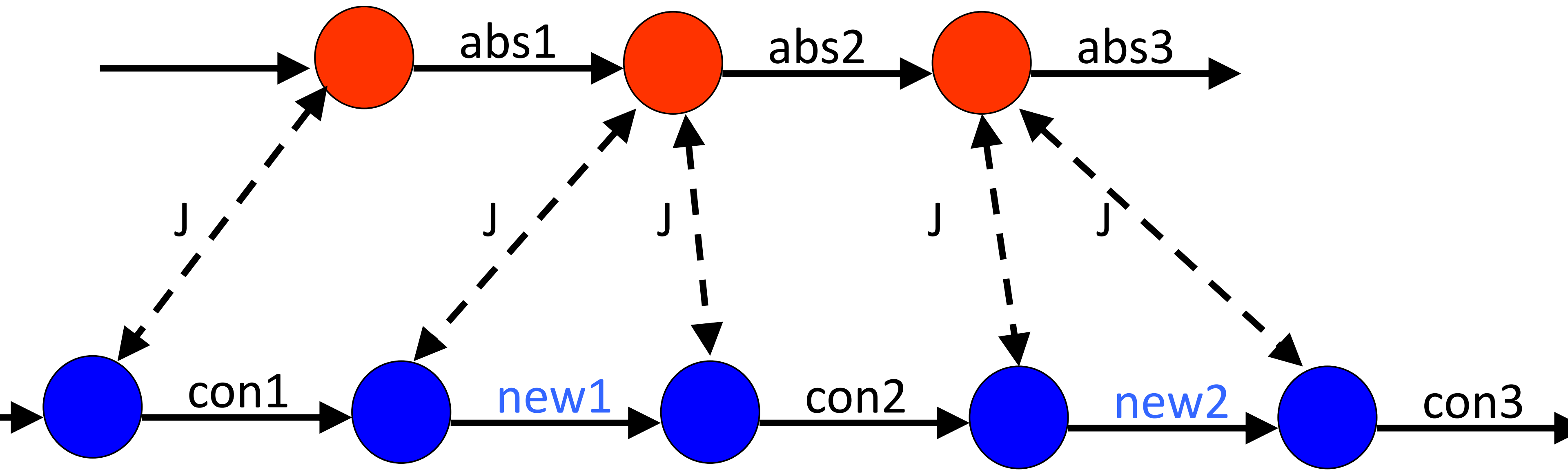
Simulation: maintaining a gluing relation



New concrete events refine *skip* (stuttering step)



Refining traces



Proof method for refinement (deterministic case)

- Suppose event *con* refines event *abs*:

  abs = when P(a) then a := E(a) end

  con = when Q(c) then c := F(c) end

- Verification of this refinement gives rise to two Proof Obligations:

GRD:	I(a), J(a,c), Q(a) ⊢ P(a)
SIM:	I(a), J(a,c), Q(a) ⊢ J(E(a), F(c))

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

Some references

Comprehensive definition of proof obligations (plus much more):

Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press 2010

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

State trace refinement:

Ralph-Johan Back and Joakim von Wright. *Trace Refinement of Action Systems.* CONCUR '94

Event trace refinement:

Michael Butler. *Stepwise Refinement of Communicating Systems*

Science of Computer Programming, 27 (2), 1996

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.
  International Journal on Software Tools for Technology Transfer (STTT), 12 (6), 2010.

- Leuschel and Butler
  ProB: An Automated Analysis Toolset for the B Method.
  International Journal on Software Tools for Technology Transfer, 10, (2), 185-203, 2008.

Rodin Demo

Access Control Example

Rodin Plug-ins www.event-b.org

- 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
- …

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 $\subseteq$ FILE $\wedge$
  dsk $\in$ file $\rightarrow$ CONT

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

events

CreateFile $\triangleq$ …

WriteFile $\triangleq$ // set contents of *f* to be *c*
  any f, c **where**
    f $\in$ file
    c $\in$ CONT
  then
    dsk(f) := c
  end

ReadFile $\triangleq$ // return contents of *f*
  any f, c! **where**
    f $\in$ file
    c! = dsk(f)
  end

Sample event traces of file store

⟨ CreateFile.f1,
WriteFile.f1.c1,
ReadFile.f1.c1, … ⟩

⟨ CreateFile.f1,
CreateFile.f2,
WriteFile.f2.c4,
WriteFile.f1.c6, … ⟩

An (infinitely) many more traces.

Refinement of file store

- Structure of file content: CONT = PAGE ⇸ DATA
- Instead of writing entire contents in one atomic step, each page is written separately:

machine filestore2
refines filestore

variables file, dsk, writing, wbuf, tdsk

invariant

$$\begin{aligned} & writing \subseteq file \\ & wbuf \in writing \rightarrow CONT \\ & tdsk \in writing \rightarrow CONT \quad \text{// temporary disk} \end{aligned}$$

Refining the *WriteFile* event

- **Abstract**: WriteFile

- **Refinement**:

  StartWriteFile

  WritePage

  EndWriteFile (refines WriteFile)

Events of refinement

StartWriteFile $\triangleq$

any f, c **where**

f $\in$ (file \ writing)

c $\in$ CONT

then

writing := writing $\cup$ {f}

wbuf(f) := c

tdsk(f) := {}

end

WritePage $\triangleq$

any f, p, d **where**

f $\in$ writing

p $\mapsto$ d $\in$ wbuf(f)

p $\mapsto$ d $\notin$ tdsk(f)

then

tdsk(f) := tdsk(f) $\cup$ { p $\mapsto$ d }

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

⟨ CreateFile.f1,
CreateFile.f2,
WriteFile.f2.c2,
WriteFile.f1.c1
… ⟩

⟨ CreateFile.f1,
StartWriteFile.f1.c1,
CreateFile.f2,
WritePage.f1.p2.c1(p2),
StartWriteFile.f2.c2,
WritePage.f1.p1.c1(p1),
WritePage.f2.p1.c2(p1),
WritePage.f2.p2.c2(p2),
EndWriteFile.f2.c2,
WritePage.f1.p3.c1(p2),
EndWriteFile.f1.c1
… ⟩

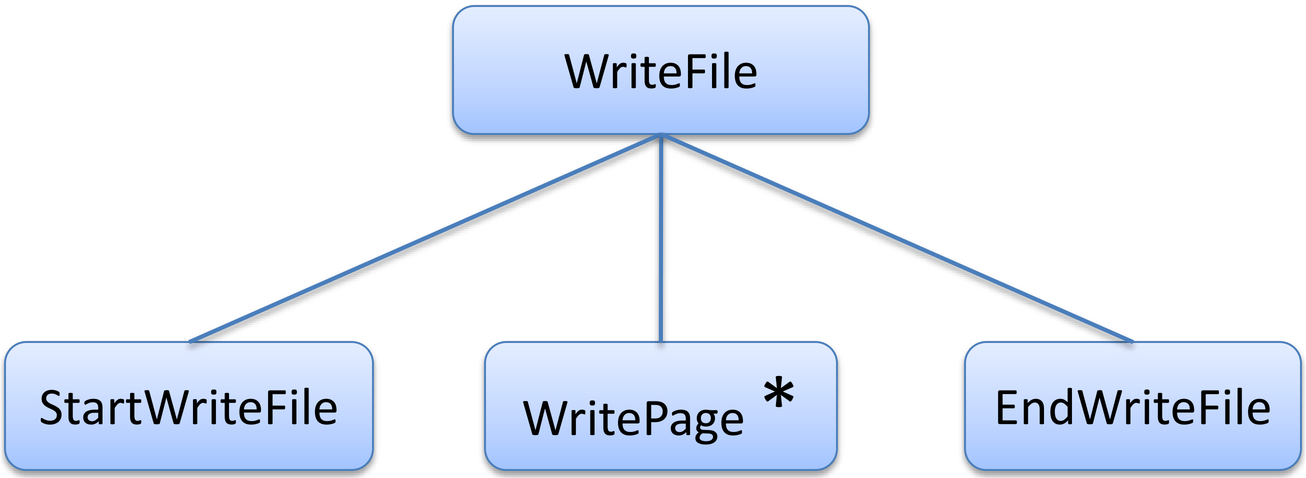
Breaking atomicity

- 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

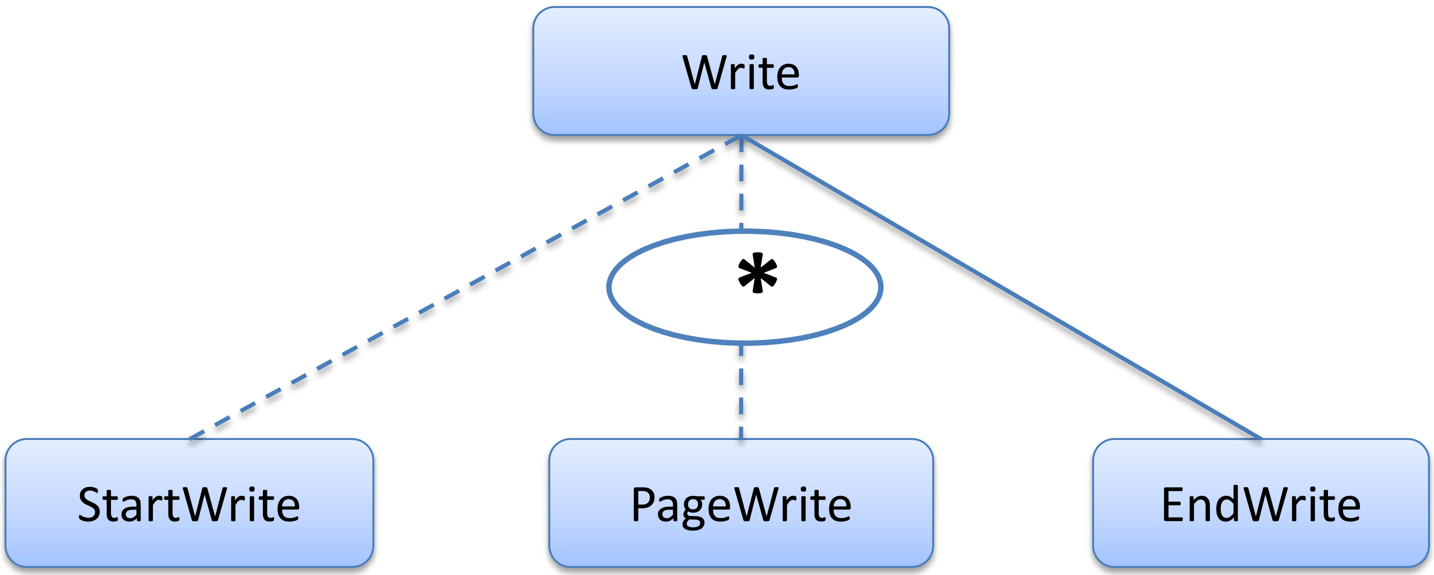
WriteFile sequencing in JSD



Sequencing is from left to right

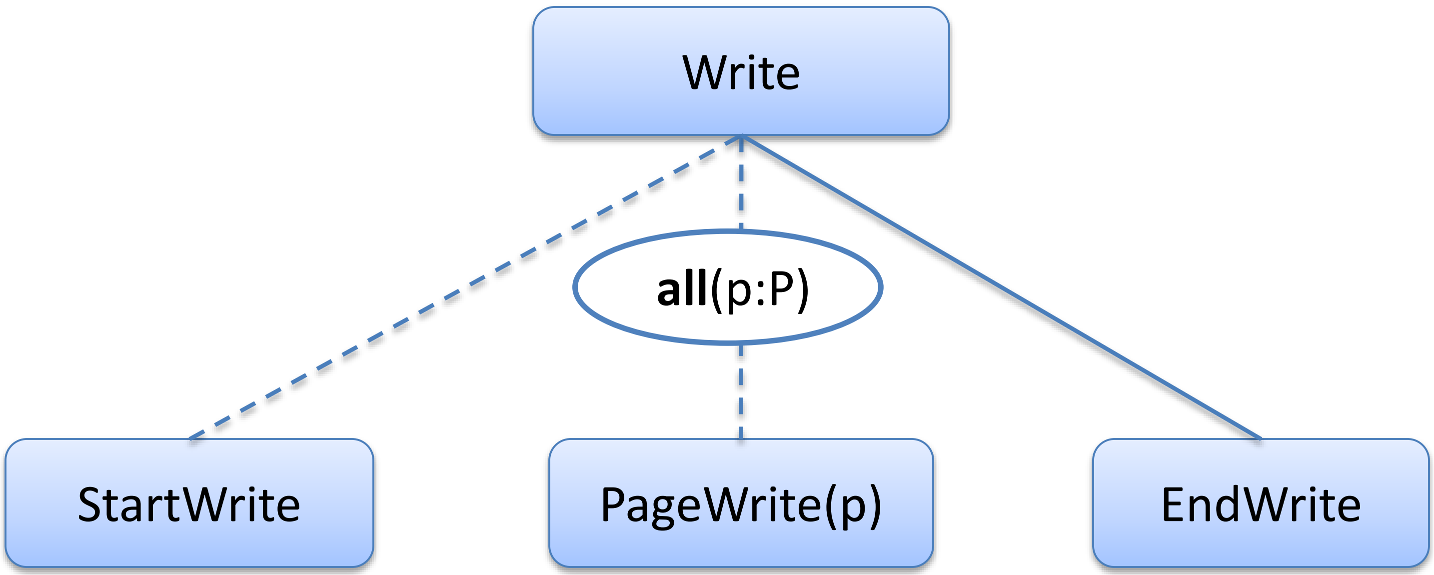
* signifies iteration

Adapting the diagrams



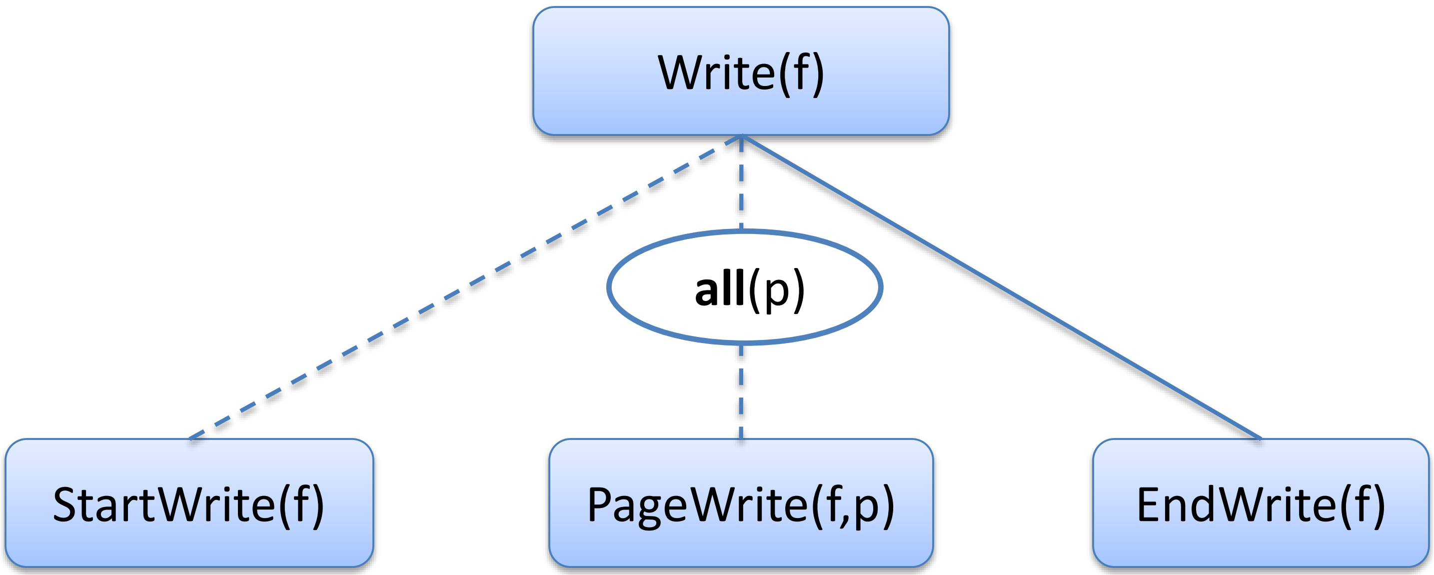
- Attach the iterator to an arc rather than a node to clarify atomicity
- Events are represented by leaves of the tree
- Solid line indicates *EndWrite* refines *Write*
- Dashed line indicates new events refining *skip*

Nondeterministic forall



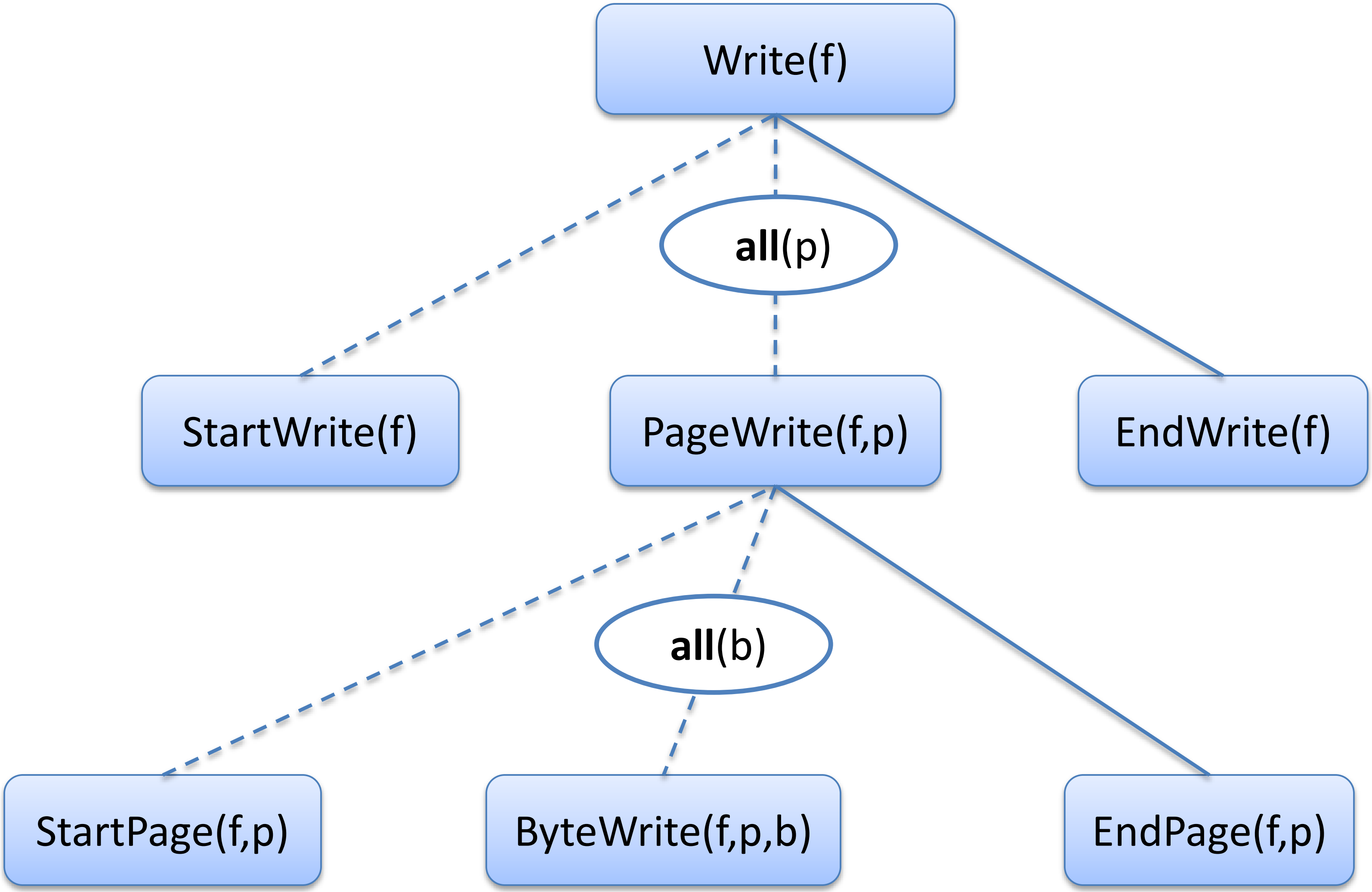
- pages may be written after *StartWrite* has occurred
- the writing is complete (*EndWrite*) once **all** pages have been written
- order of *PageWrite* events is nondeterministic
- this abstract program structure represents atomicity refinement explicitly

Interleaving of multiple instances

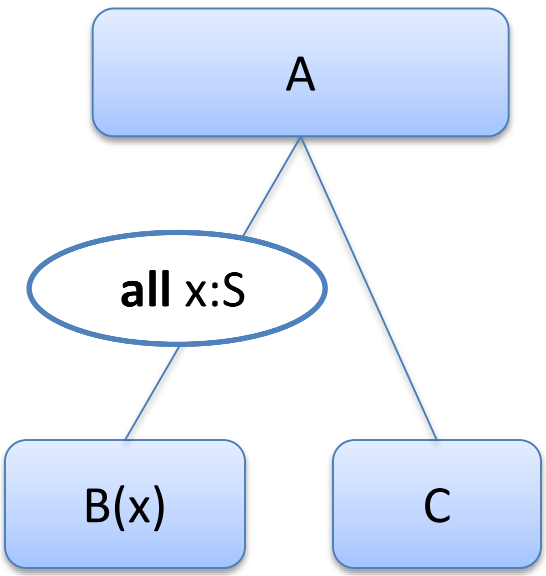


- Multiple write "processes" for different files may interleave
  - (sub-)events of *Write(f1)* may interleave with (sub-)events of *Write(f2)*
  - (sub-)events of *Write(f1)* may interleave with (sub-)events of *Read(f1)*
- *interleaving can be reduced with explicit guards (e.g., write lock)*

Hierarchical refinement



Event-B encoding



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

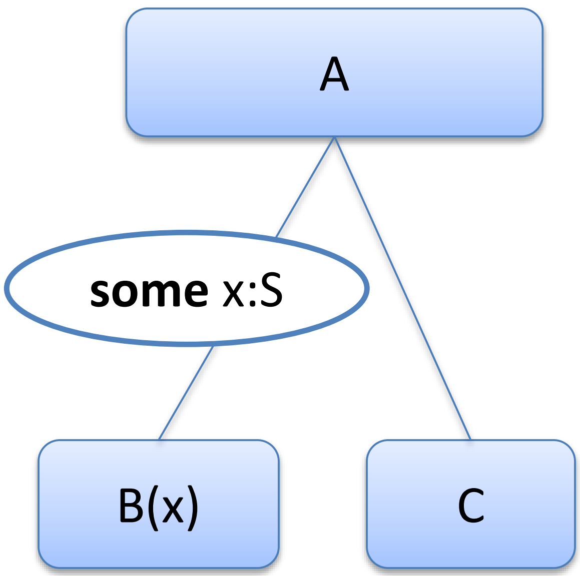
Events:

B ≜ $x \in S \setminus B$ ❼ $B := B \cup \{x\}$

C ≜ $B = S \land \neg C$

❼ $C := \text{TRUE}$

SOME program structure



Events:

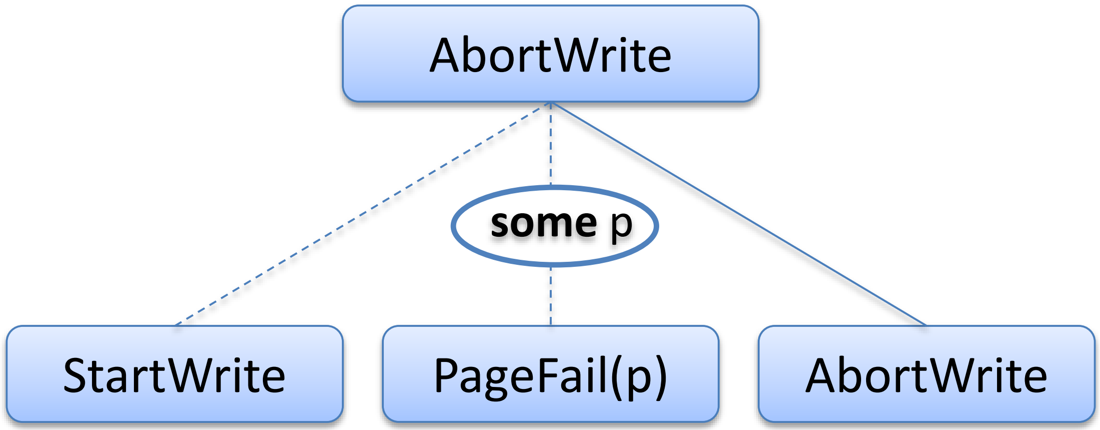
$B \triangleq x \in S \setminus B$ ❼ $B := B \cup \{x\}$

$C \triangleq B \neq \{\} \land \neg C$ ❼ $C := TRUE$

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

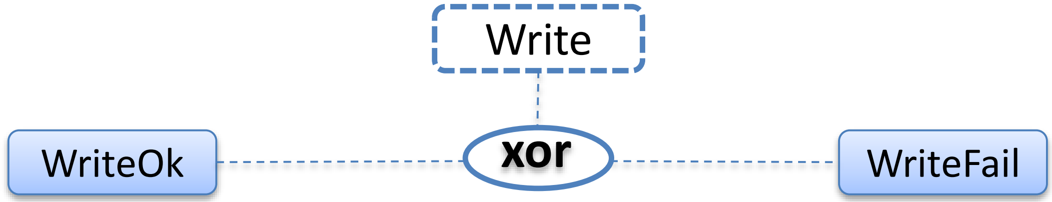
B(x’) may occur after C for other x’

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*

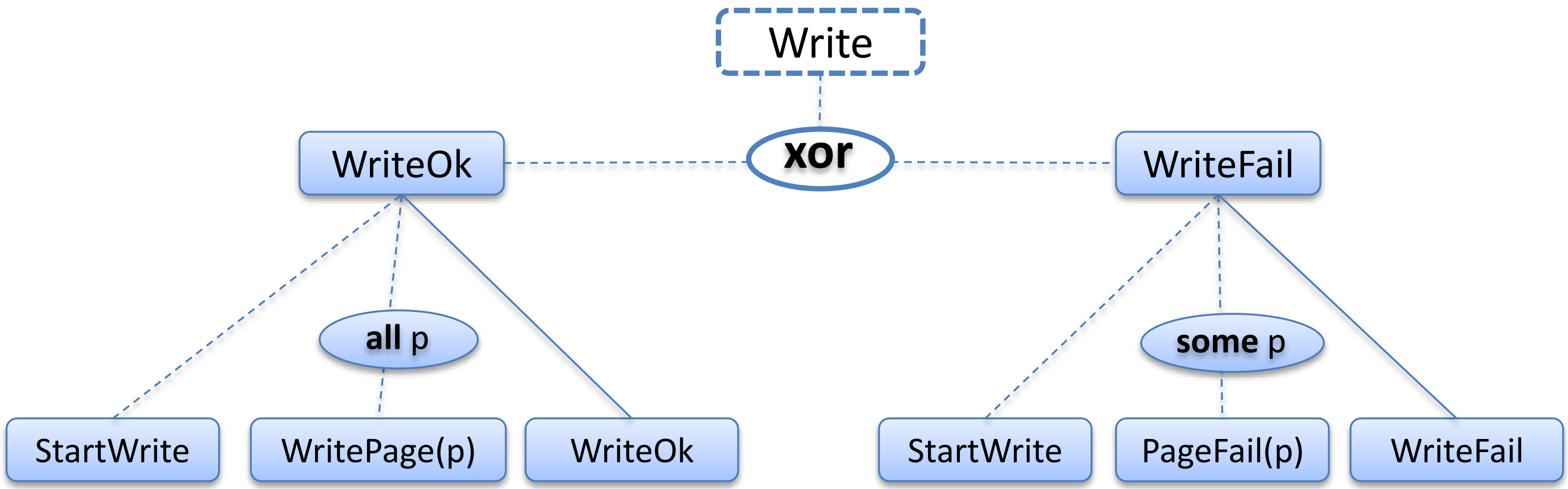
Separation of concerns



WriteOk ≙
begin
disk := file
end

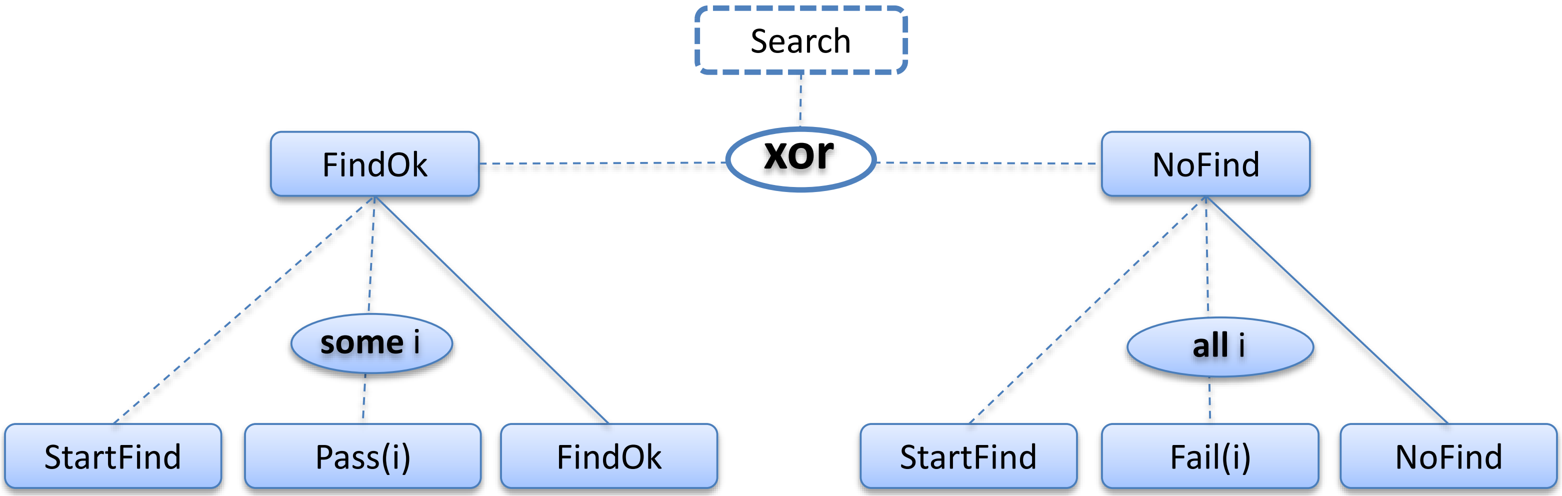
WriteFail ≙
begin
skip
end

Layered refinement



- M0: two events - *WriteOk* and *WriteFail*
- M1: refine atomicity of *WriteOk*
- M2: refine atomicity of *WriteFail*

Search

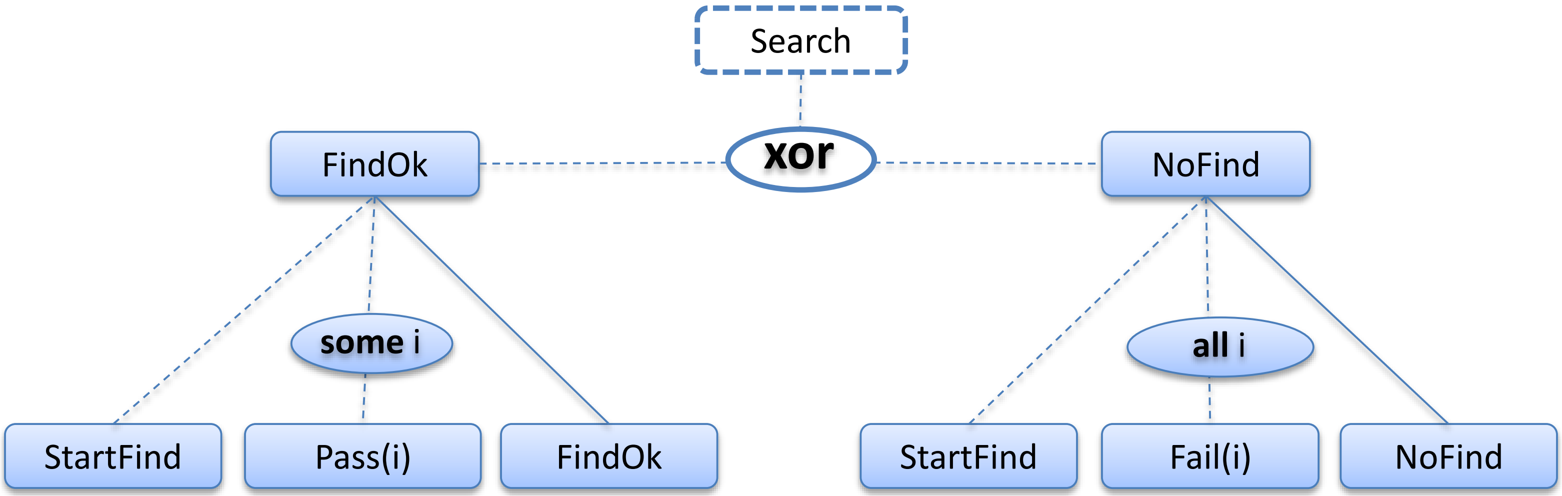


- *FindOk*: find a point in *S* satisfying property *P* $x \in S \cap P$

or

- *NoFind*: determine that no point in *S* satisfies $S \cap P = \{\}$

Invariants for verification

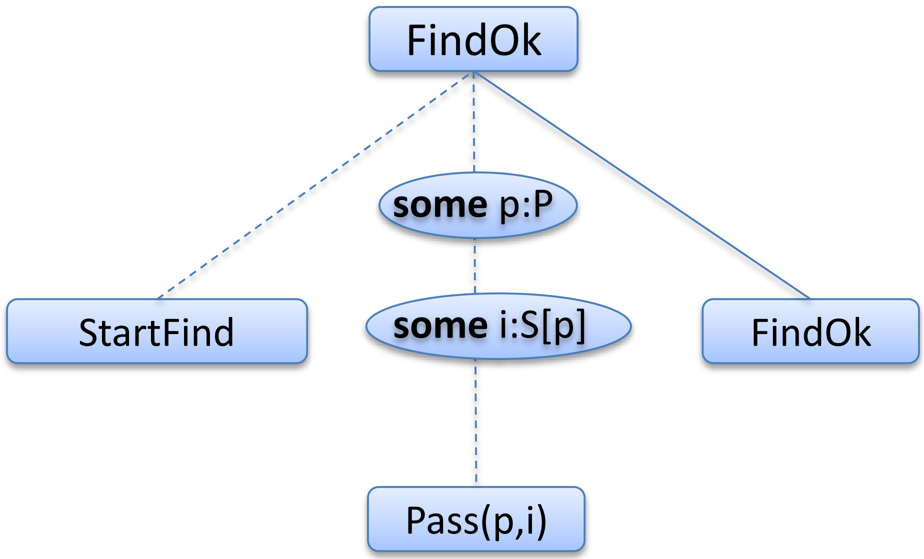


- Pass $\subseteq$ S $\cap$ P
- Fail $\subseteq$ S \ P

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

$$db \in object \rightarrow DATA$$

Commit = /* update a set of objects *os* */

any os, update

where

$os \subseteq object \land$

$update \in (os \rightarrow DATA) \rightarrow (os \rightarrow DATA)$

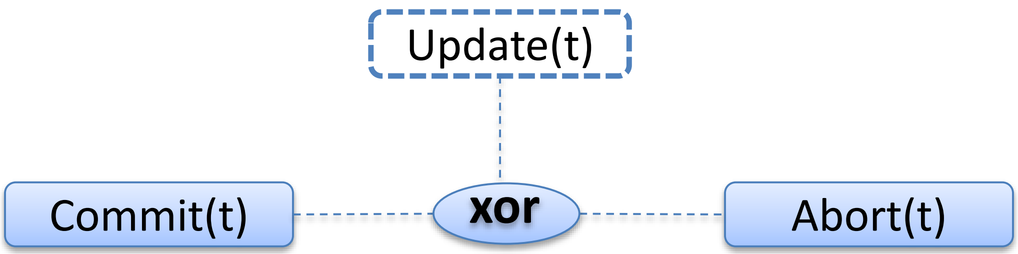
then

db := db **<+** update(os ◁ db)

end

Update Transaction

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



Refinement by replicated database

$$\text{ldb} \in \text{site} \to (\text{object} \to \text{DATA})$$

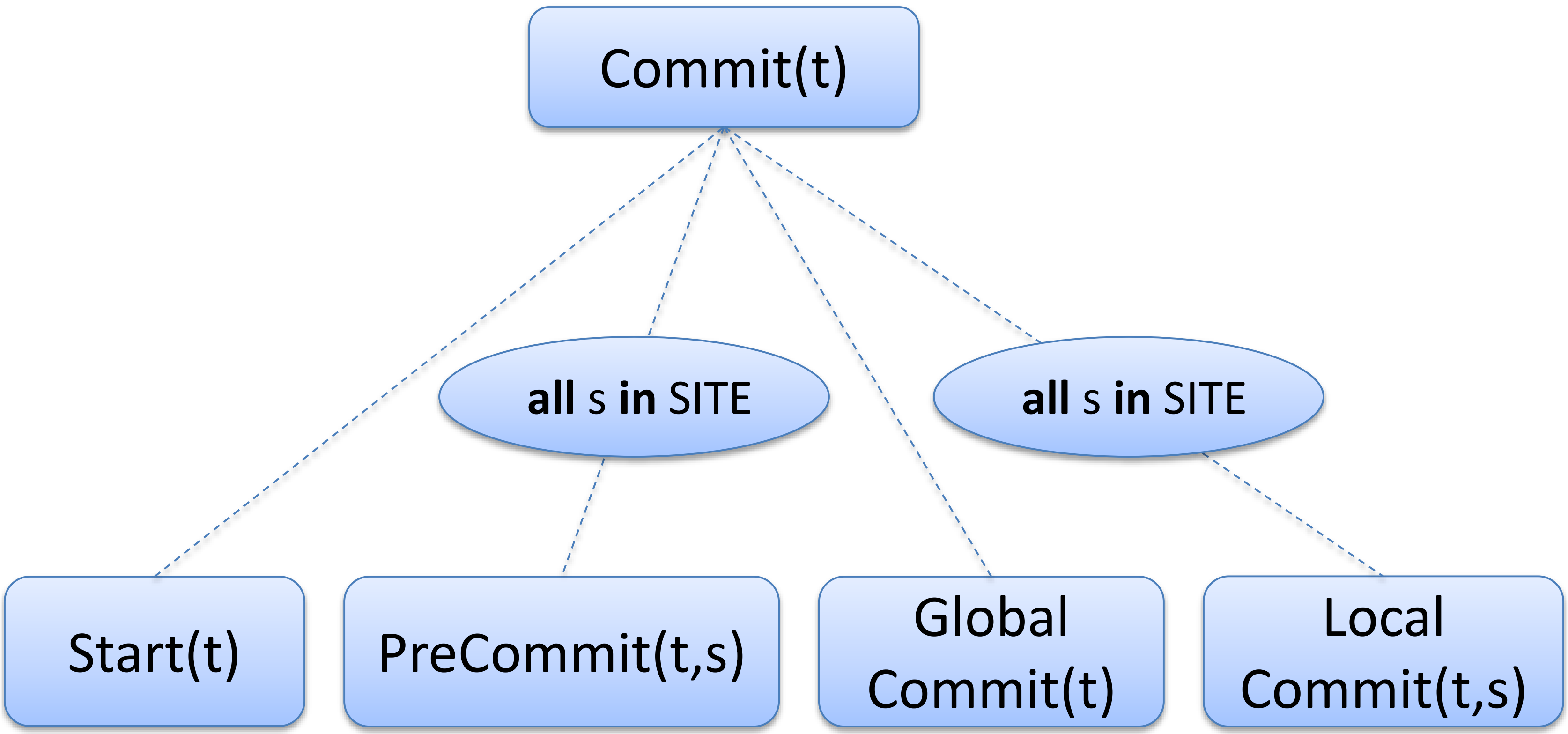
Update is by two phase commit:

PreCommit followed by Commit

Global commit if all sites *pre-commit*

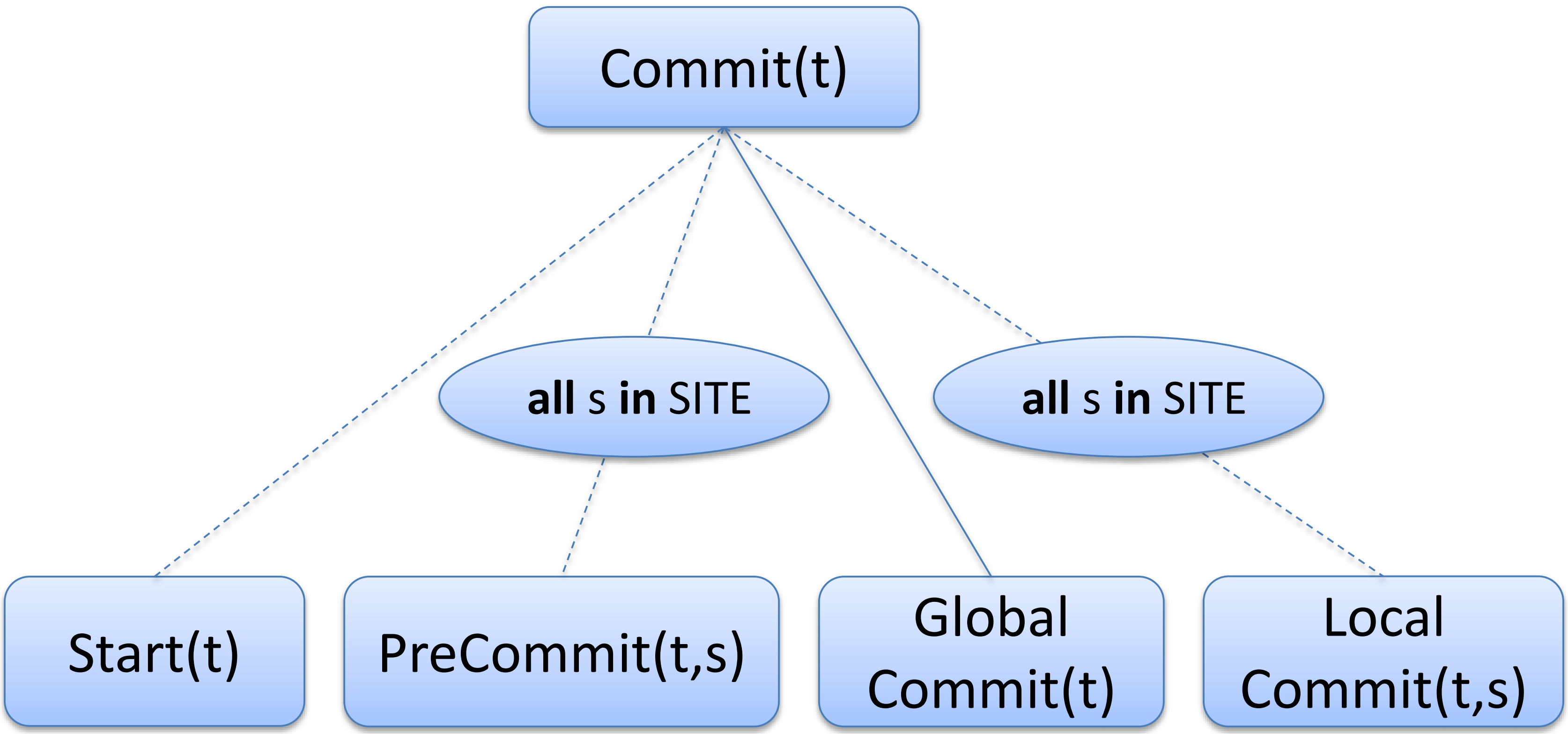
Global abort if at least one site aborts

Event refinement diagram for *Commit*



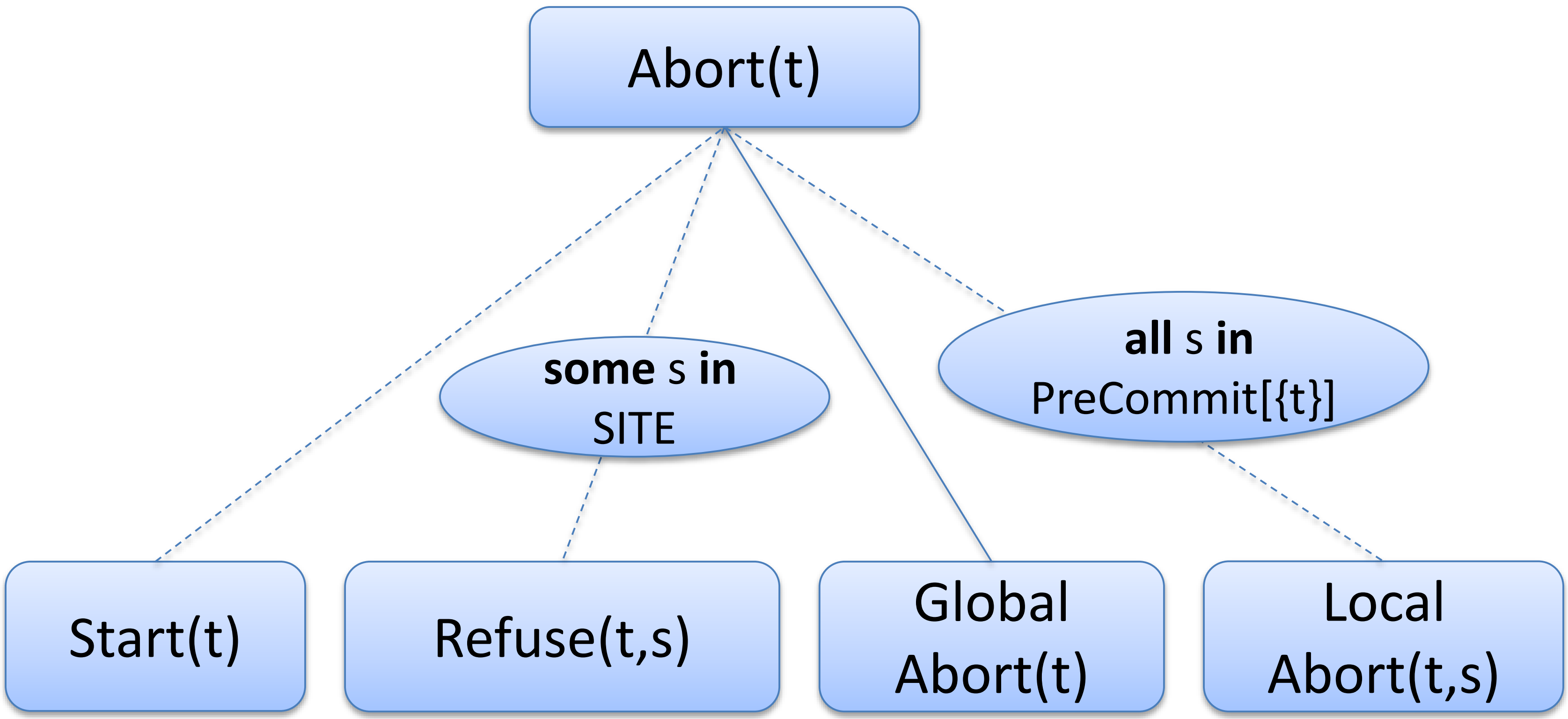
Which event refines the abstract *Commit*?

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

- *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*

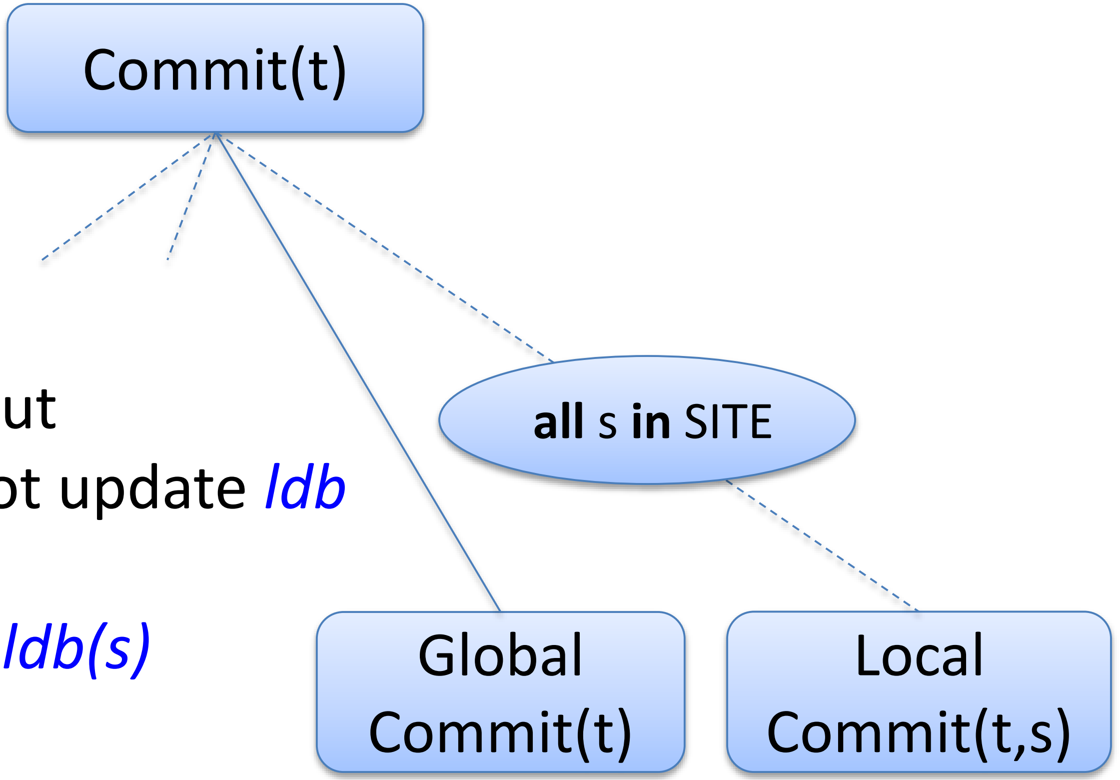
Read transactions

- 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:
  $\forall s, o \cdot o \notin dom(lock(s)) \Rightarrow (ldb(s))(o) = db(o)$

- But $(ldb(s))(o) = db(o)$ is broken by *GlobalCommit*

Global and local commit not synchronised



Commit updates *db*, but
GlobalCommit does not update *ldb*

LocalCommit updates *ldb(s)*

How are *db(o)* and *ldb(s)(o)* related in between *GlobalCommit* and *LocalCommit*?

Another gluing invariant

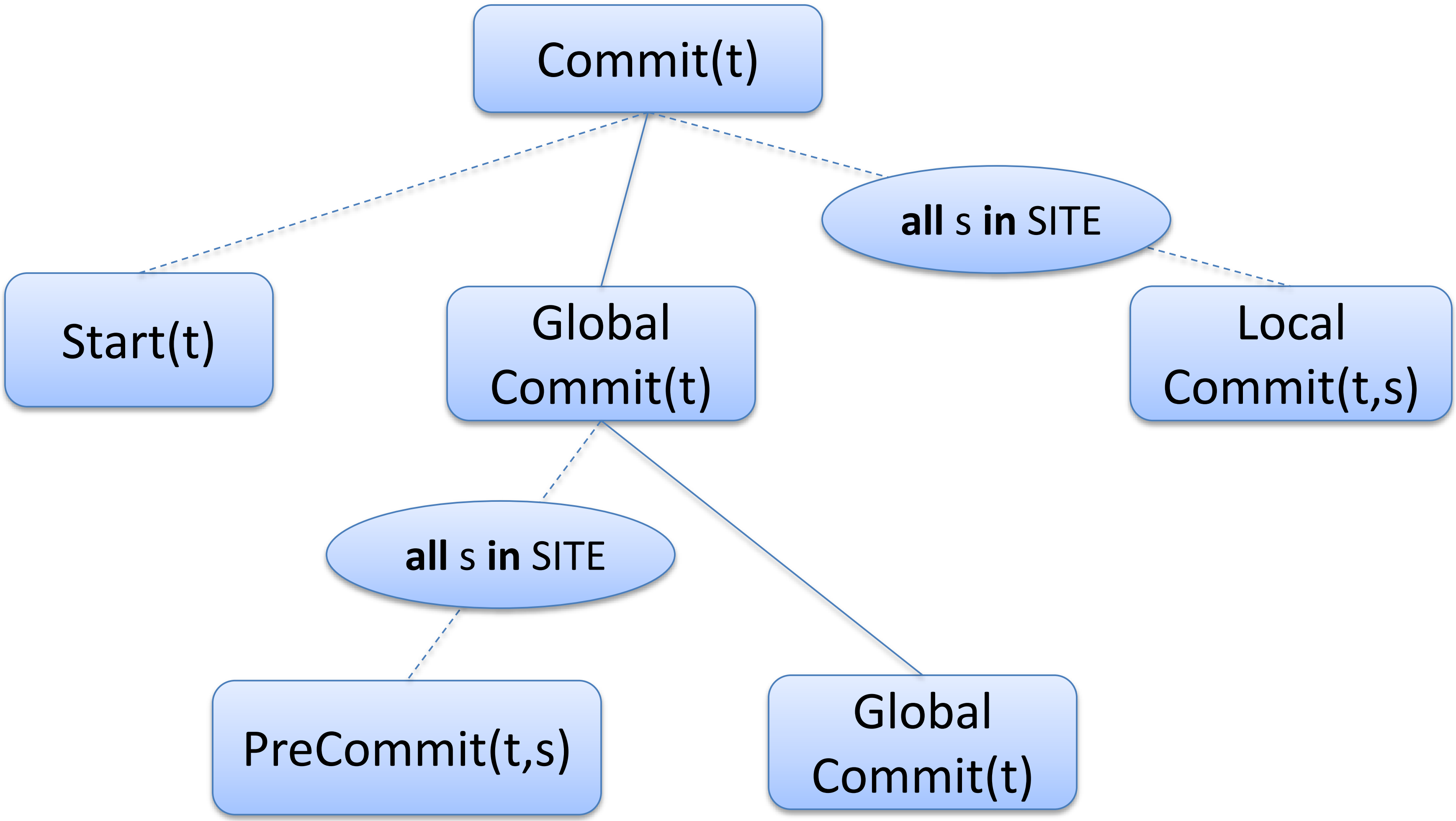
$$
\begin{array}{l}
t \in GlobalCommit \quad \wedge \\
t \mapsto s \;\notin\; LocalCommit \quad \wedge \\
os = tos[t] \quad \wedge \quad o \in os \quad \wedge \\
U = upd(t) \quad \wedge \quad L = os \lhd ldb(s)
\end{array}
$$

$$\Rightarrow$$

$$db(o) \;=\; (U(L))(o)$$

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

UNIVERSITY OF Southampton

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

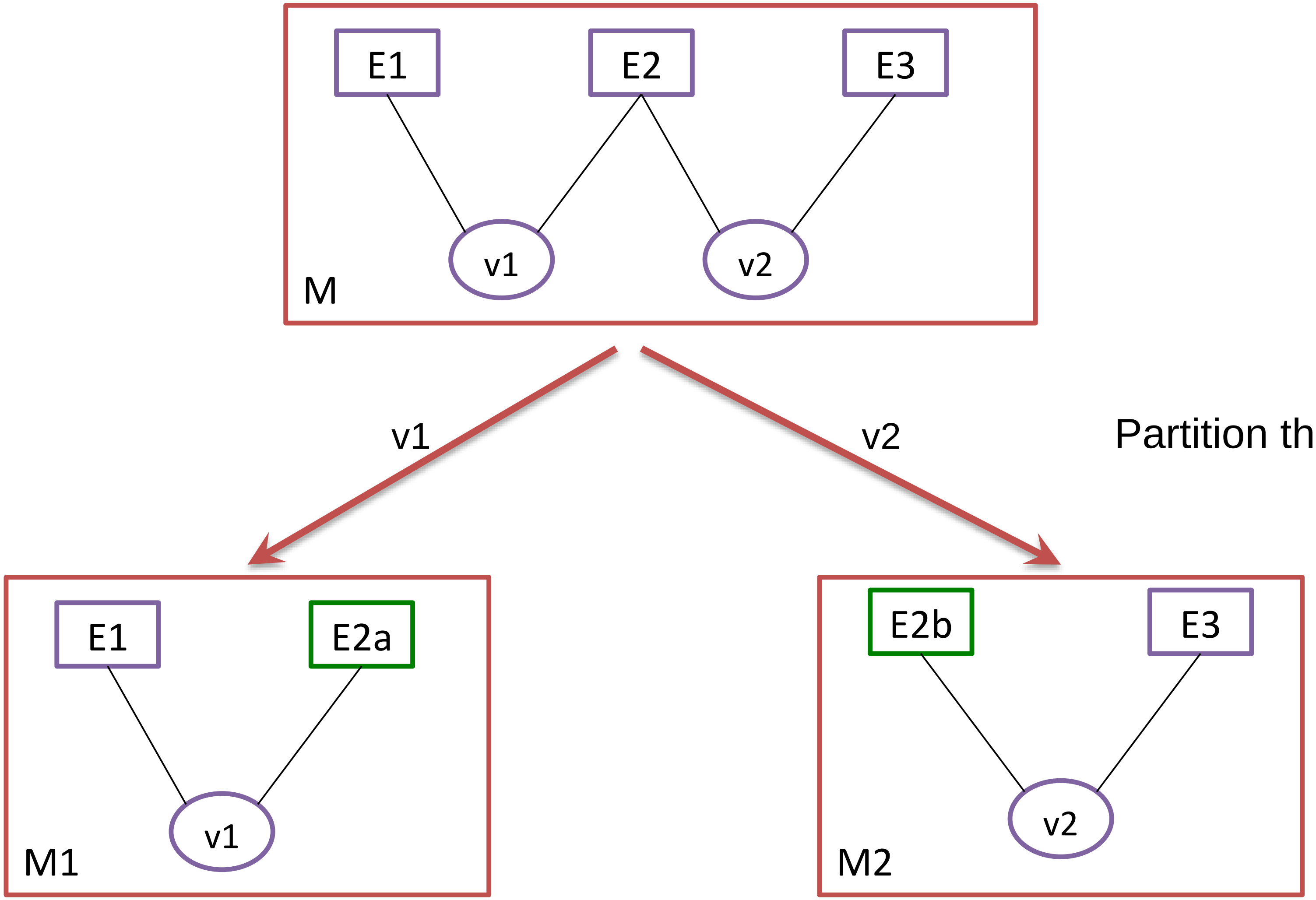
Model Decomposition styles

- 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

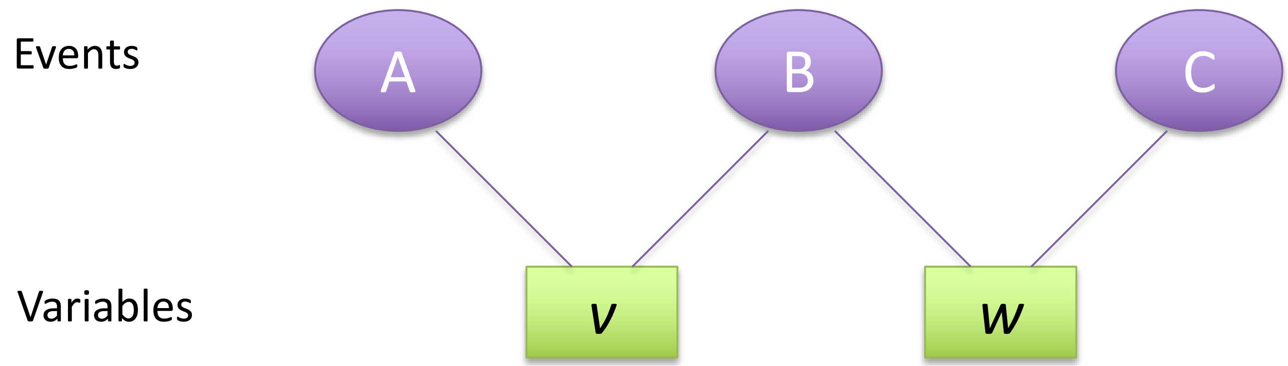
- Both styles supported by a decomposition plug-in

Shared Event Decomposition



Partition the variables

Shared Event Decomposition – by example

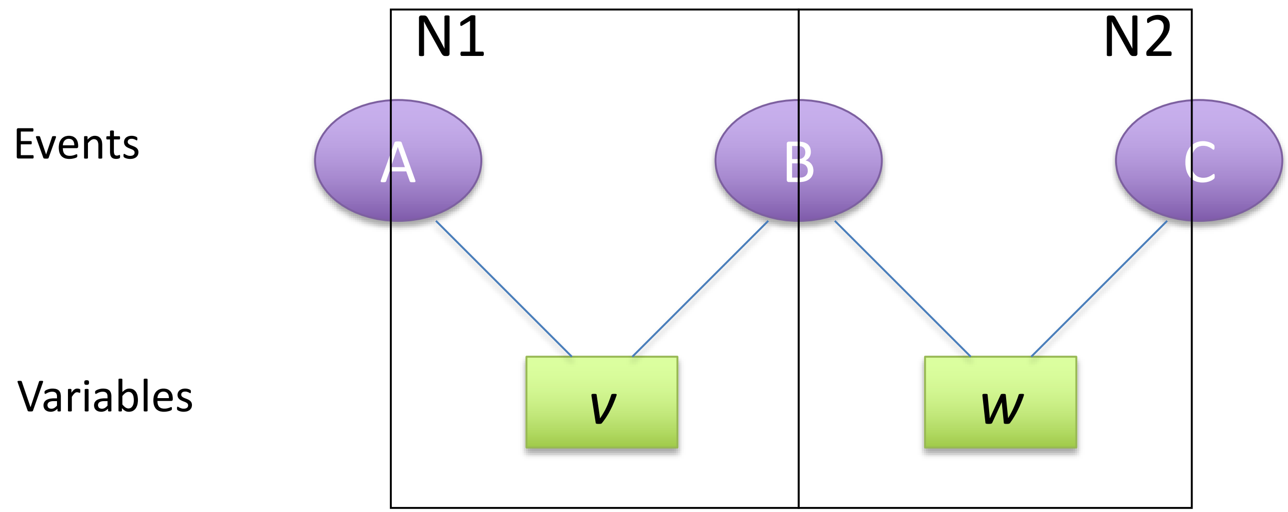


A ≜ v := v+1

B ≜ **when** v>0 ∧ w<M **then** v := v-1 || w := w+1 **end**

C ≜ **when** w>0 **then** w := w-1 **end**

Decompose by partitioning variables



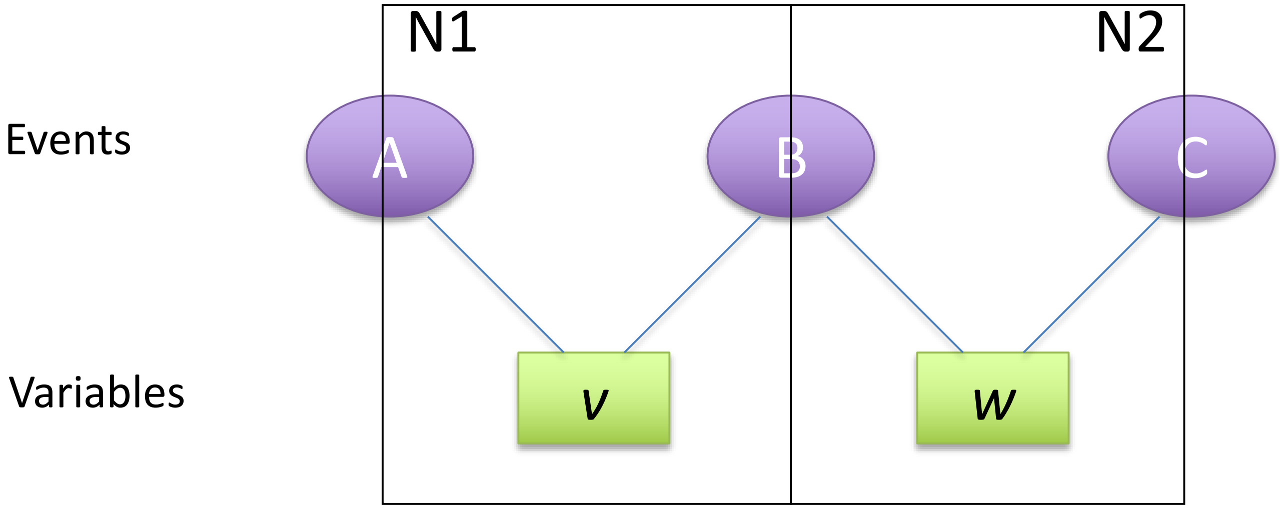
A **≜** v := v+1

B **≜** **when** v>0 ∧ w<M **then** v := v-1 || w := w+1 **end**

C **≜** **when** w>0 **then** w := w-1 **end**

B event needs to be split into *v*-part and *w*-part

Parallel Event Split



B ≜ **when** v>0 ∧ w<M **then** v := v-1 || w := w+1 **end**

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

B1 ≜ **when** v>0 **then** v := v-1 **end**

B2 ≜ **when** w<M **then** w := w+1 **end**

Synchronised events with parameter passing

B ≜ **any** x **where** $0 < x \leq v$

then v := v-x || w := w+x **end**

B can be split into 2 events that have x in common:

B1 ≜ **any** x **where** $0 < x \leq v$ **then** v := v-x **end**

B2 ≜ **any** x **where** $x \in \mathbb{Z}$ **then** w := w+x **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 =
any p **where**
G1(x, p)
G2(y, p)
then
x := H1(x, p)
y := H2(y, p)
end

Ex =
any p **where**
G1(x, p)
then
x := H1(x, p)
end

Ey =
any p **where**
G2(y, p)
then
y := H2(y, p)
end

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
```

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

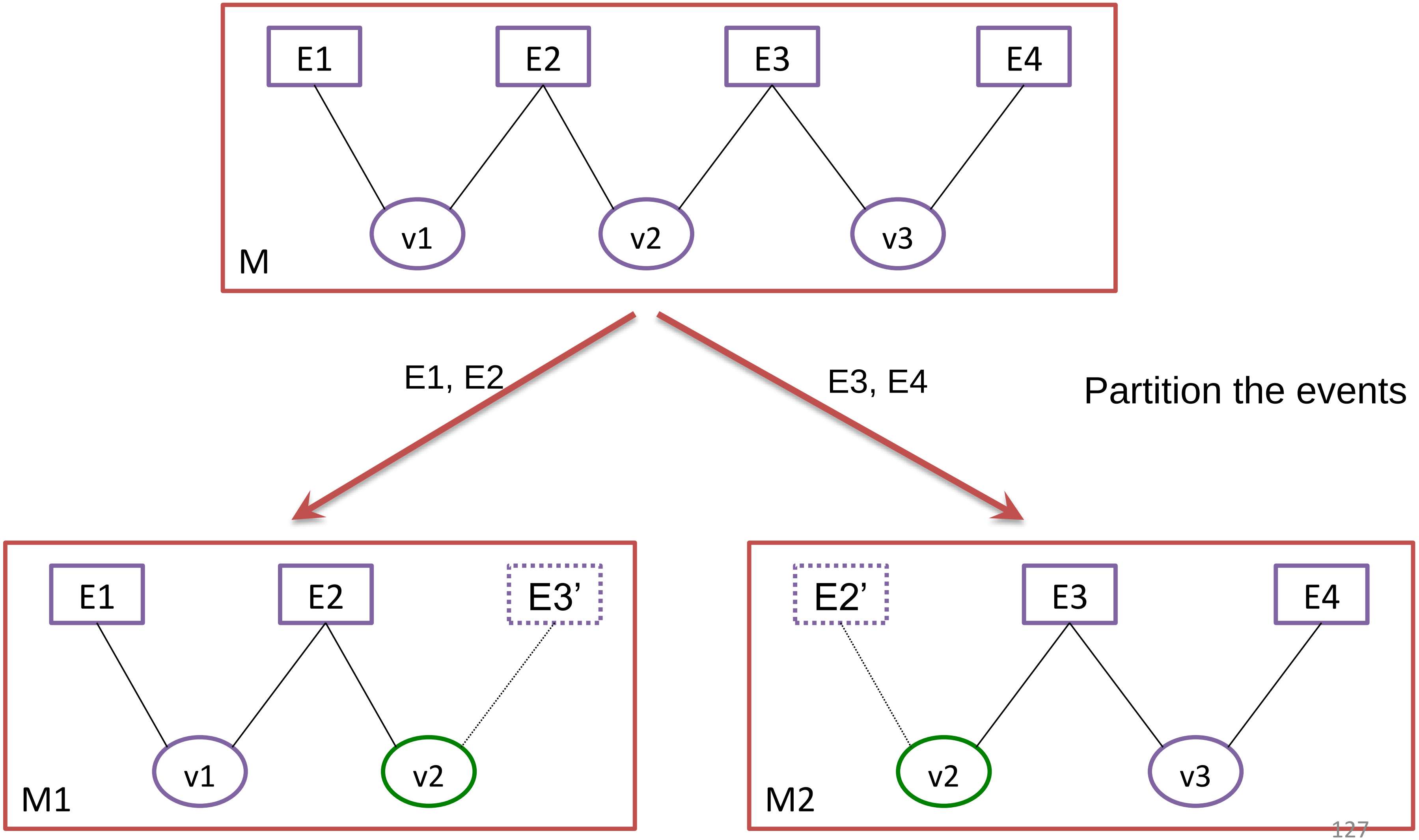
```
composed machine  M’
refines  M
Includes  L, P
events
    A  =  L.A
    B  =  L.B   ||   P .B
    C  =  P.C
end
```

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



Refinement after decomposition

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

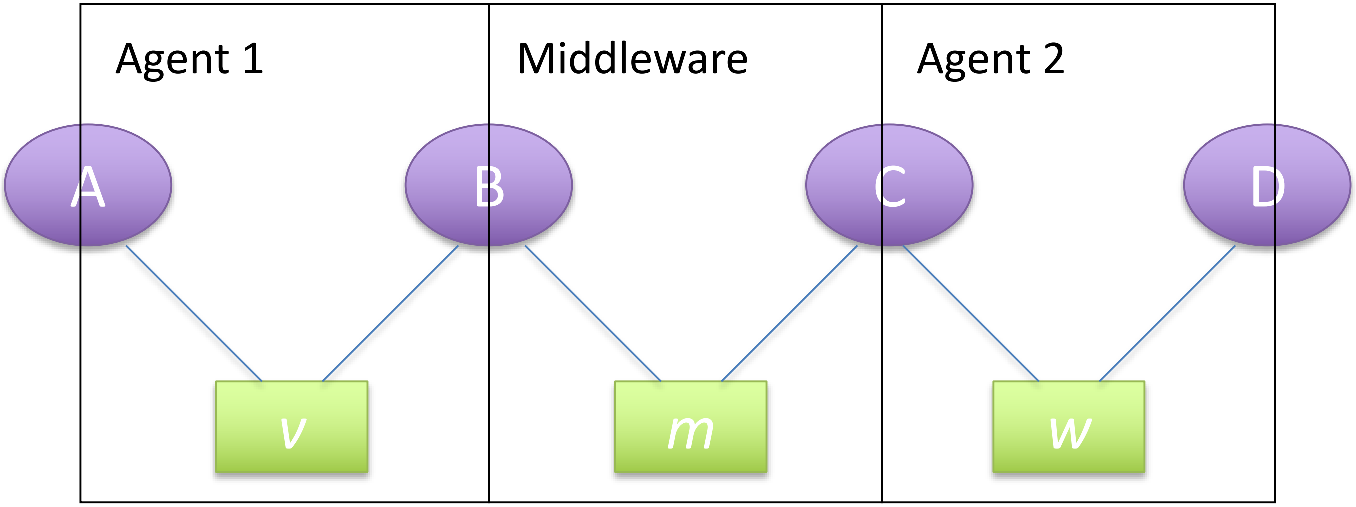
- Shared variable: can refine sub-model provided
  - External events are not refined (rely condition)
  - Private events in M1 that affect shared variables must refine some external event of M2, e.g., E3 refines E3'
  - Shared variables are not refined.
  - Invariants used in refinement are preserved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

- Butler, M. (2009) ***Decomposition Structures for Event-B***. In: Integrated Formal Methods iFM2009, LNCS 5423.
- Abrial, J.-R. and Hallerstede, S. (2007) ***Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B***. Fundam. Inf., 77(1-2).

- Silva, R., Pascal, C., Hoang, T. S. and Butler, M. (2011) ***Decomposition Tool for Event-B***. Software: Practice and Experience, 41 (2).
- Salehi Fathabadi, A., Rezazadeh, A. and Butler, M. (2011***)*** ***Applying Atomicity and Model Decomposition to a Space Craft System in Event-B***. In: Third NASA Formal Methods Symposium, 2011.
- Salehi Fathabadi, A., Butler, M. and Rezazadeh, A. (2012) ***A Systematic Approach to Atomicity Decomposition in Event-B***. In, *SEFM 2012.*

- http://www.ecs.soton.ac.uk/people/mjb/publications

END

UNIVERSITY OF Southampton

Towards a Method for Decomposition

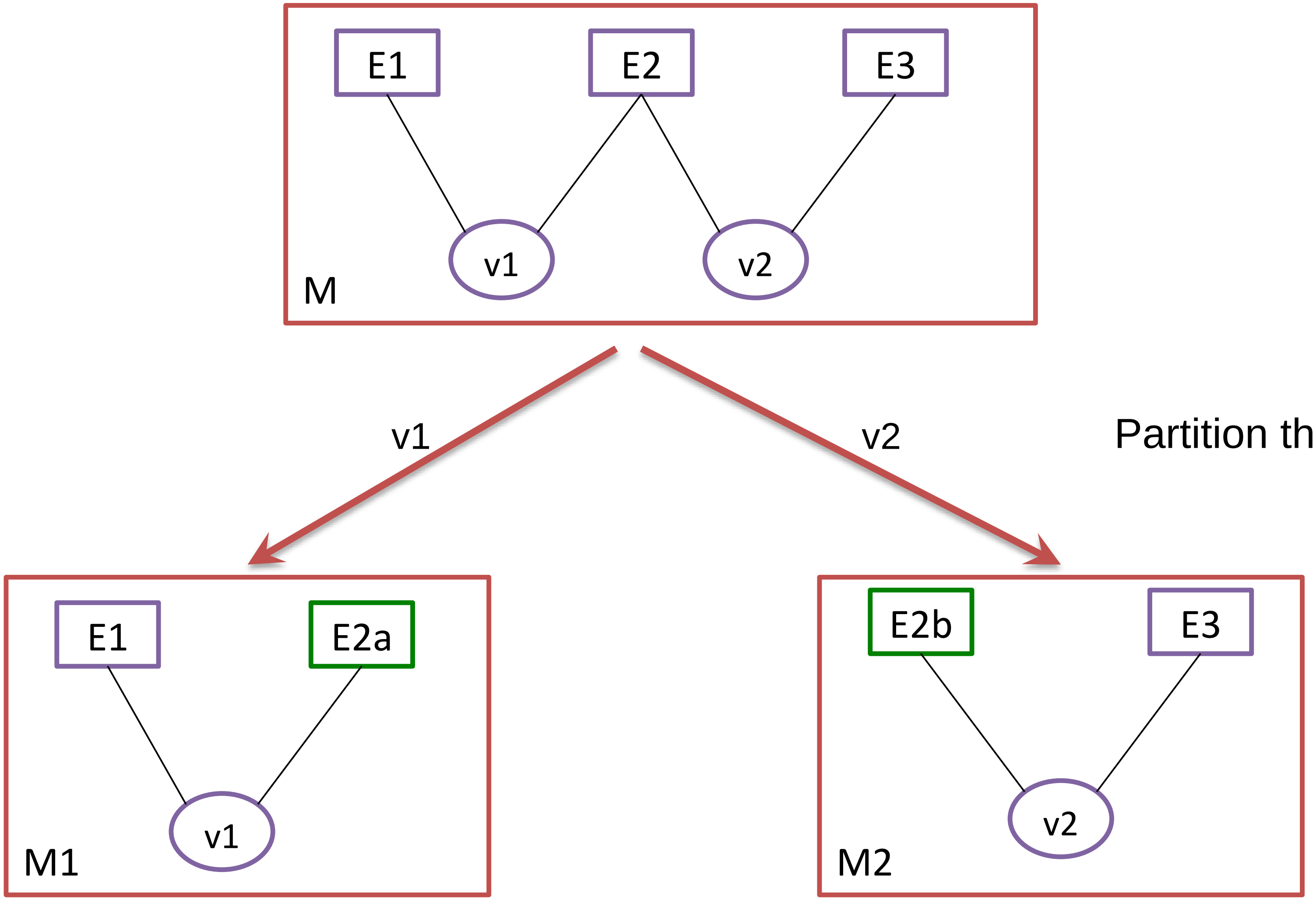
Michael Butler
users.ecs.soton.ac.uk/mjb
www.event-b.org

Marktoberdorf 2012

Decomposition

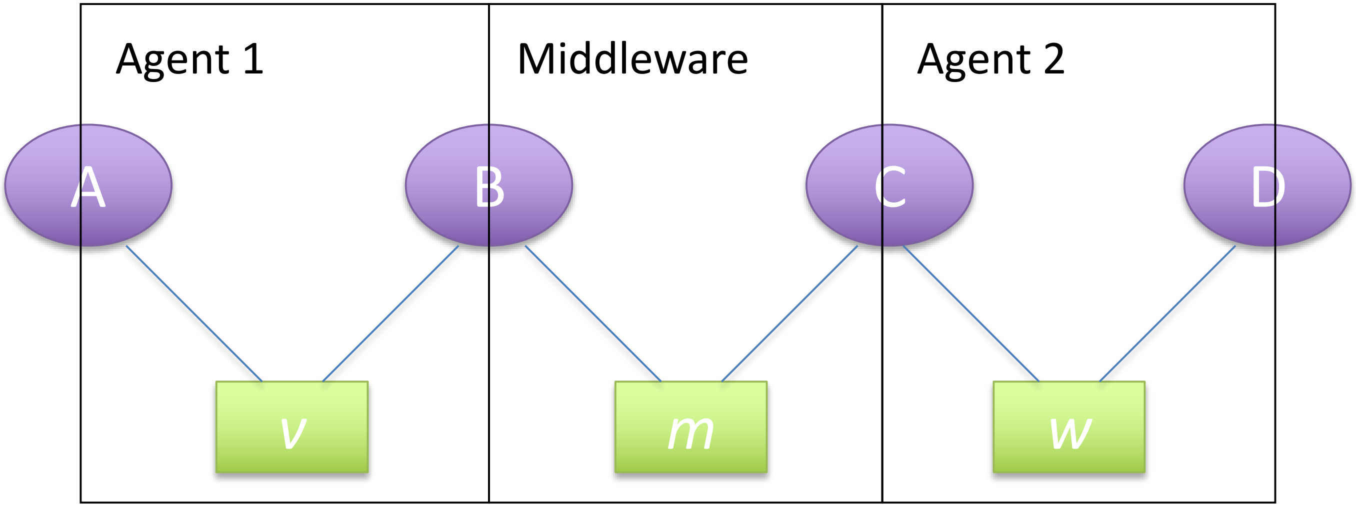
- Beneficial to model systems abstractly with little architectural structure and large atomic steps
  - e.g., *file transfer, replicated database transaction*
- Refinement and decomposition are used to add structure and then separate elements of the structure
- Atomicity decomposition: Decomposing large 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

Shared Event Decomposition



Partition the variables

Asynchronous distributed system

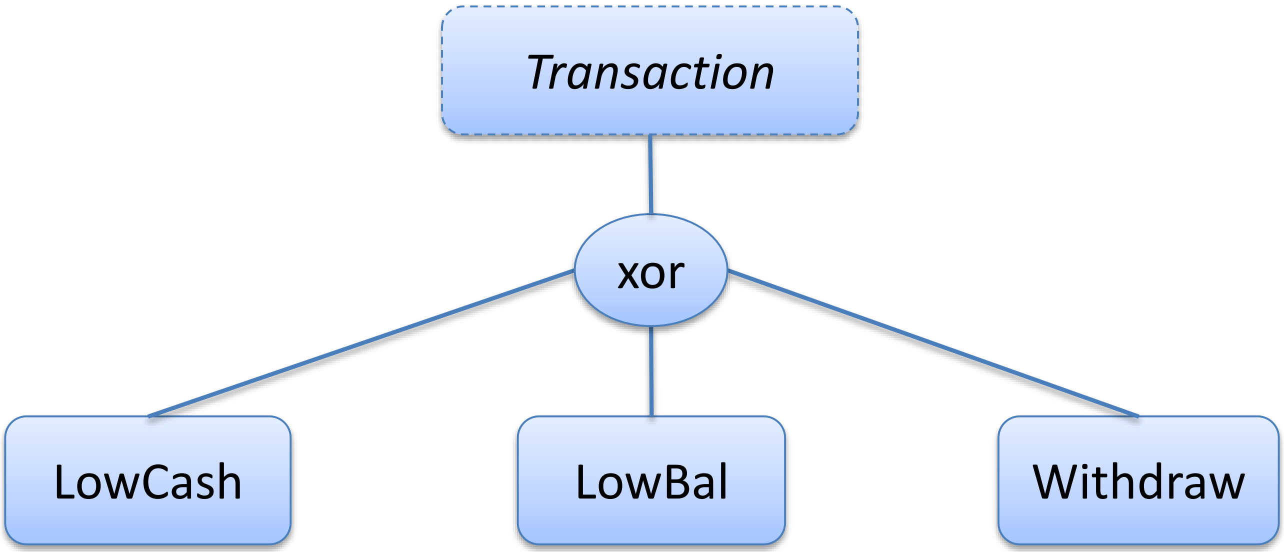


For distributed systems, agents do not interact directly.

Instead they interact via some middleware, e.g., the Internet

Atomicity *and* machine decomposition of ATM

Abstract Events for Cash *Withdrawl*

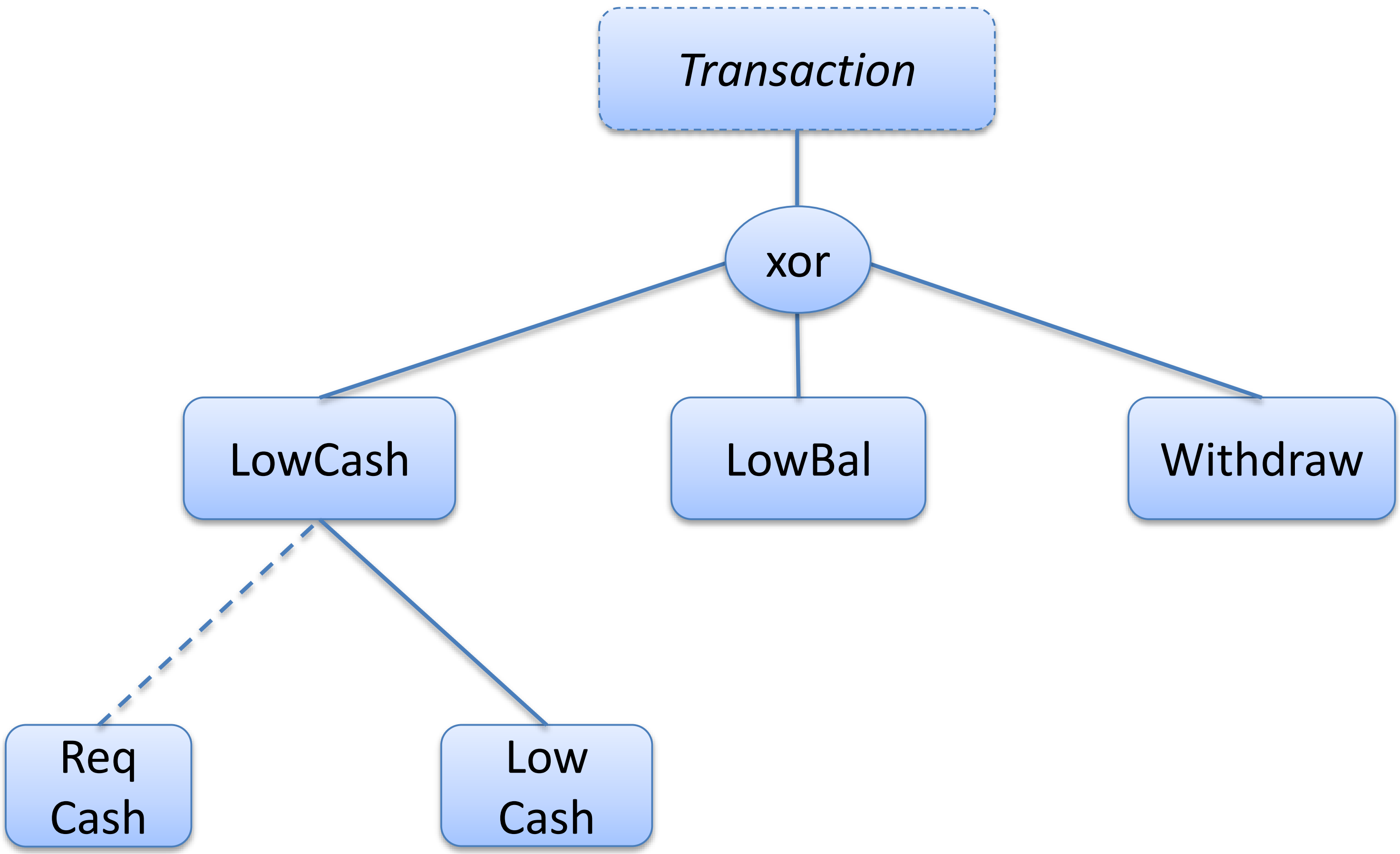


An ATM transaction results in one of three outcomes

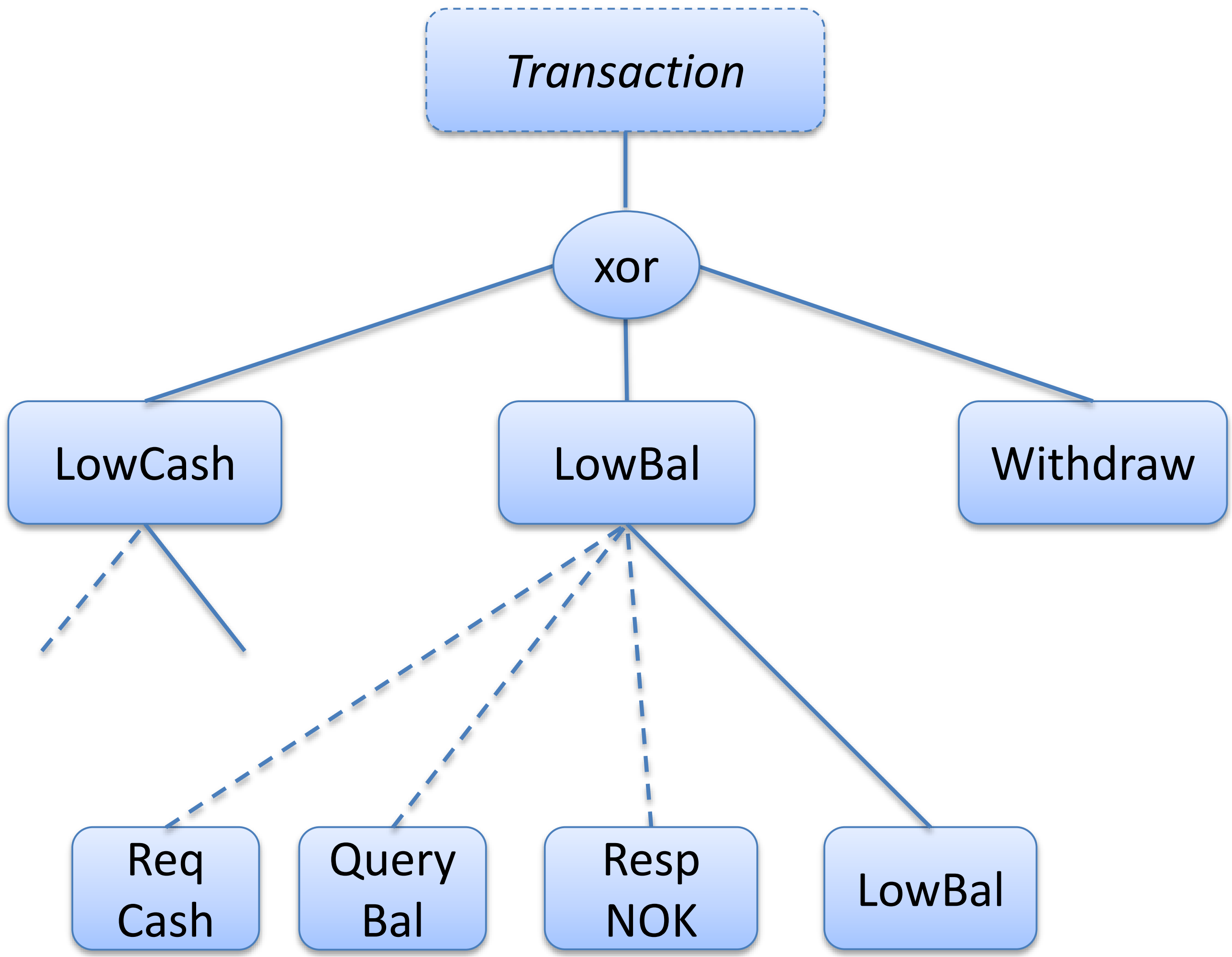
Distributed implementation with ATM and Bank server:

- *LowCash* only affects the ATM
- *LowBal* and *Withdraw* affect ATM and Bank

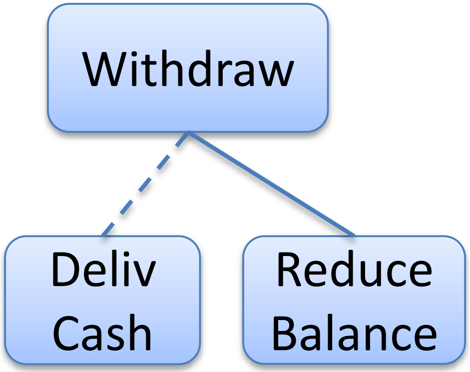
LowCash: separate user request from ATM response



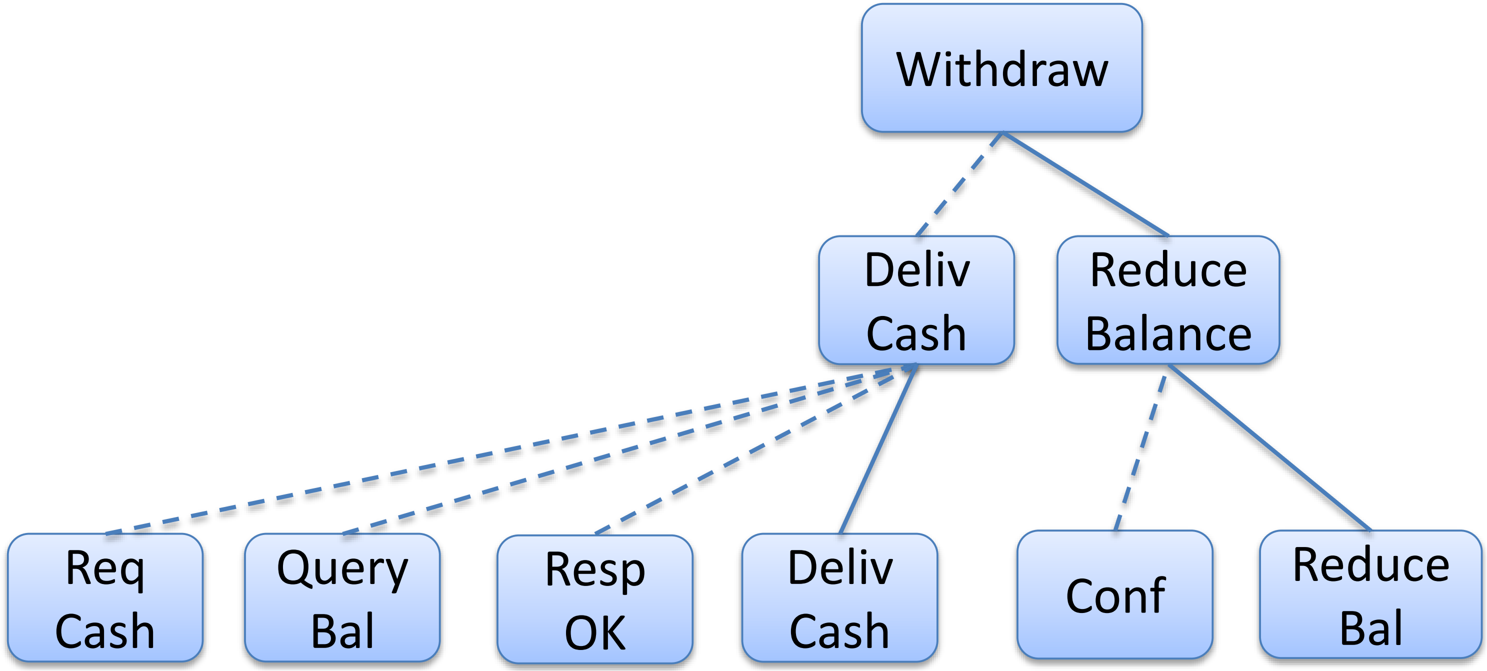
LowBal: introduce protocol steps



Withdraw: separate cash delivery and balance reduction

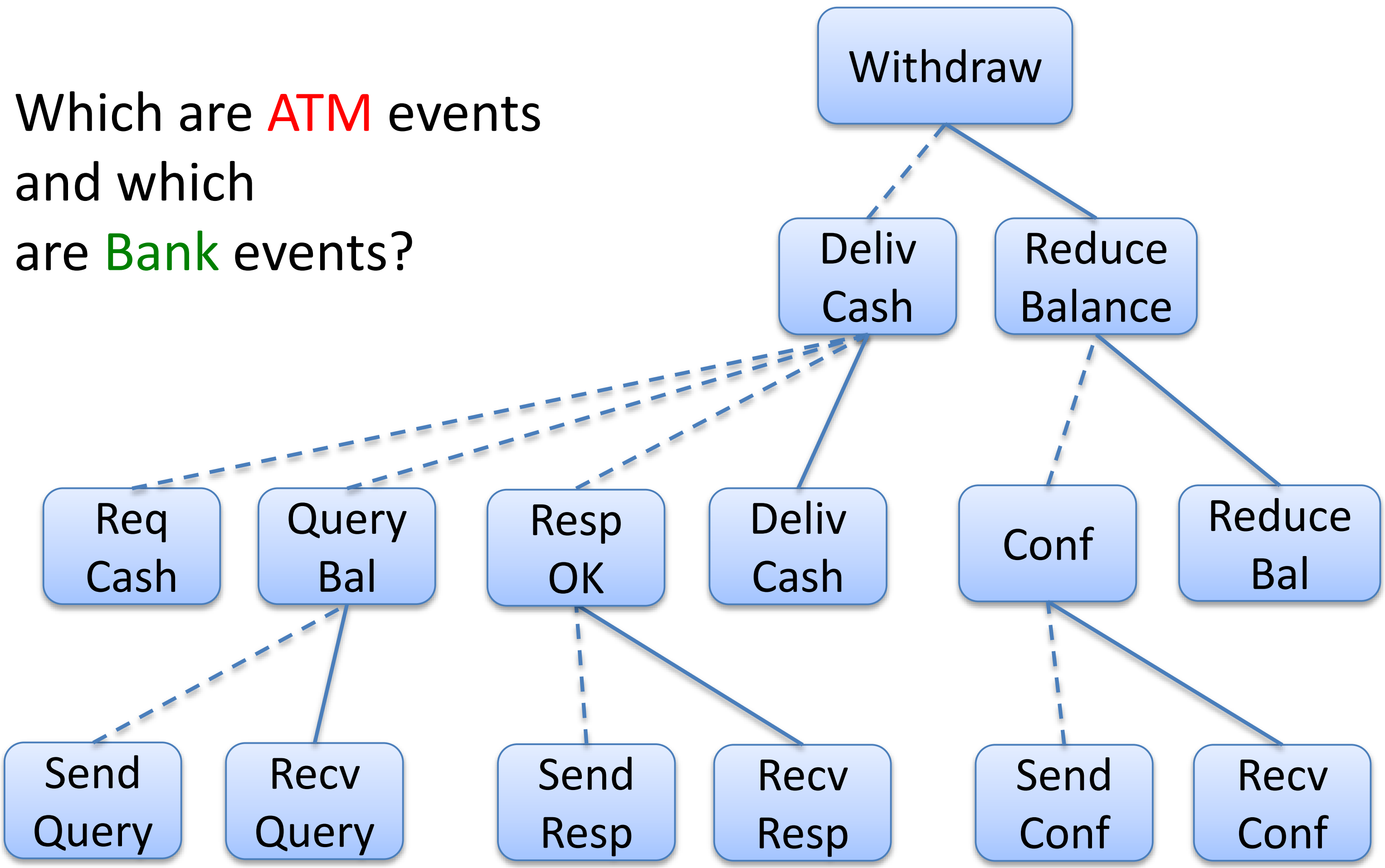


Withdraw: protocol steps

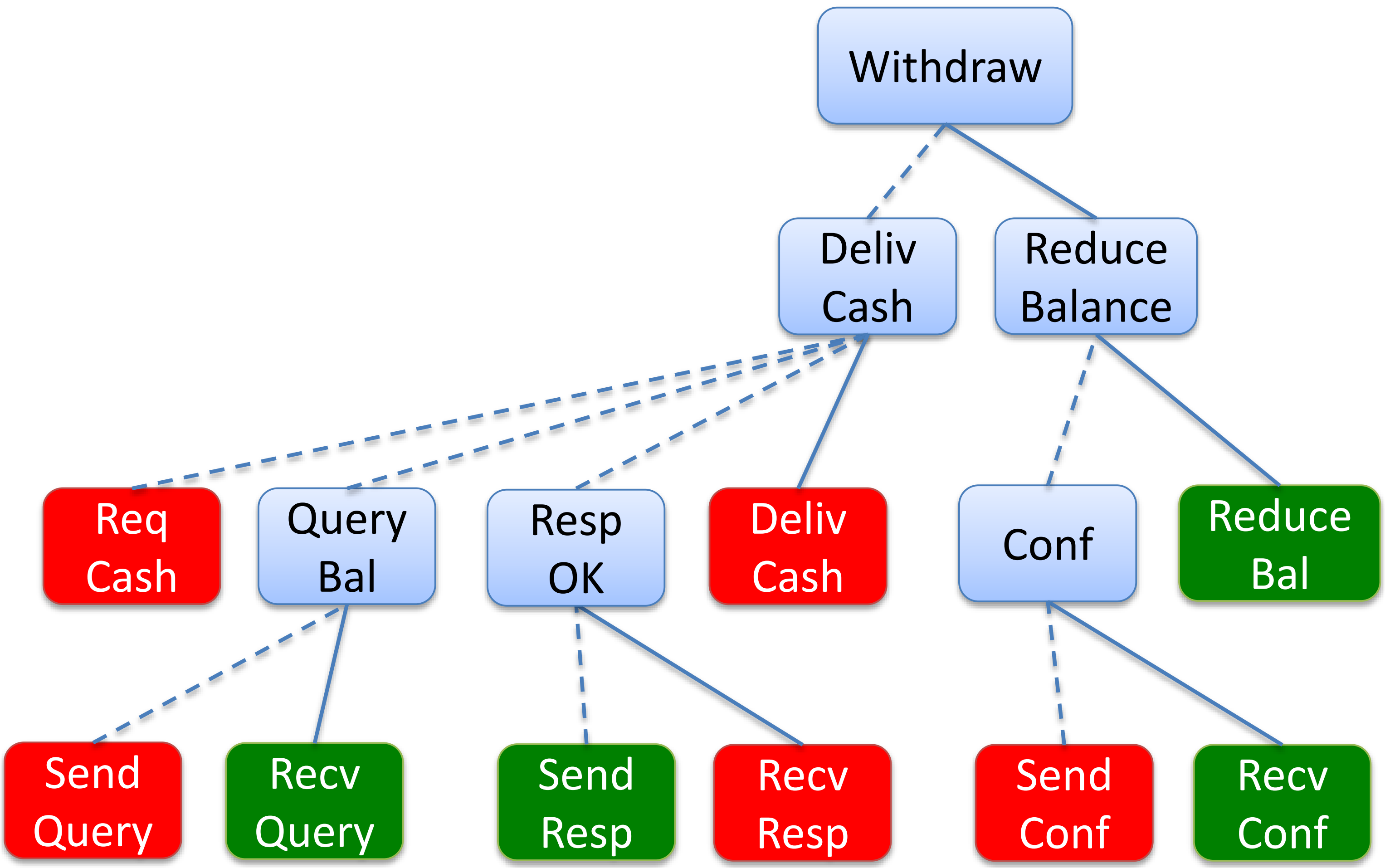


Separate sending and receiving for protocol steps

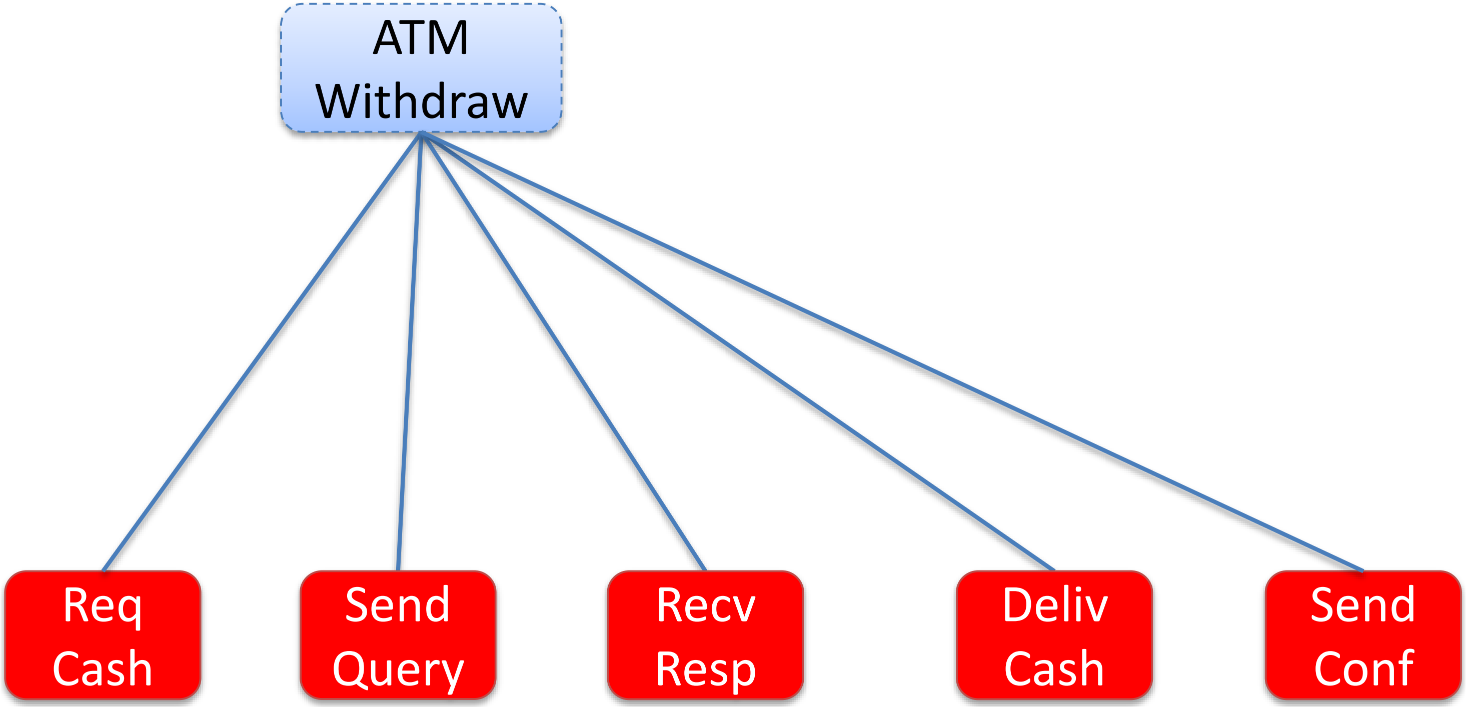
Which are ATM events
and which
are Bank events?



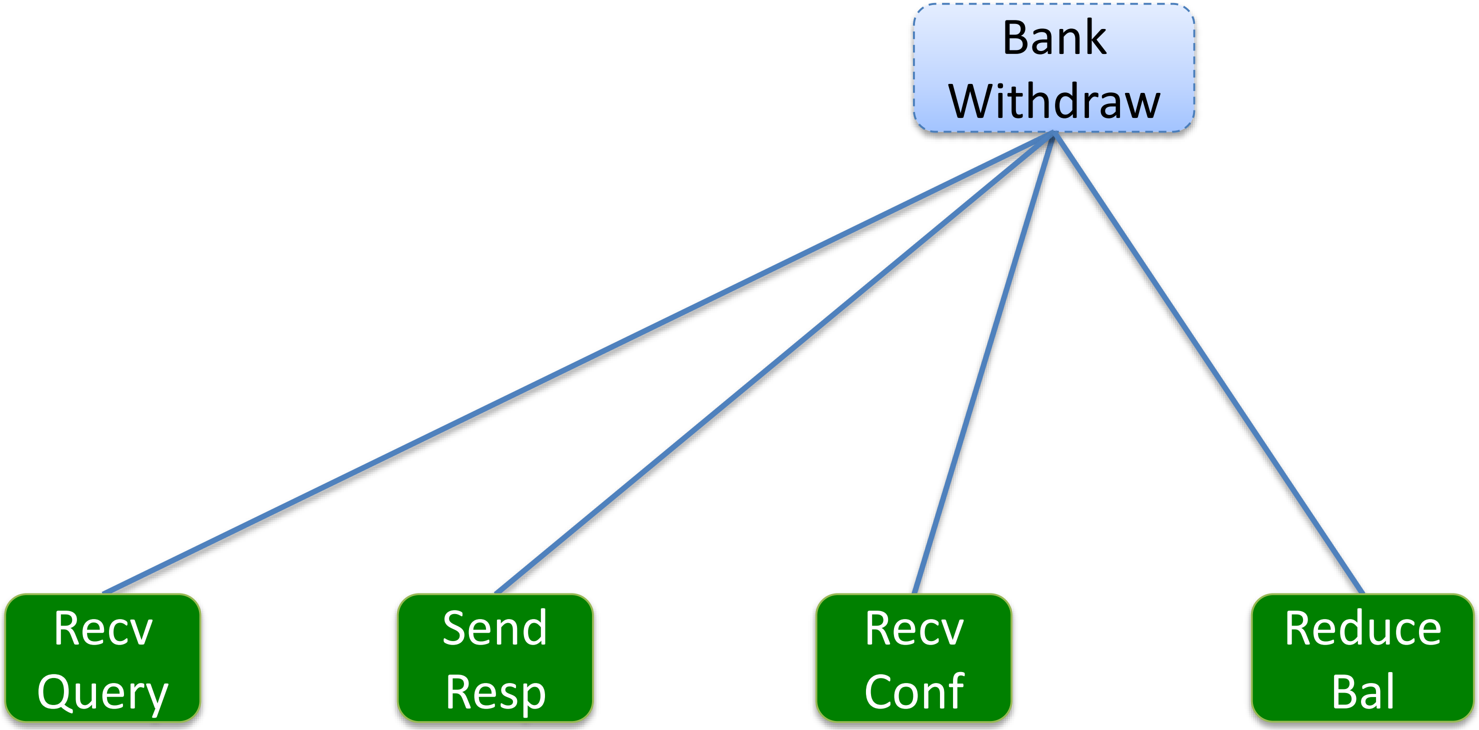
Distinguish ATM and Bank events



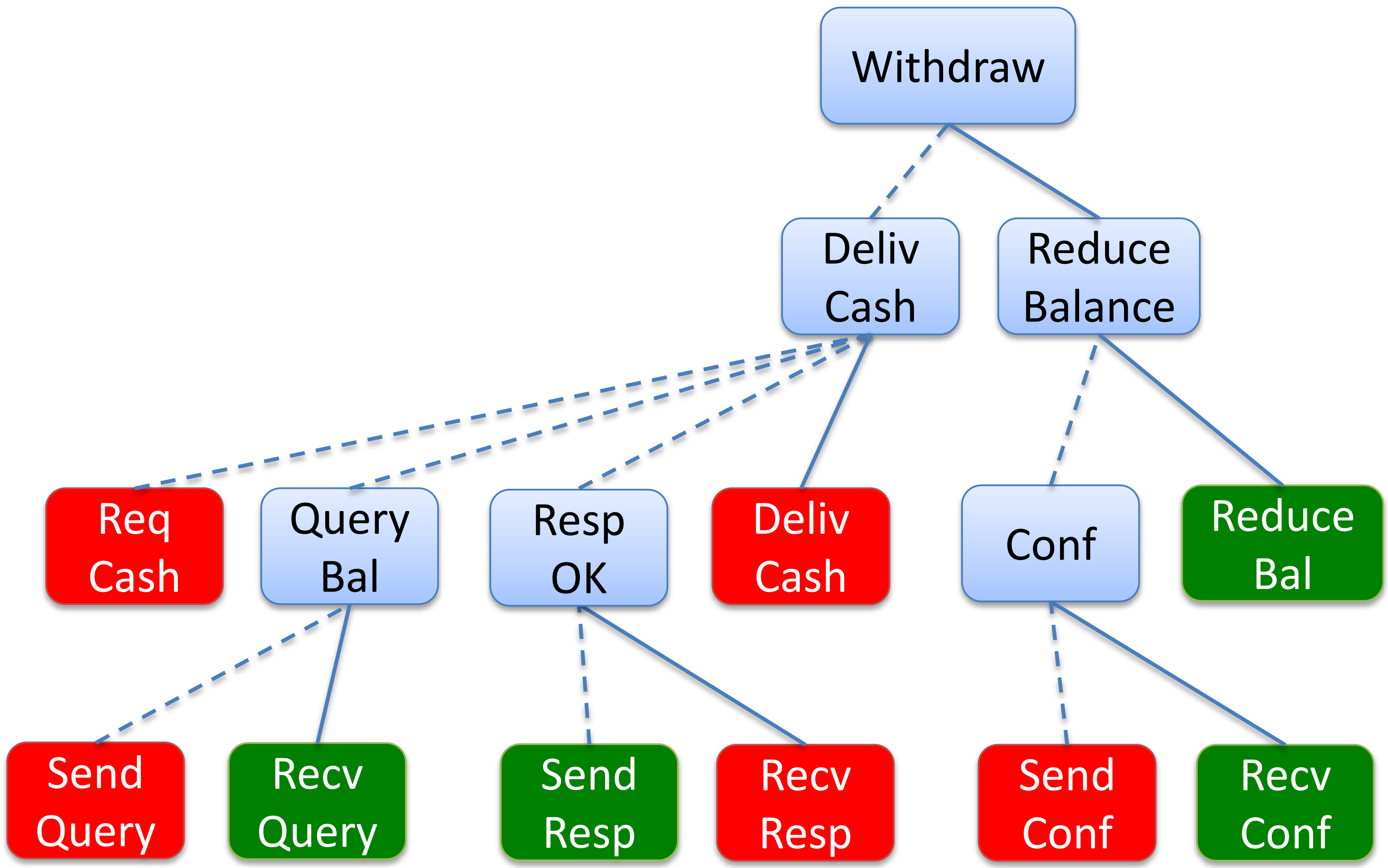
Extract ATM behaviour



Extract Bank behaviour

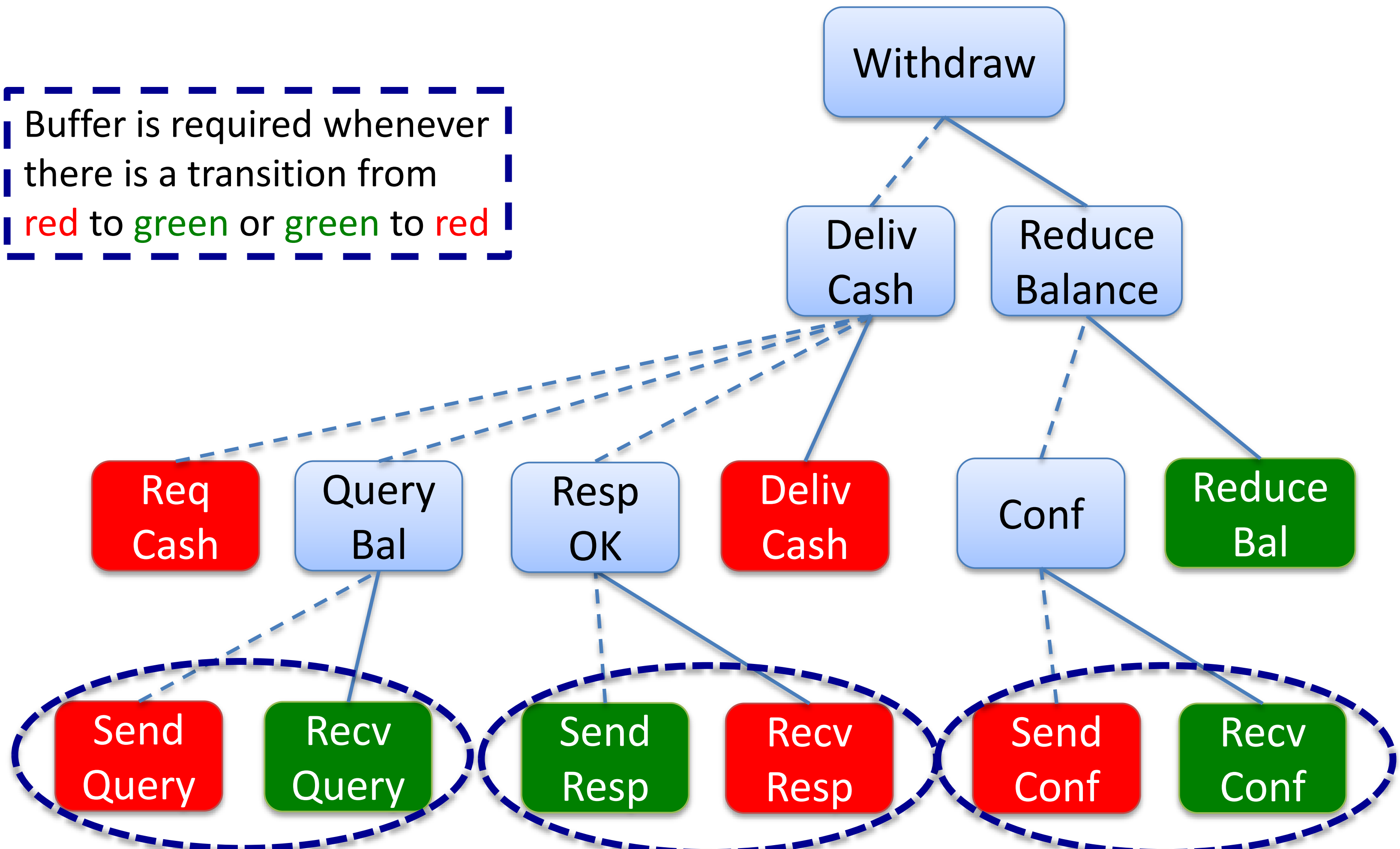


What about communication between ATM and Bank ?

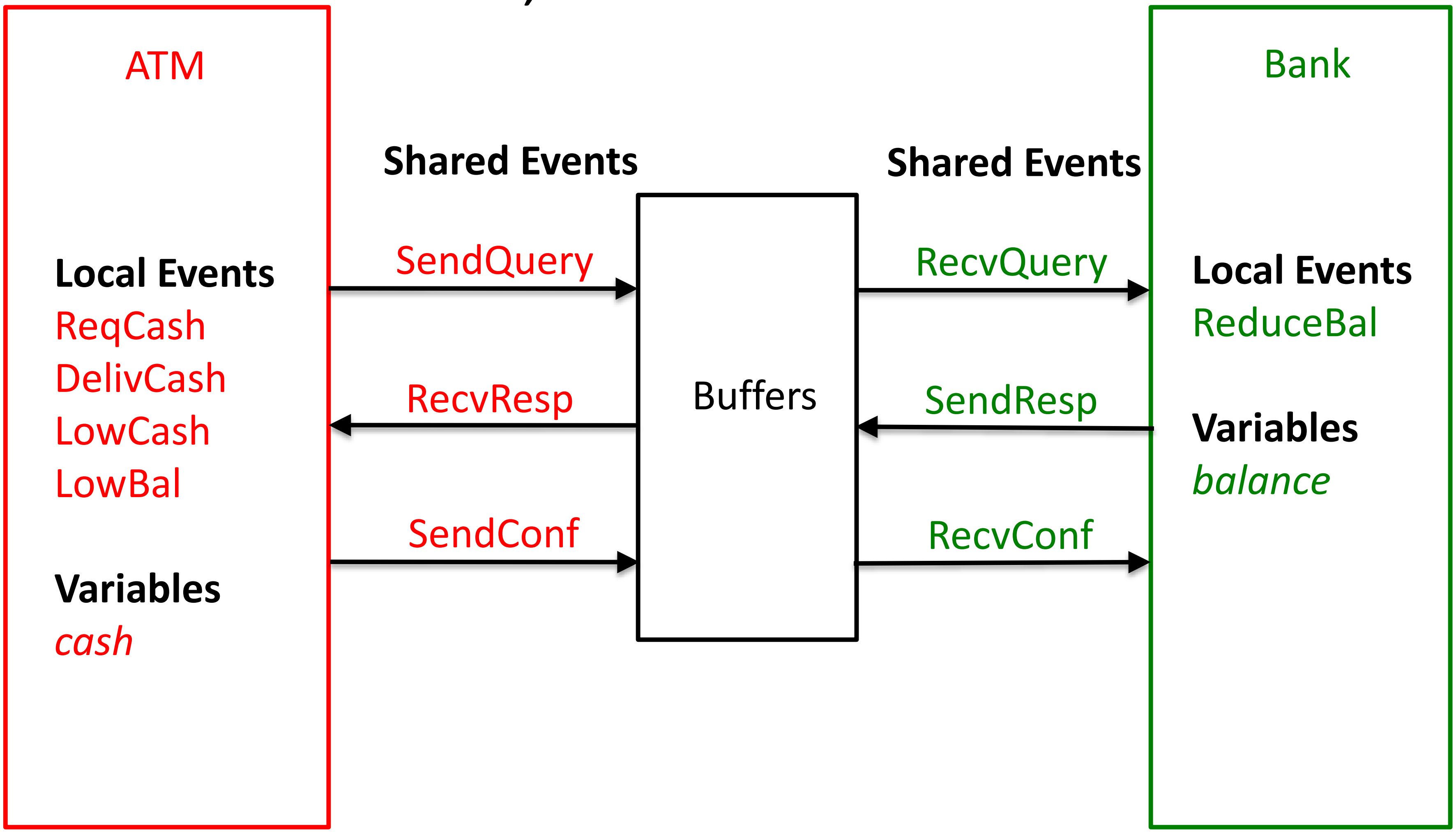


Identify need for asynchronous communication

Buffer is required whenever there is a transition from red to green or green to red



Decompose model into ATM, Bank and Buffers

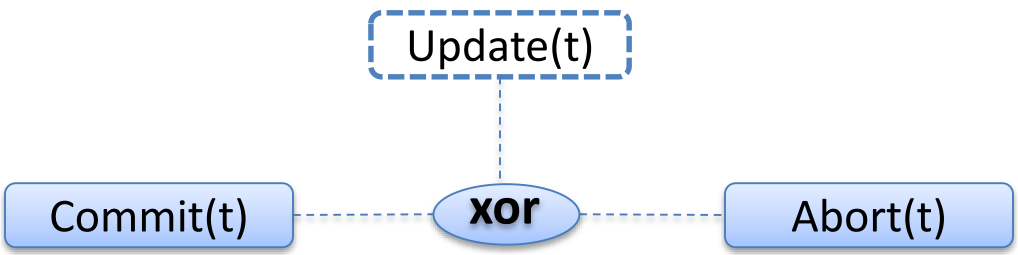


Decomposition of replicated database

Abstraction of Distributed Database

Abstract model:

$$db \in object \rightarrow DATA$$

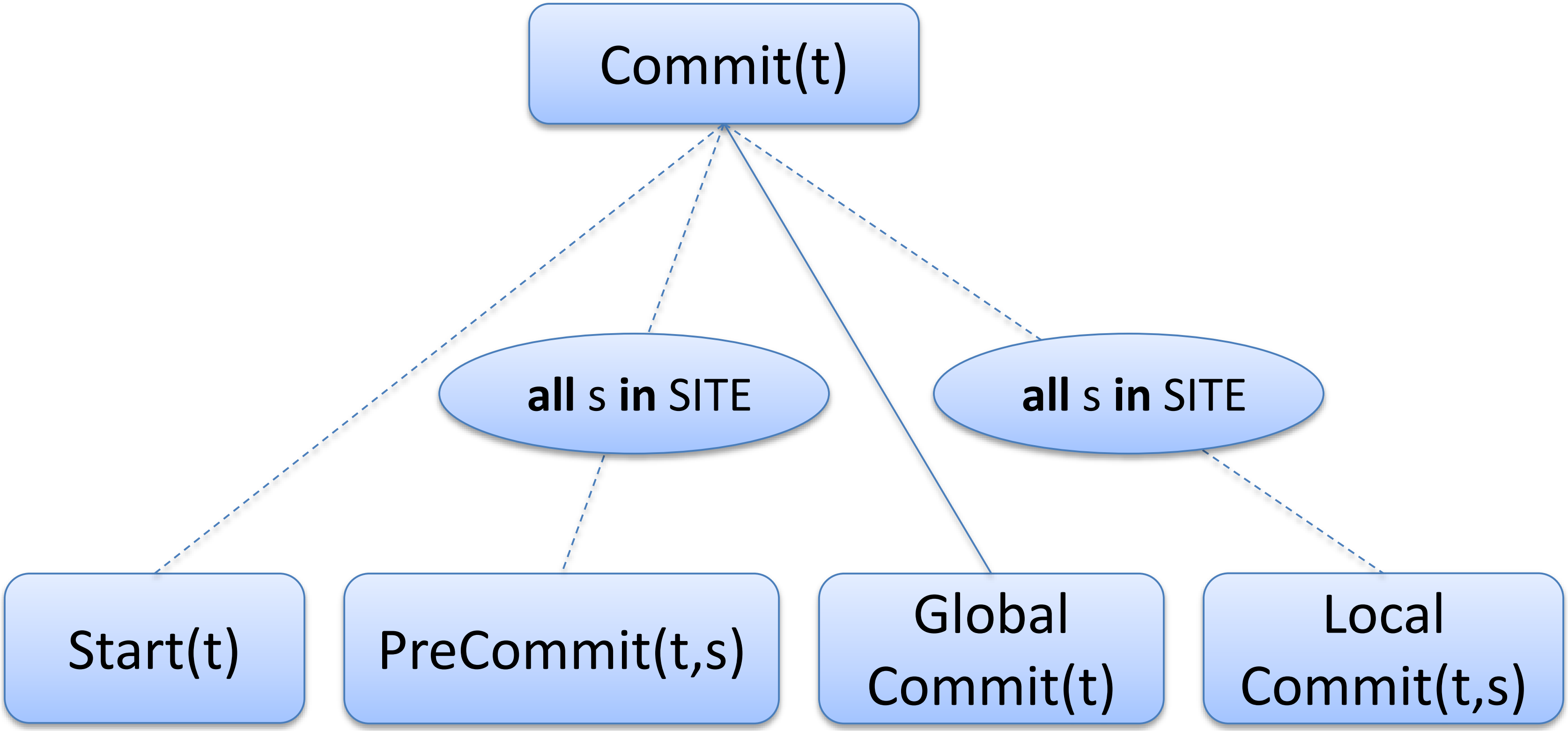


Refinement by replicated database

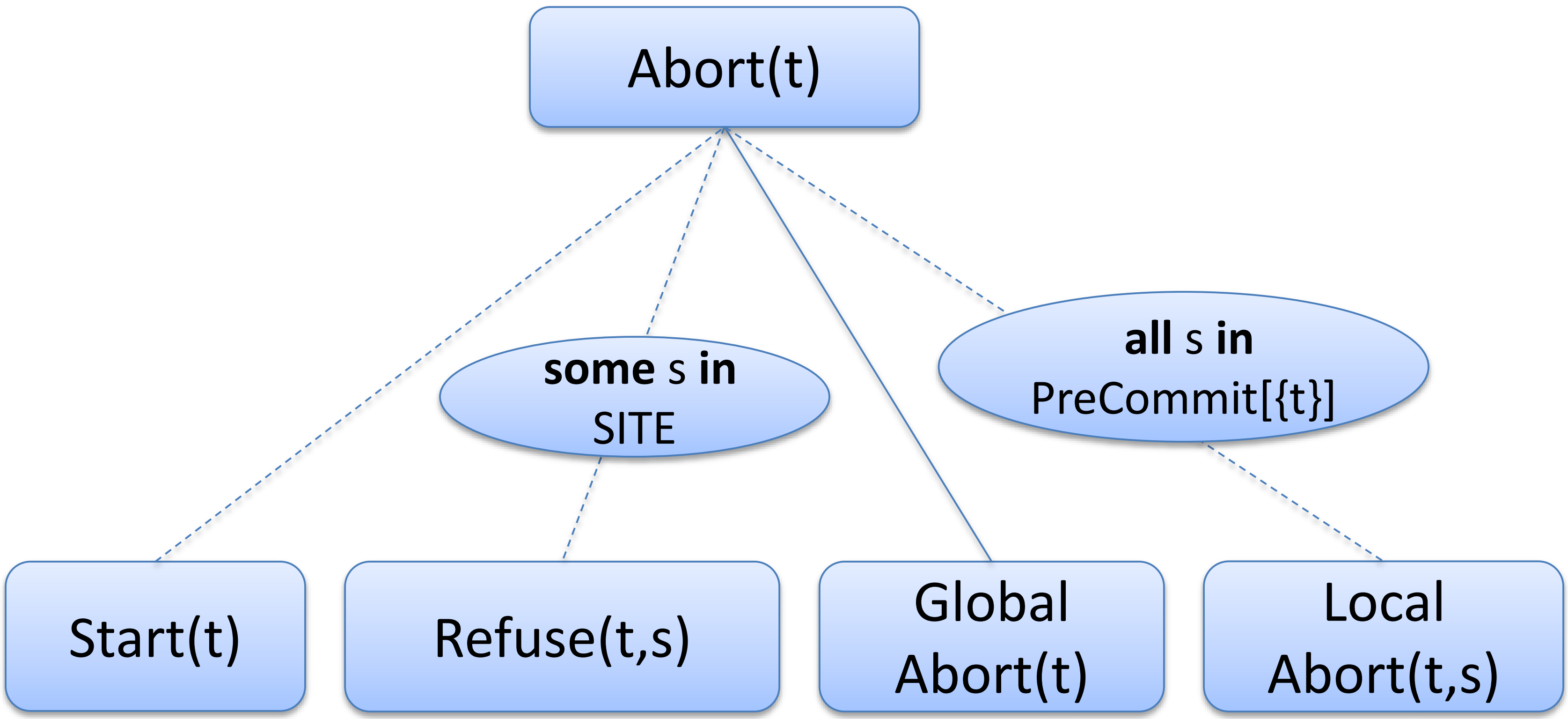
$$\text{ldb} \in \text{site} \rightarrow (\text{object} \rightarrow \text{DATA})$$

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

Structured refinement of *Commit*



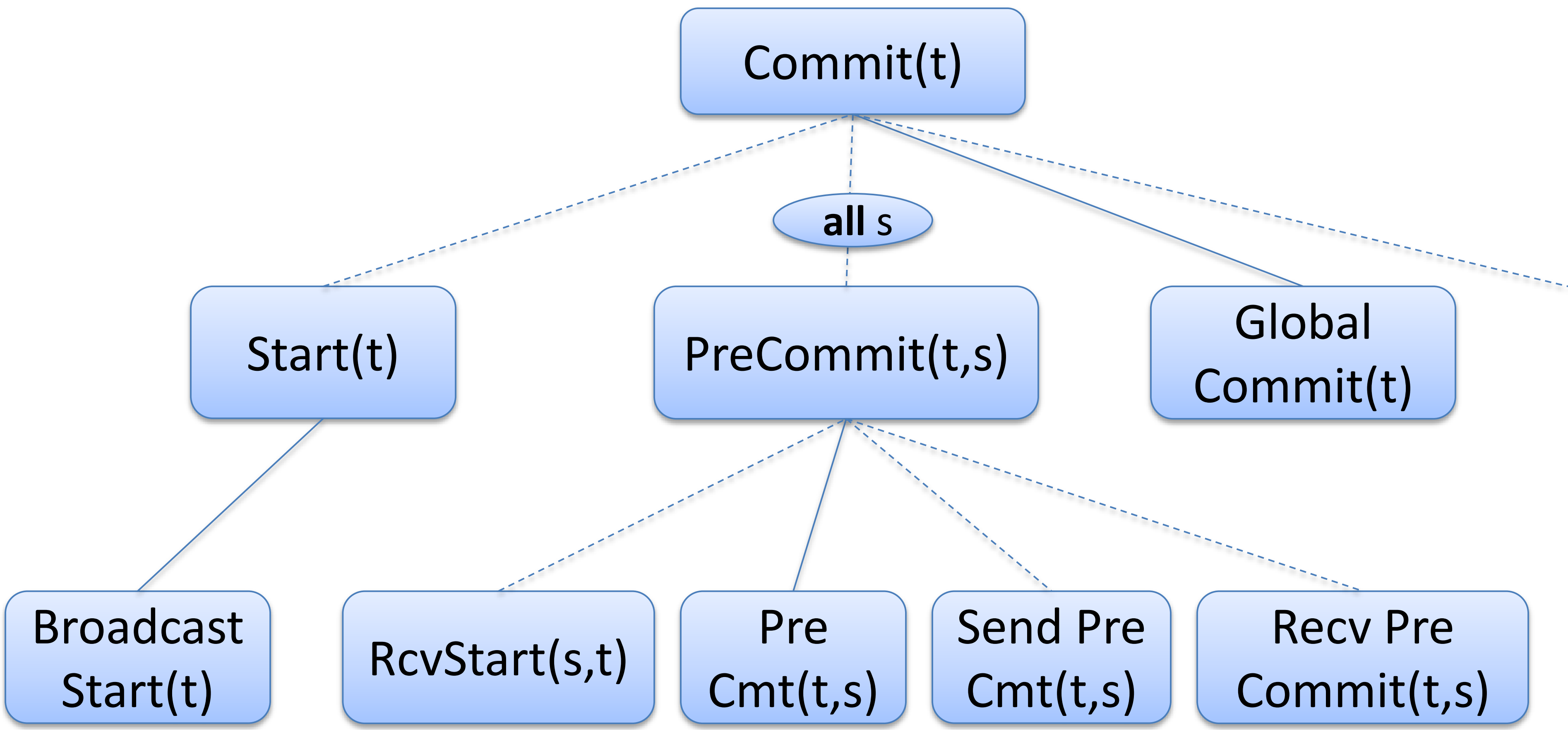
Structured refinement of *Abort*



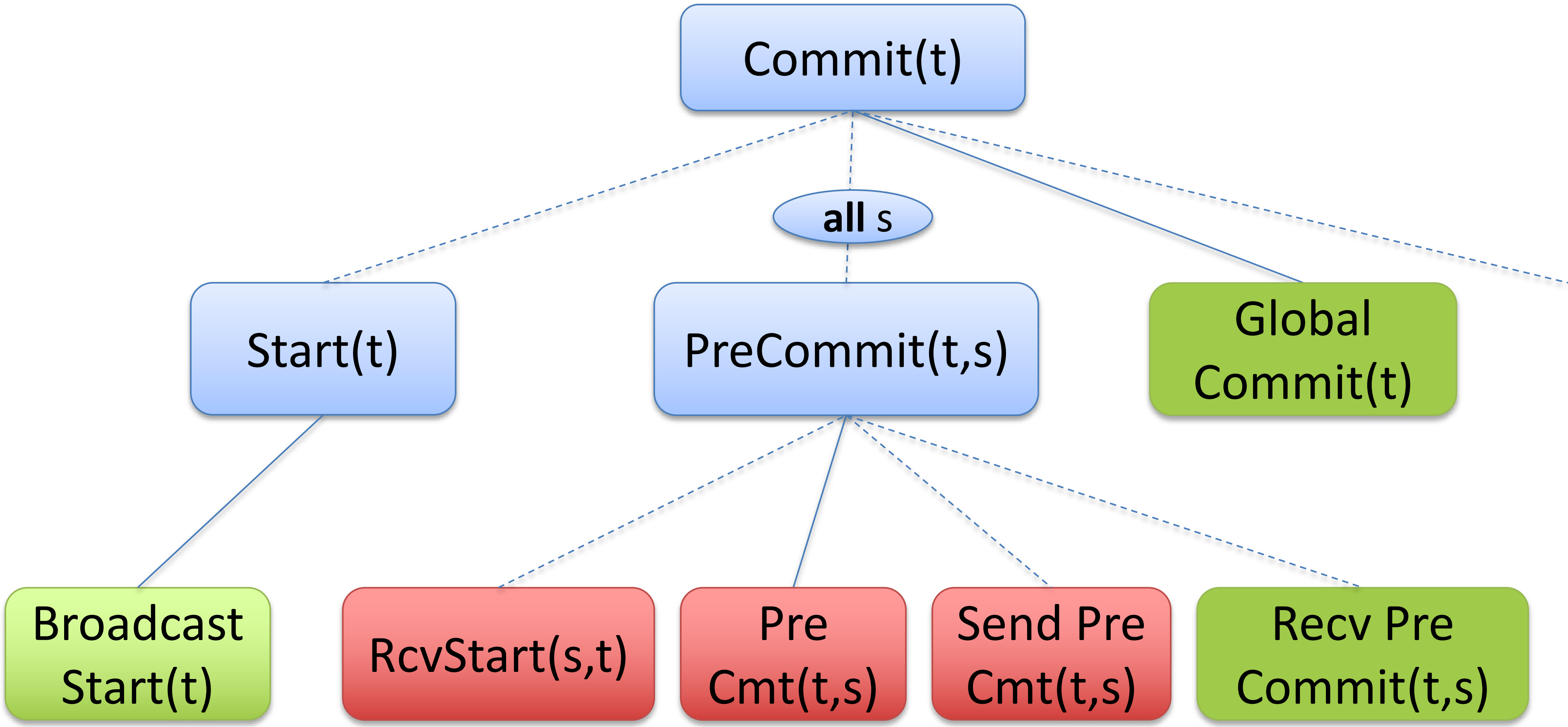
Towards a distributed system

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

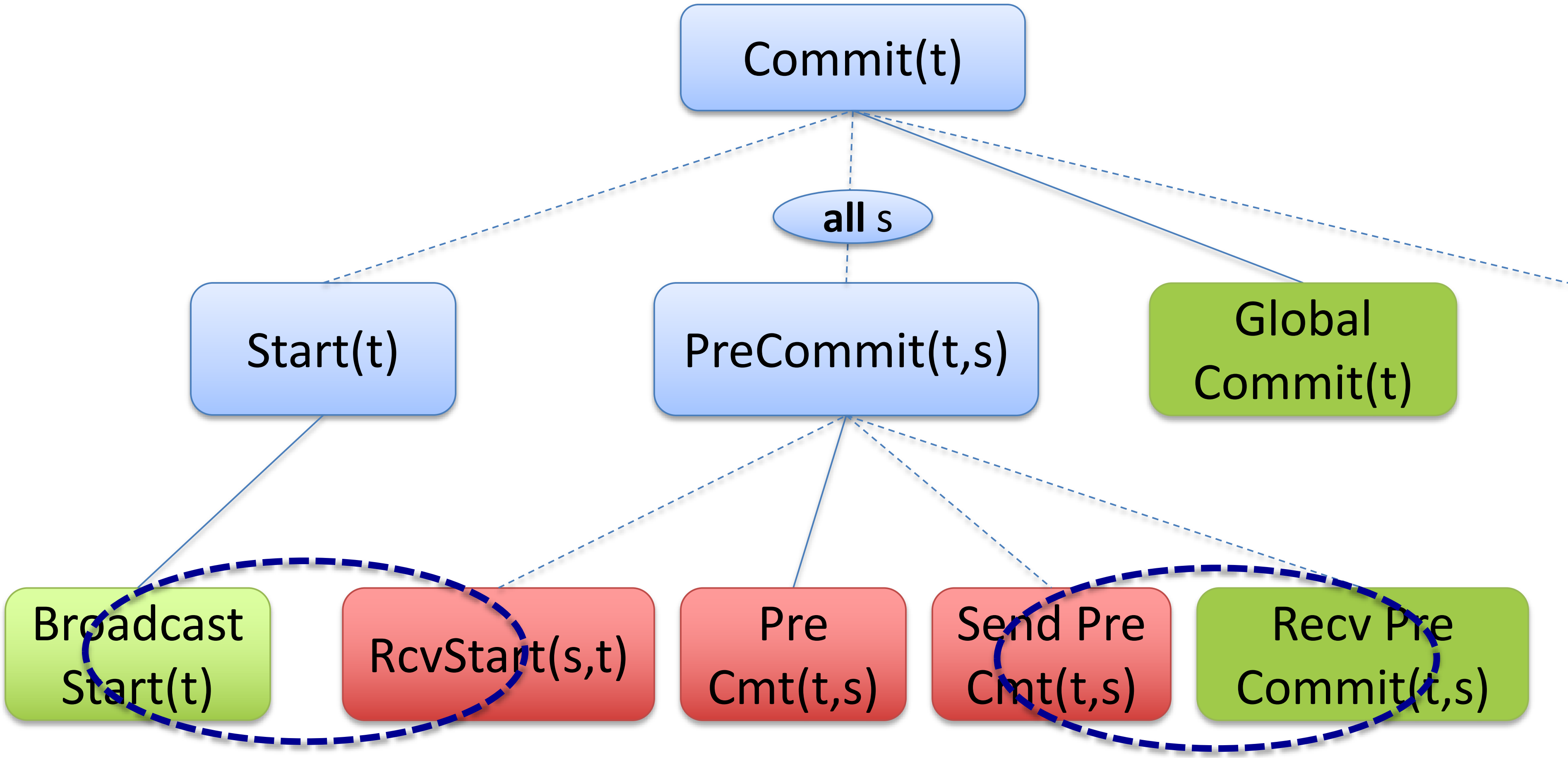
Introducing messaging



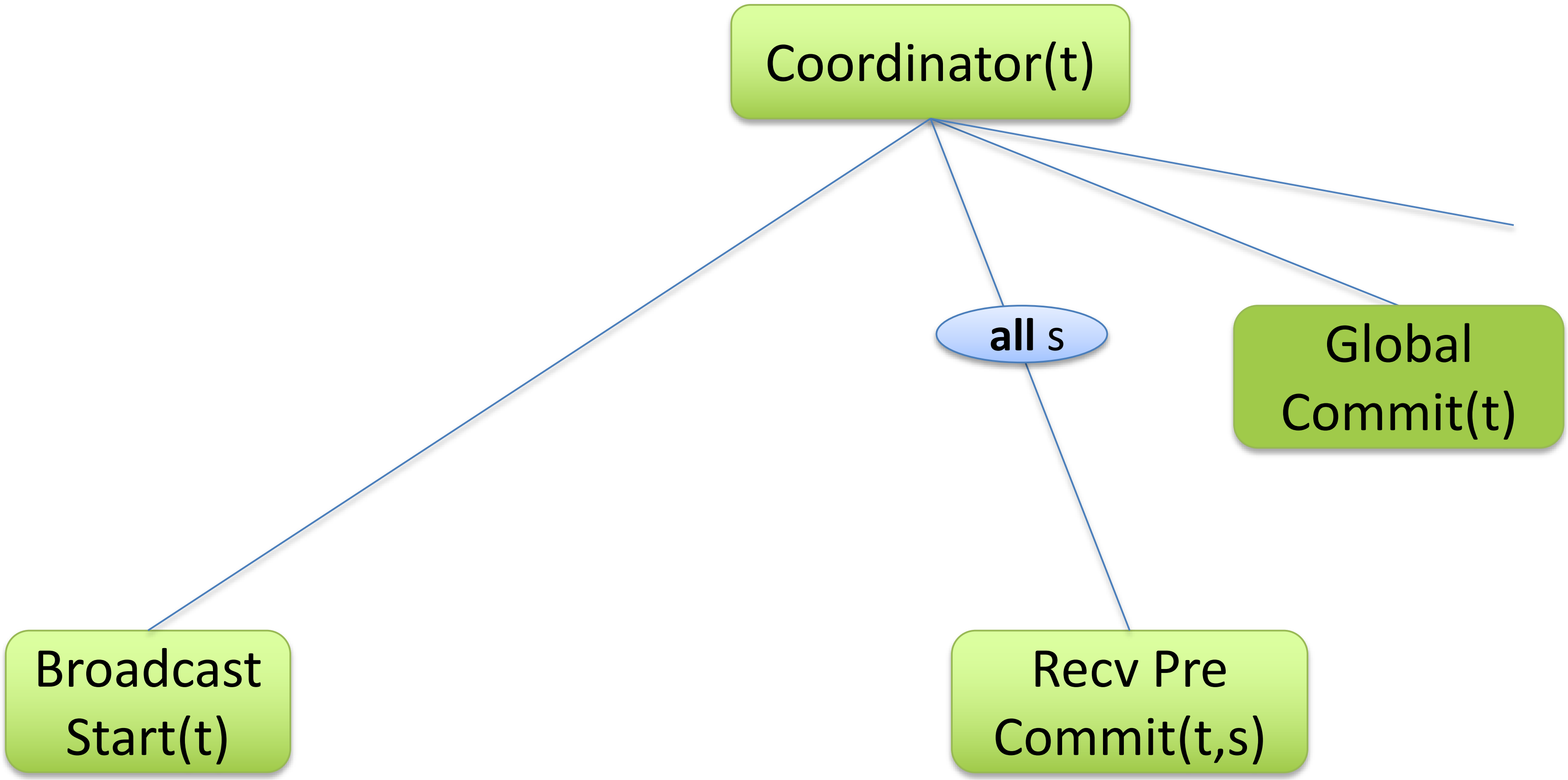
Separate coordinator and worker events



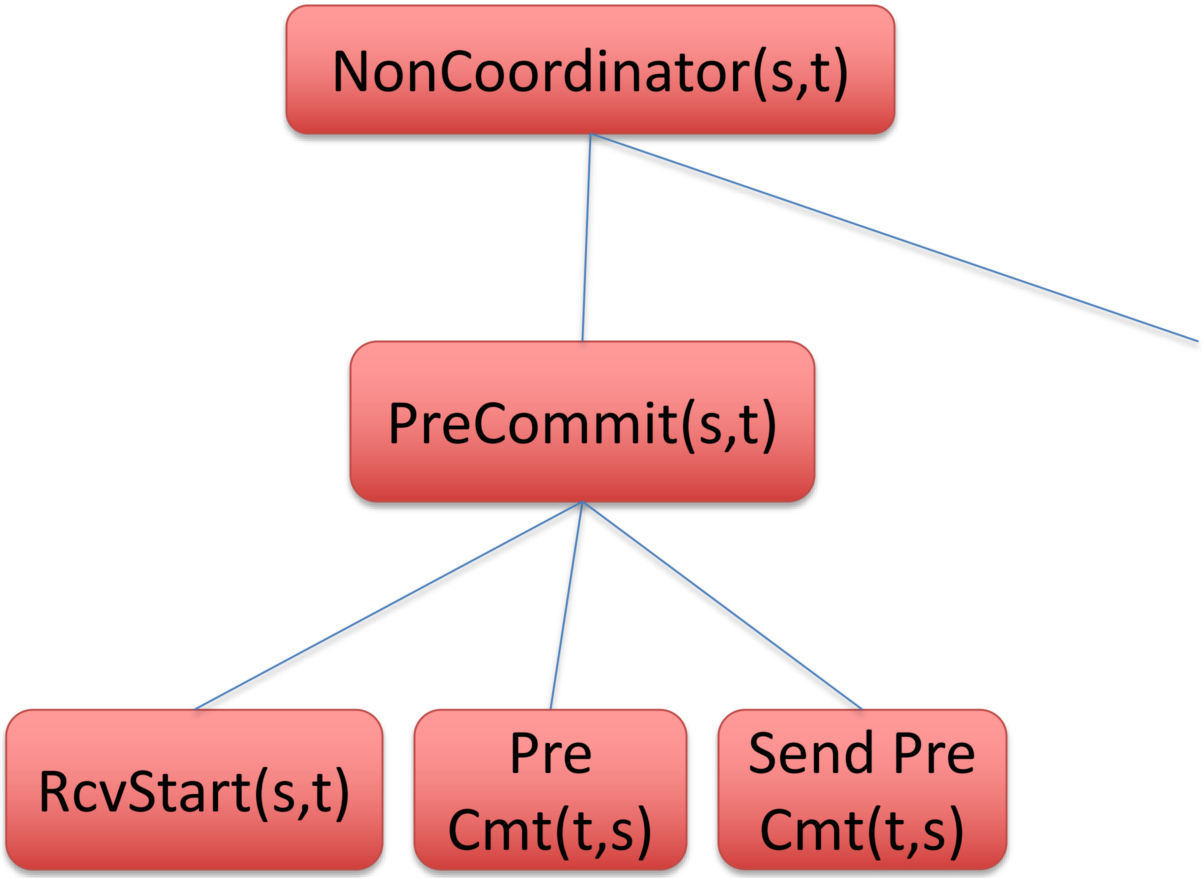
Identify communications buffers



Coordinator abstract program



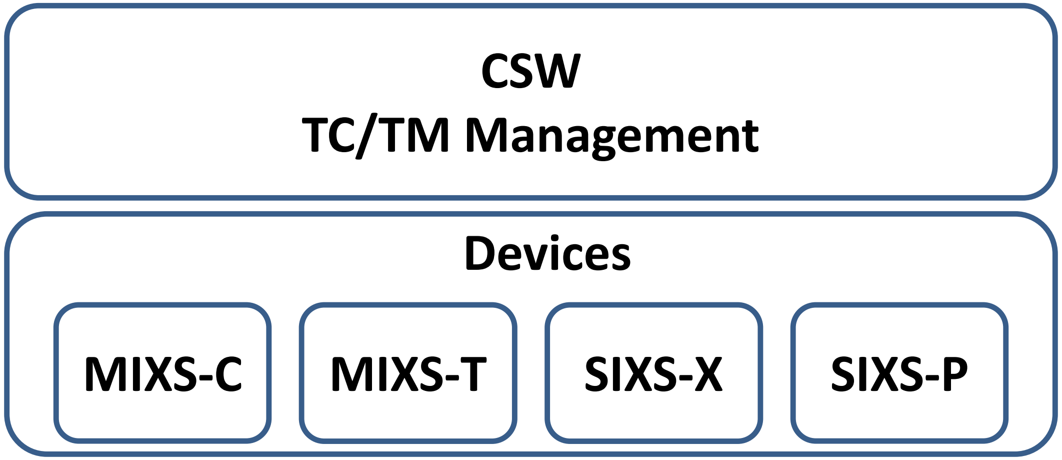
Worker behaviour



Other case studies

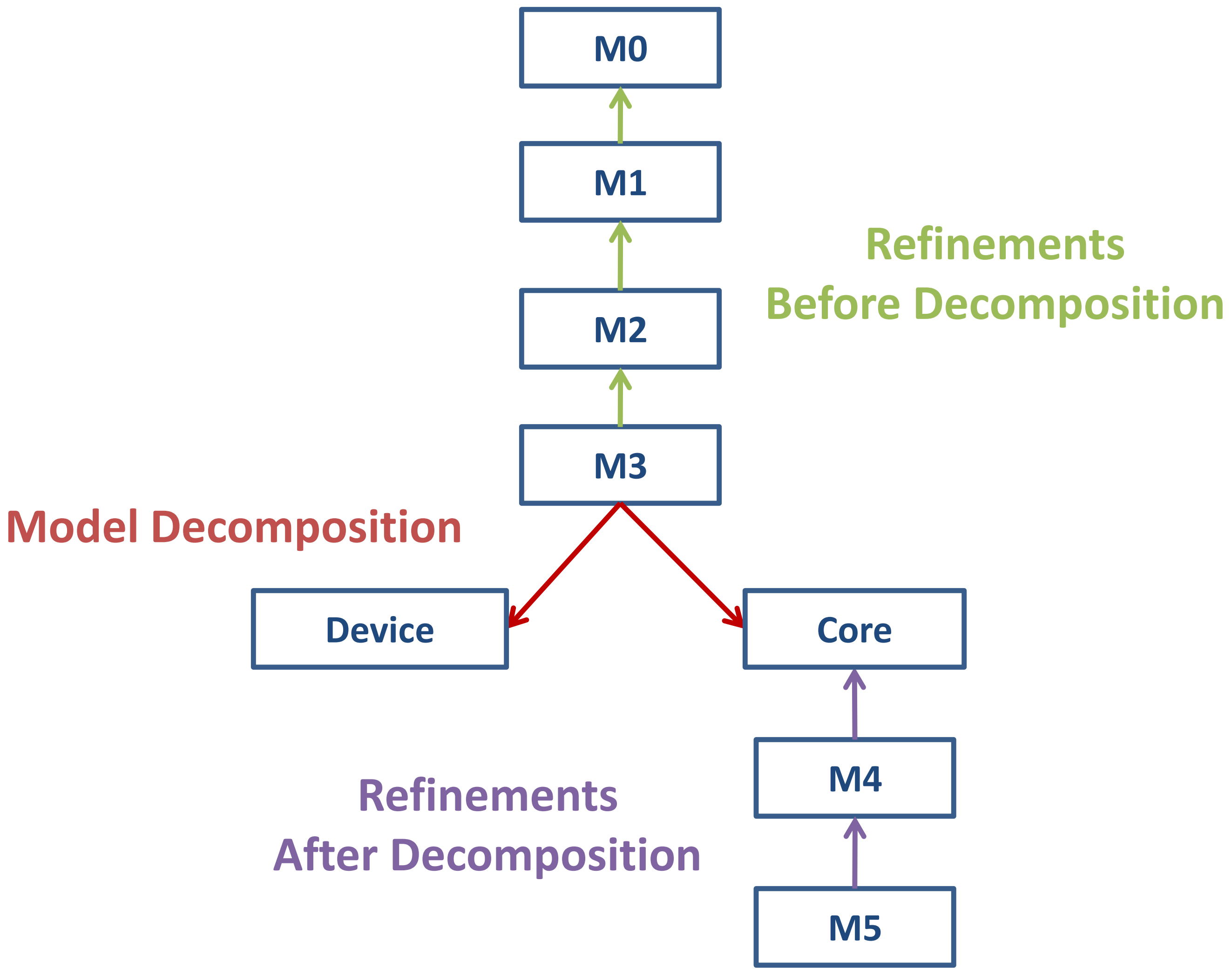
- Multimedia protocol (Asieh Salehi)
- Data manipulation in satellite (Asieh Salehi)
- Railway network (Renato Silva)
- Automotive control (Sanaz Yeganefard)

Space Craft System

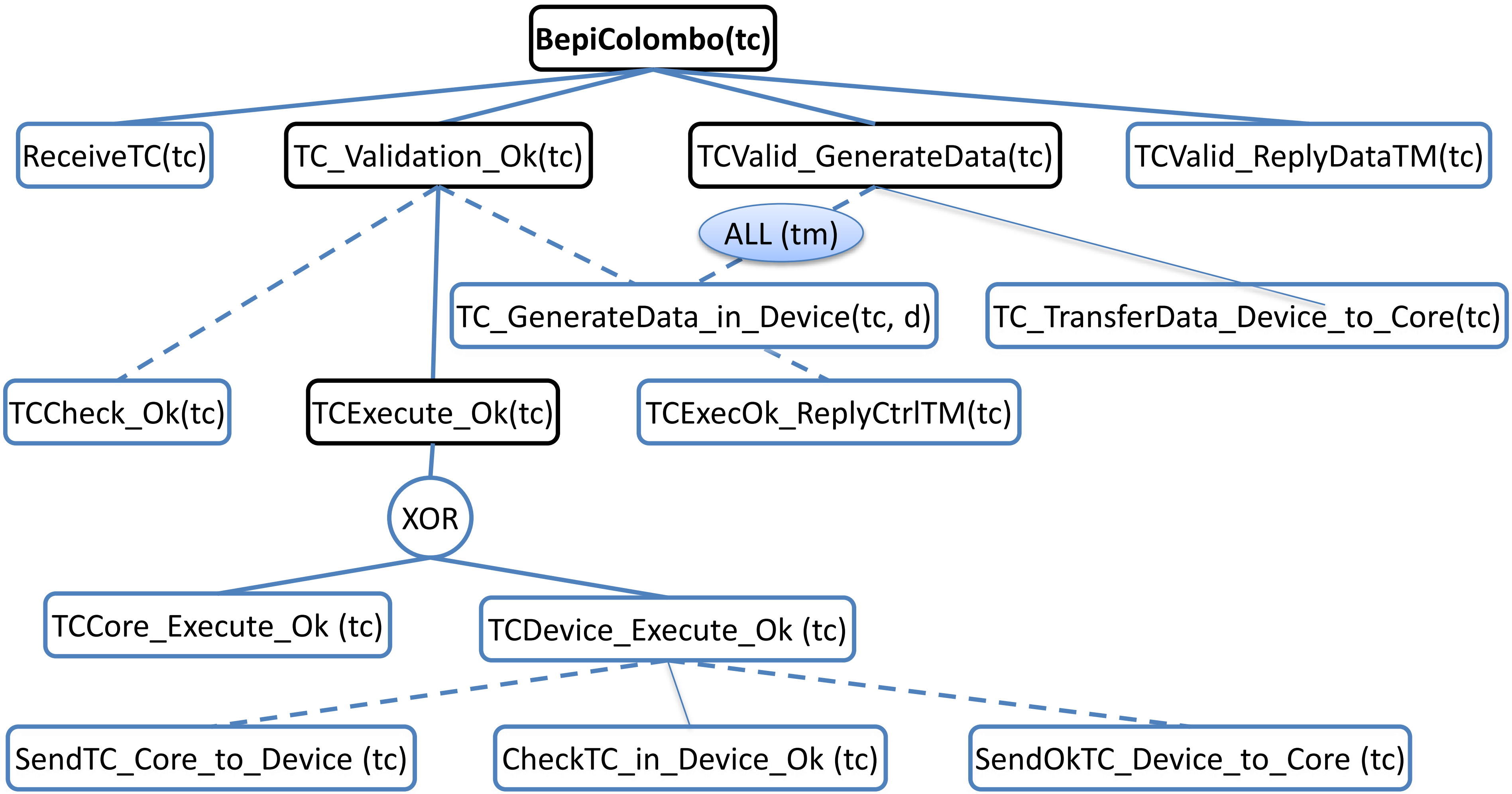


- A TeleCommand (TC) is received by the Core from Earth.
- The syntax of the received TC is check in the core.
- Further semantic checking has to be carried out either in the core or devices based on the type of TCs.
- For all received TCs, a control TeleMessage (TM) is generated and sent back to Earth.
- For some particular types of TC, one or more data TMs are generated and sent back to Earth.

Space Craft Development

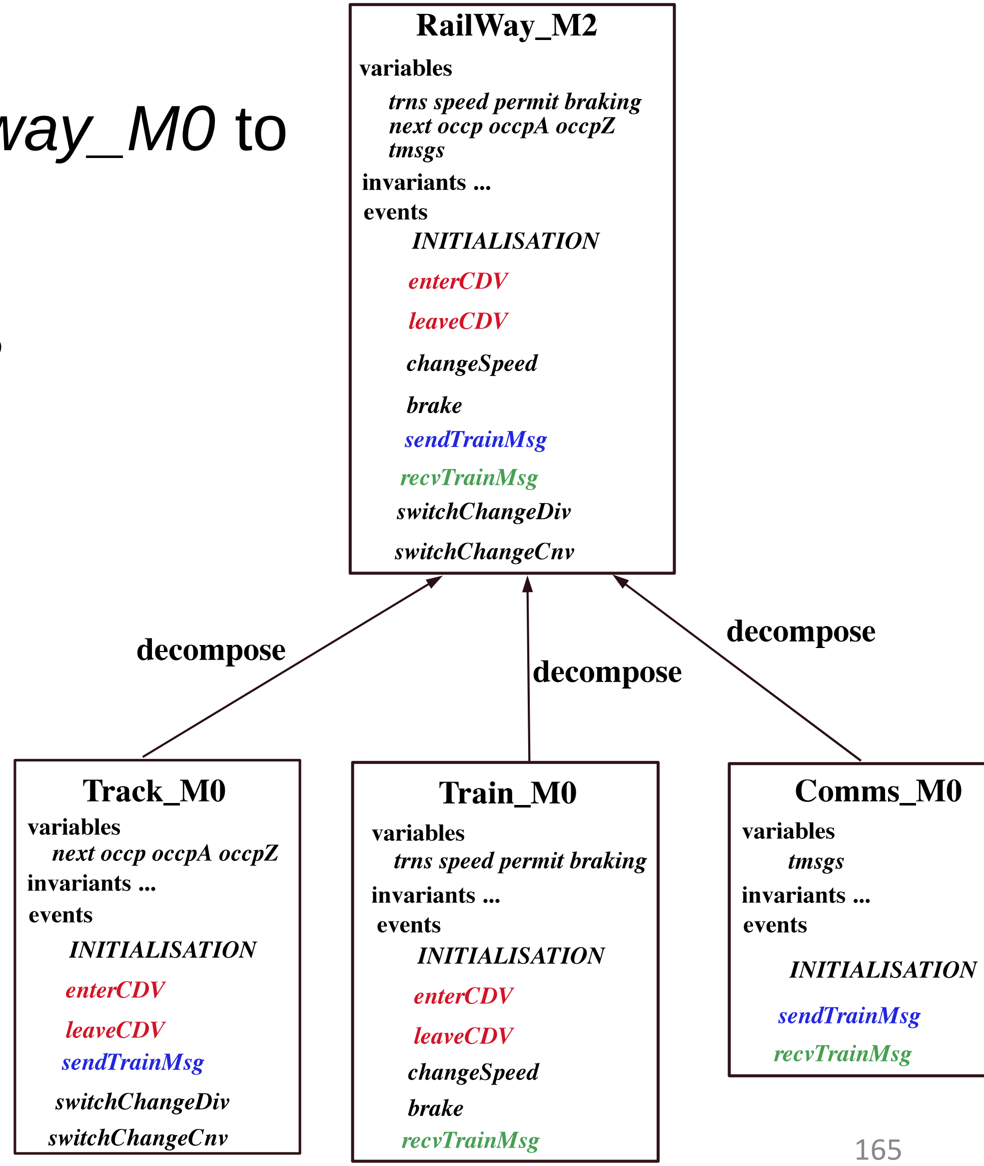


Event refinement structure



Railway System Decomposition

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



Some references

- Butler, M. (2009) ***Decomposition Structures for Event-B***. In: Integrated Formal Methods iFM2009, LNCS 5423.
- Abrial, J.-R. and Hallerstede, S. (2007) ***Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B***. Fundam. Inf., 77(1-2).

- Silva, R., Pascal, C., Hoang, T. S. and Butler, M. (2011) ***Decomposition Tool for Event-B***. Software: Practice and Experience, 41 (2).
- Salehi Fathabadi, A., Rezazadeh, A. and Butler, M. (2011) ***Applying Atomicity and Model Decomposition to a Space Craft System in Event-B***. In: Third NASA Formal Methods Symposium, 2011.
- Salehi Fathabadi, A., Butler, M. and Rezazadeh, A. (2012) ***A Systematic Approach to Atomicity Decomposition in Event-B***. In, *SEFM 2012.*

- http://www.ecs.soton.ac.uk/people/mjb/publications

Code Generation from Event-B

A. Edmunds, A. Rezazadeh, M. Butler (2012). *Formal modelling for Ada implementations: Tasking 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

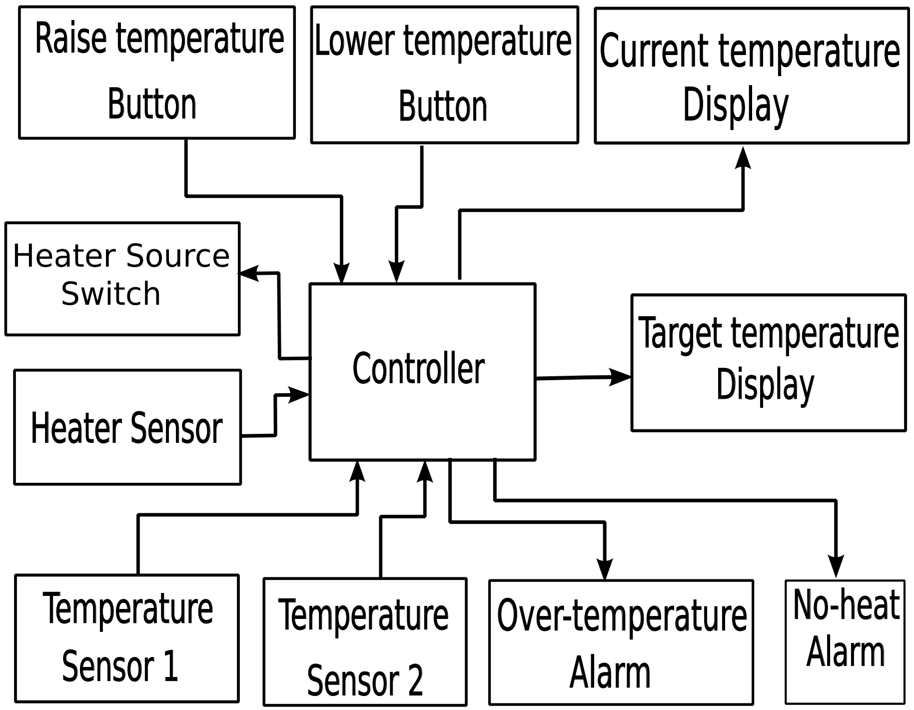
- Tasking Machine (Event-B machine +explicit control flow term)
  - system may have several parallel tasking machines
  - add structured control flow to machine: ;   / If / While
  - atomic steps in a task correspond to atomic events
- Environment Machine
  - Similar to tasking machine but only intended 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 case study

Heating Controller Block Diagram

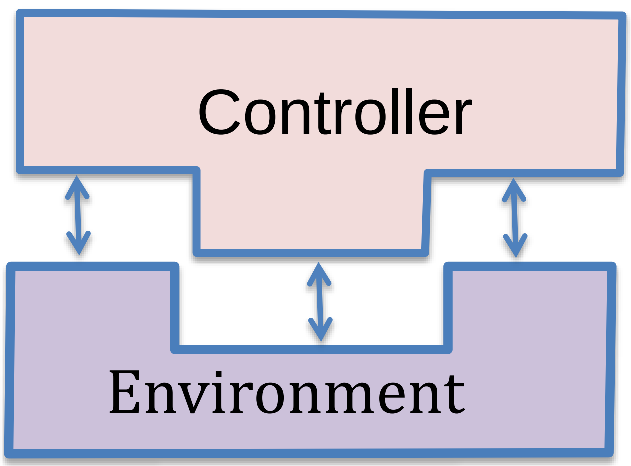


Main Functions

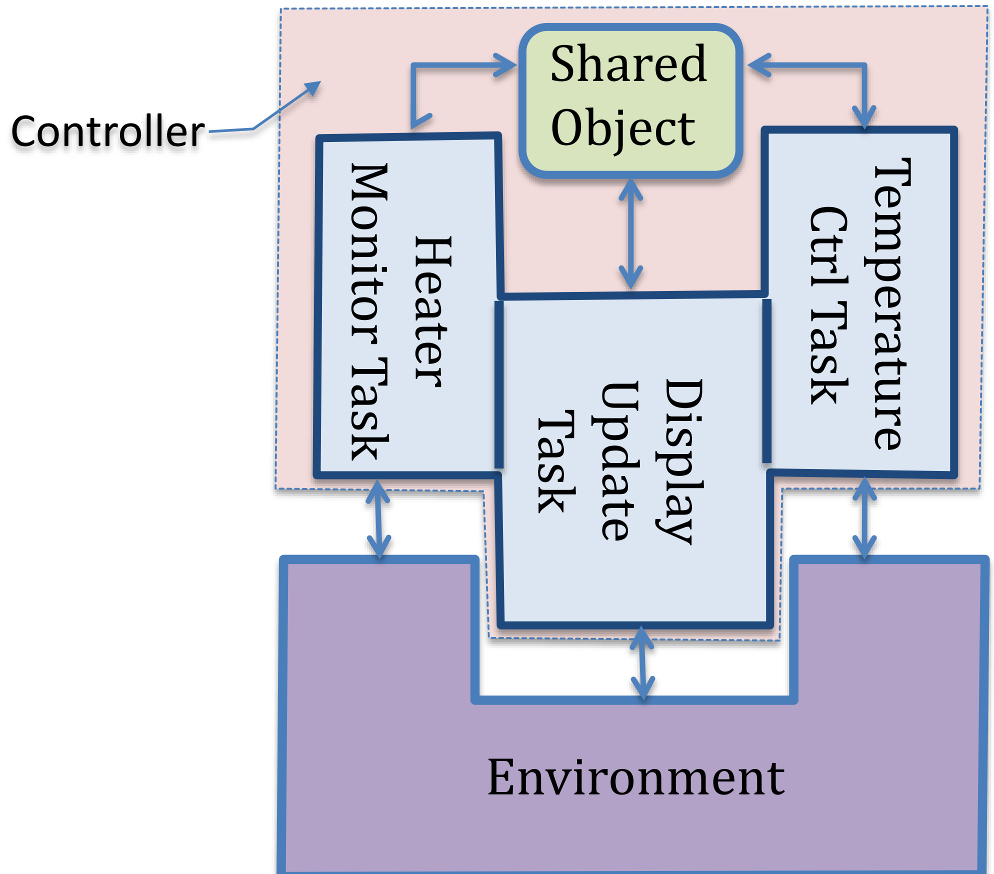
- Adjusting Target Temperature
- Sensing temperature
- Displaying current and target temperatures
- Activating/Deactivating Alarms
- Change target temperature
- Power on/off Heater
- Sensing heater status

Decomposition to tasks

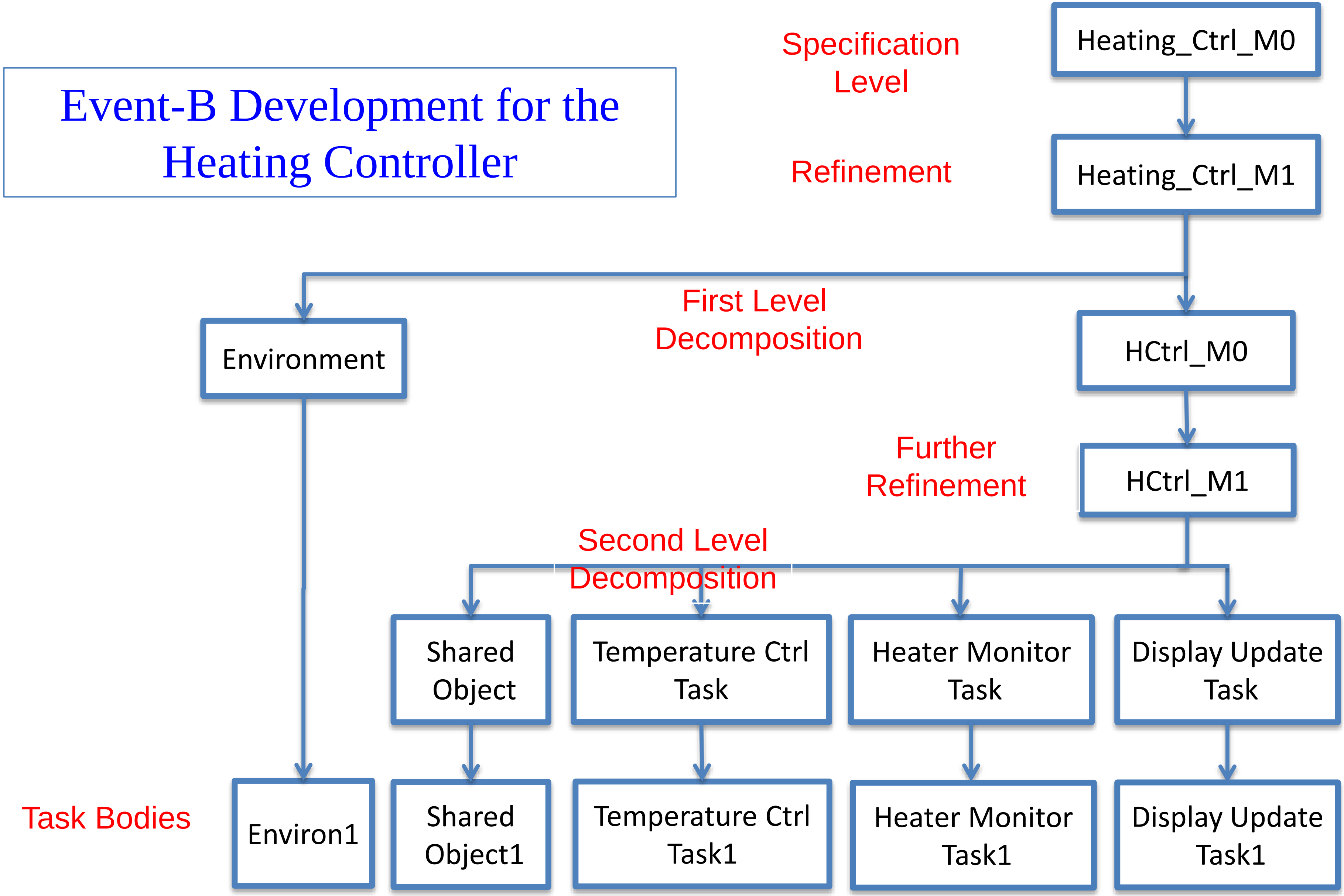
Decomposing the Controller from its Environment



Decomposition of the Controller into Tasks and a shared Object



Event-B Development for the Heating Controller



Not (yet) supporting…

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

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