UNIVERSITY OF Southampton

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

deploy

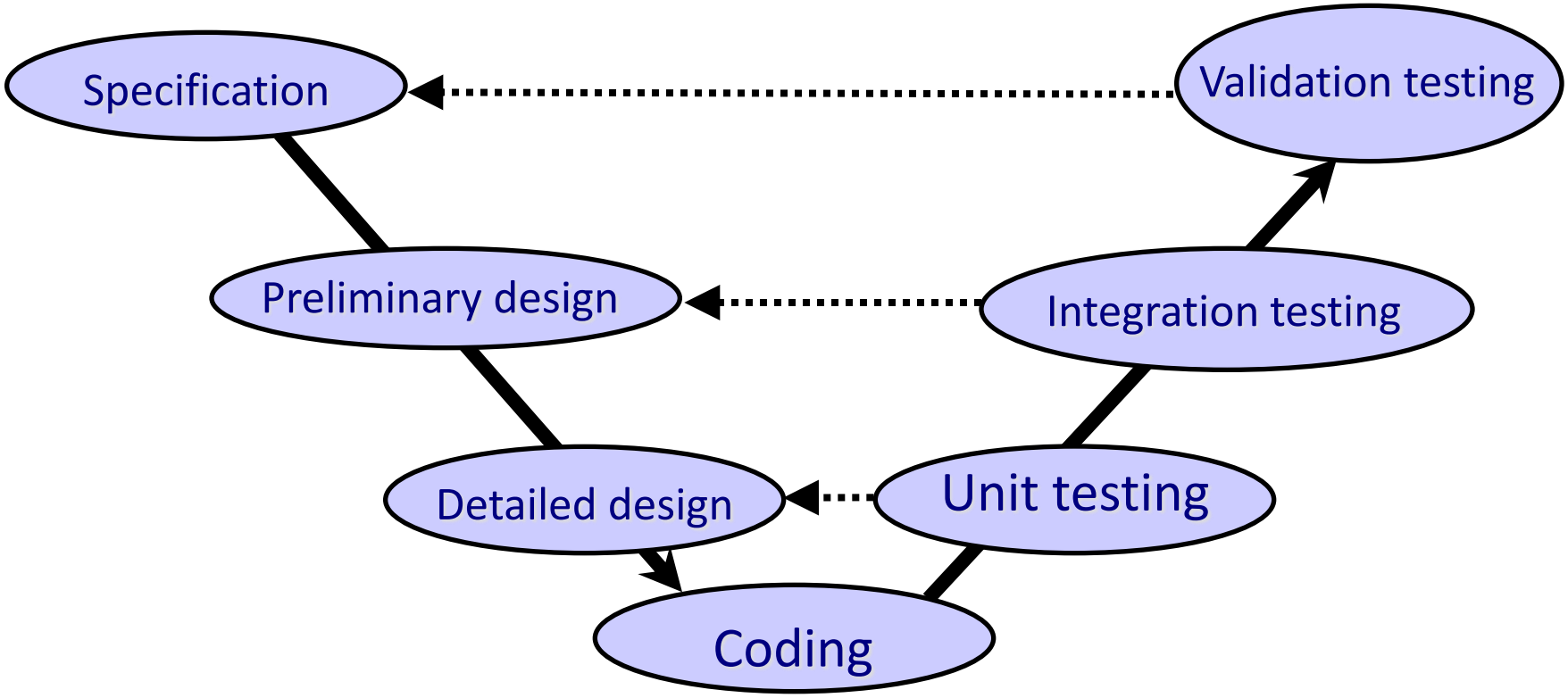
Contents

- 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

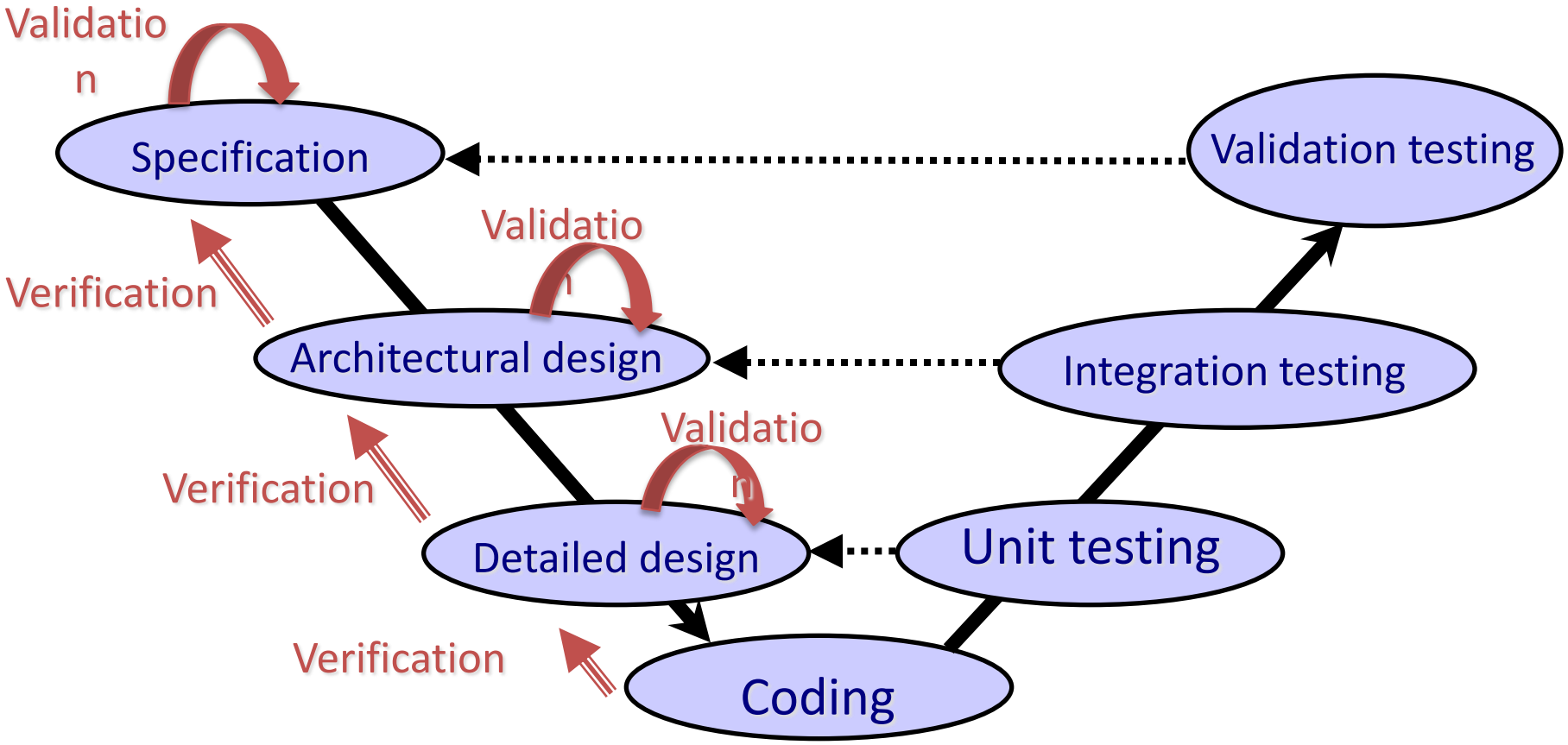
- Mathematical techniques for formulation and analysis of systems

- 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



Rapid prototyping*versus* modelling

- Rapid prototying: 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

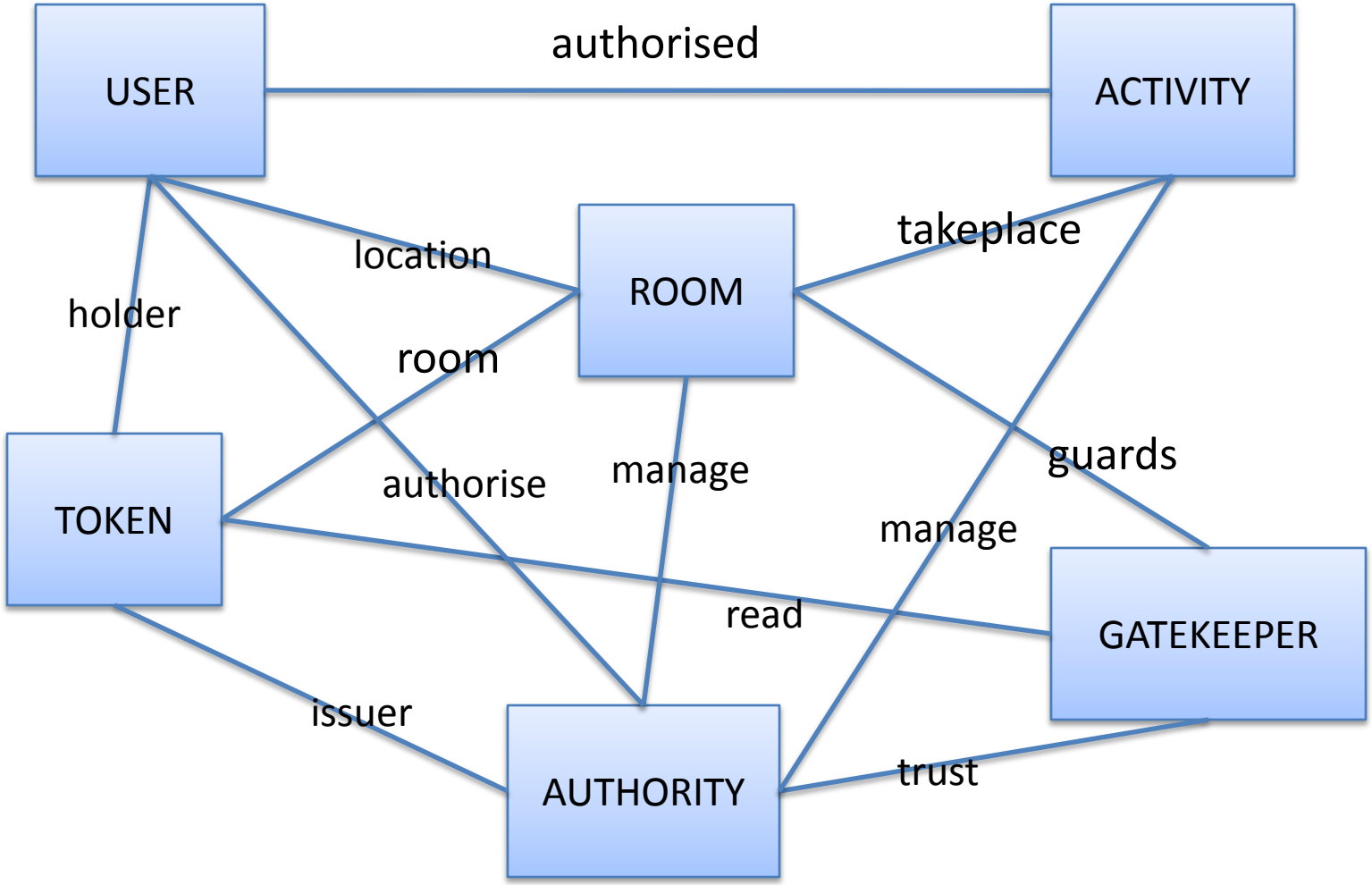
- Example: access control system

- Example intended to give a feeling for:
  - modelling language
  - abstraction and refinement
  - role of verification

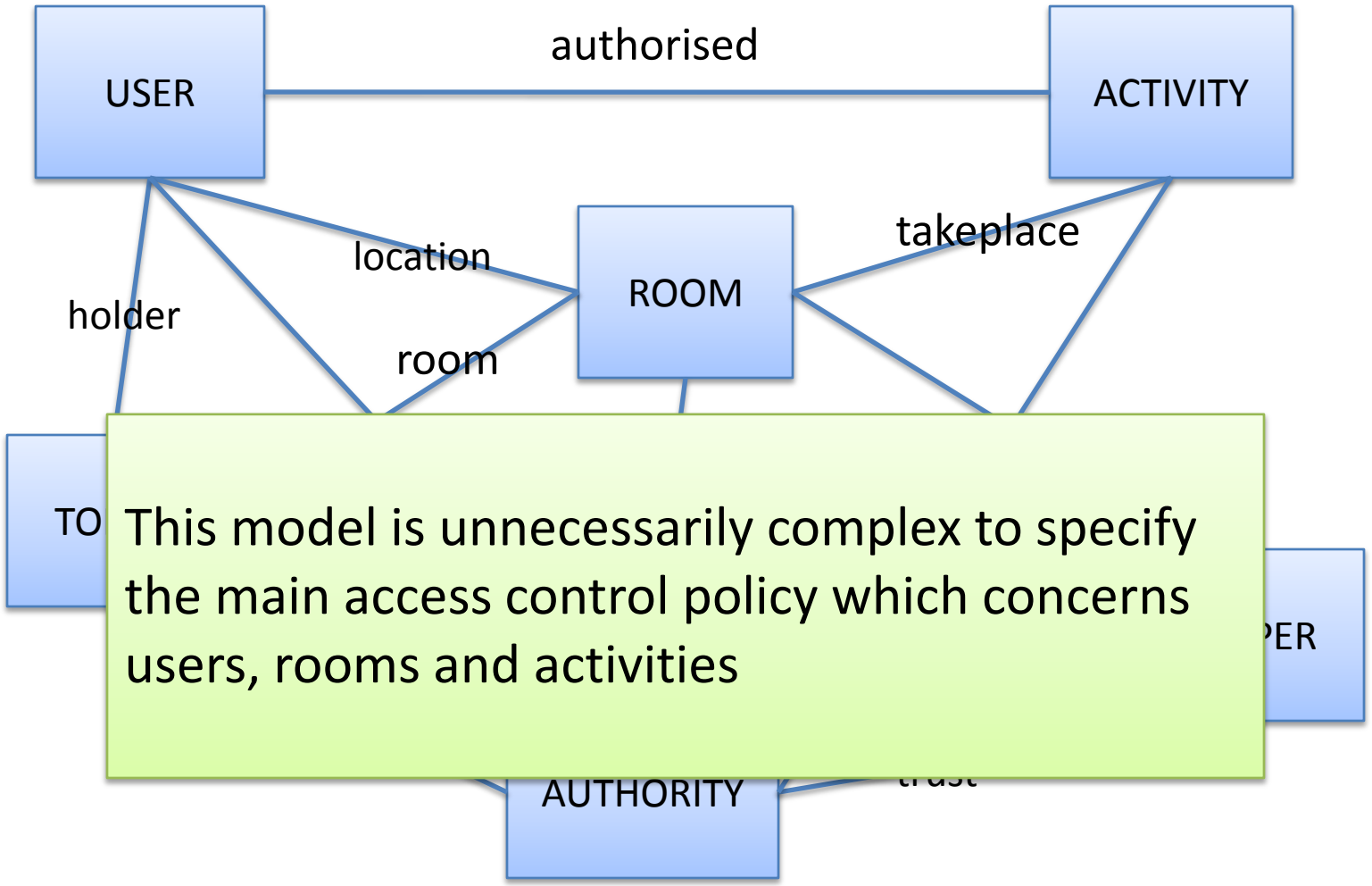
Access control system

- 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

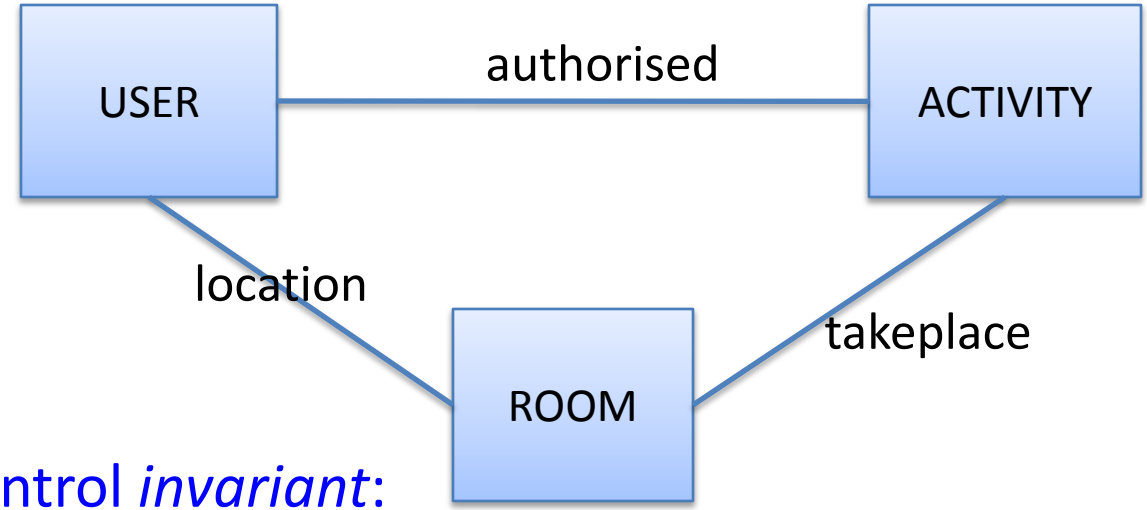
Entity-relationship diagram



Entity-relationship diagram



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

$$location(u) = r \Rightarrow takeplace[r] \subseteq authorised[u]$$

Abstraction: focus on key entities in the problem domain

Enter a room

Enter $\triangleq$

when

grd1 : $u \in User$

grd2 : $r \in Room$

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

then

act1 : $location(u) := r$

end

Does this operation maintain the security invariant?

Remove authorisation

RemoveAuth(u,a) $\triangleq$

when

grd1 : $u \in User$

grd2 : $a \in Activity$

grd3 : $u \mapsto a \in authorised$

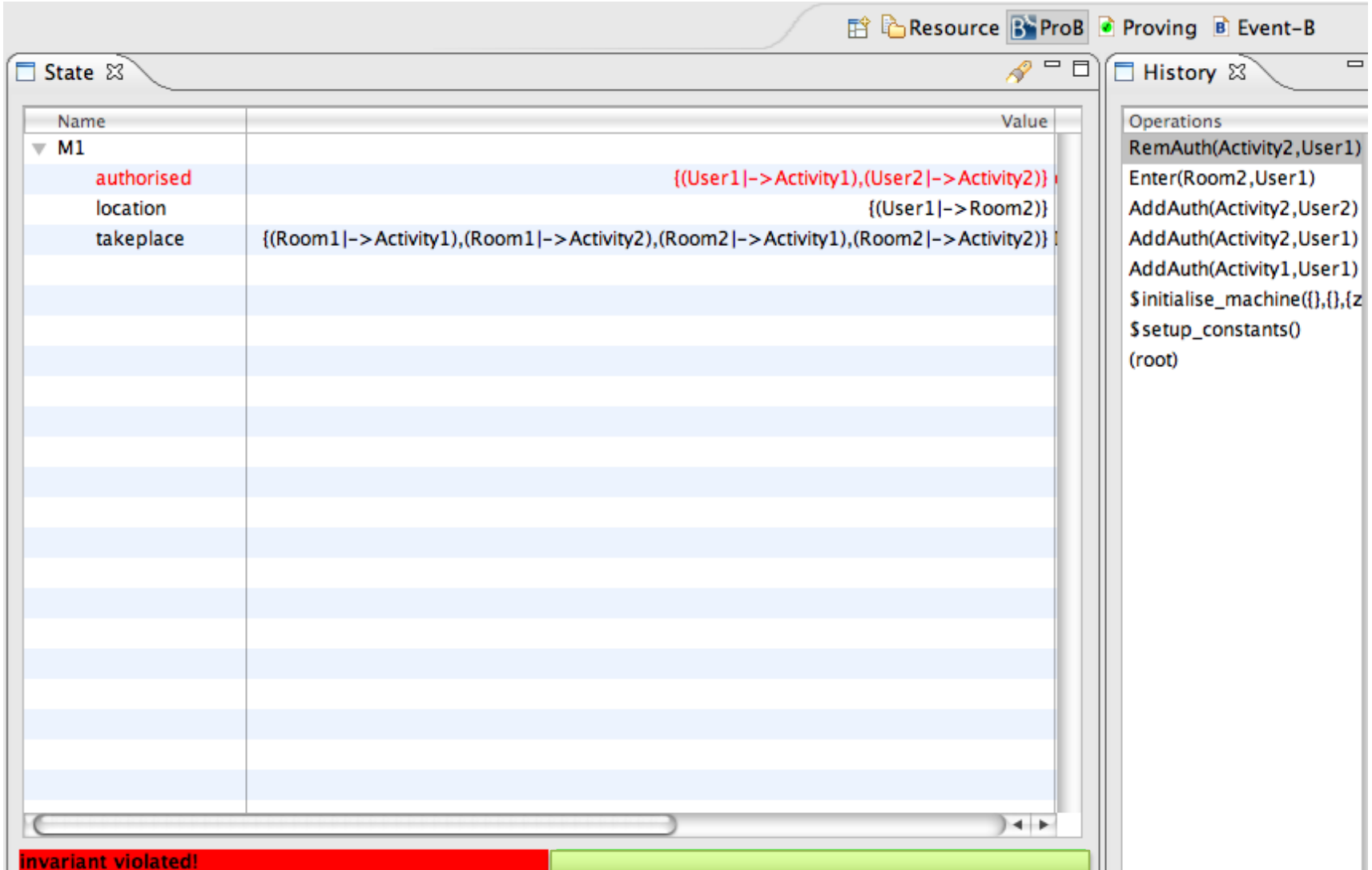
then

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

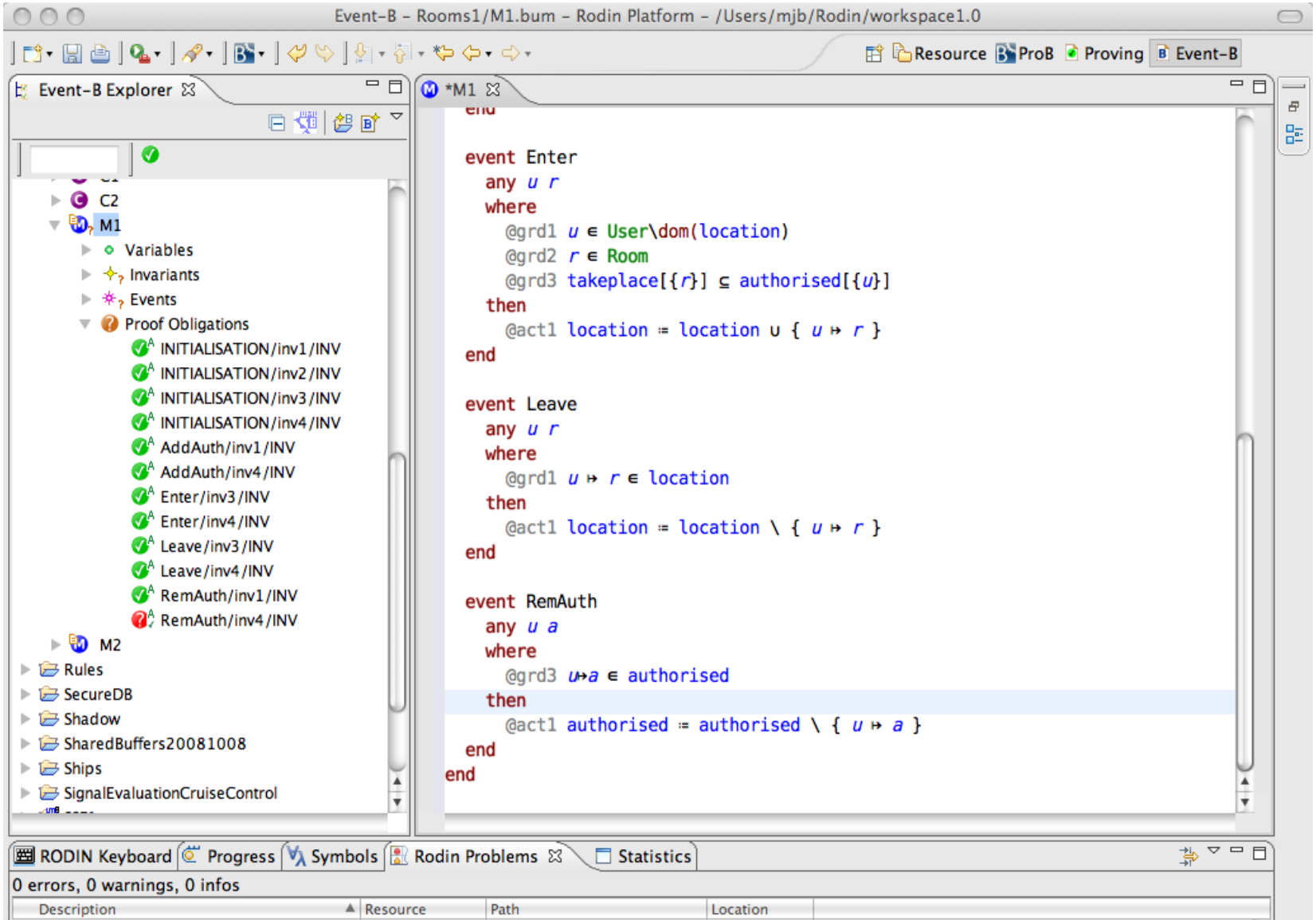
end

Does this operation maintain the security invariant?

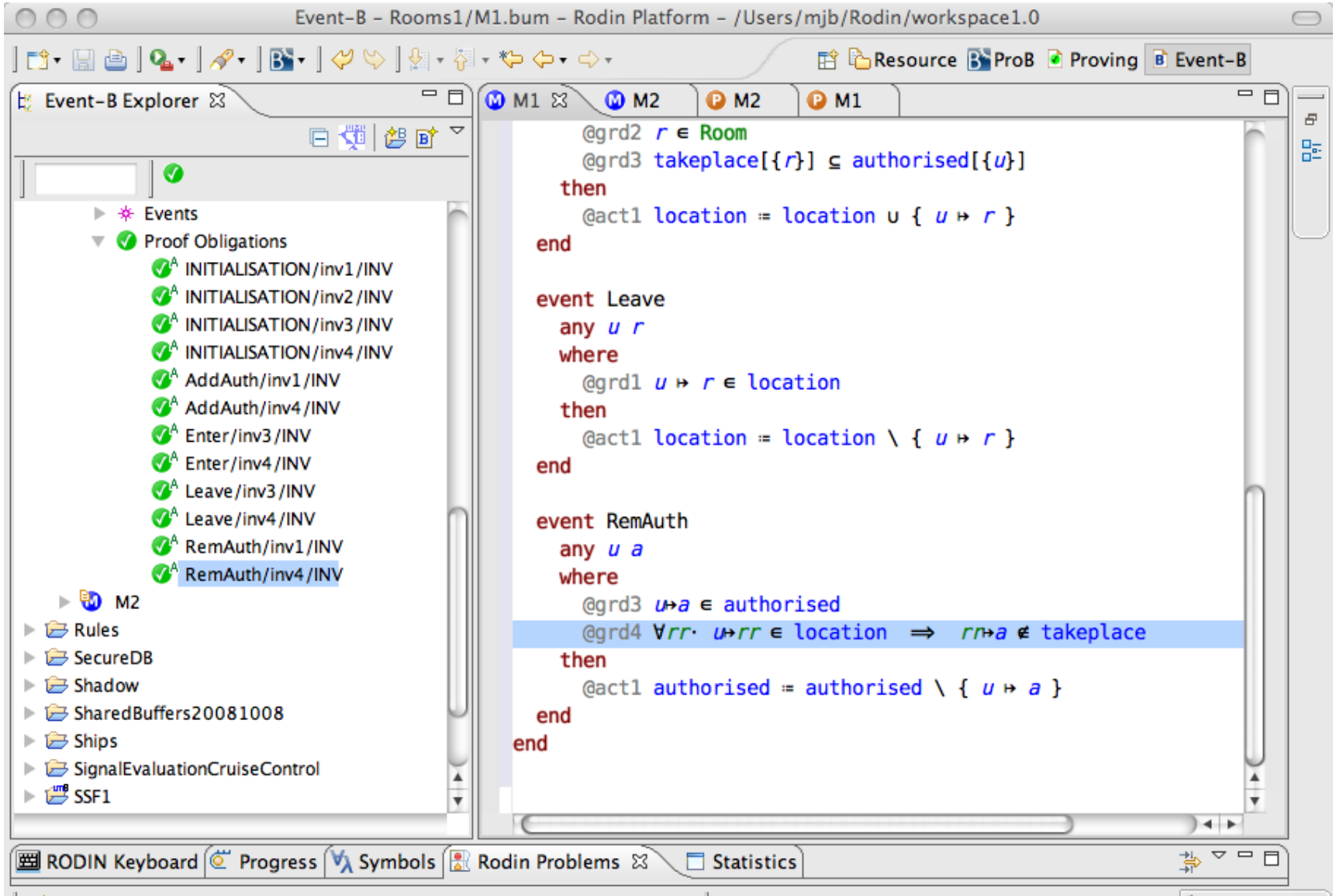
Counterexample from model checking with ProB plug-in for Rodin



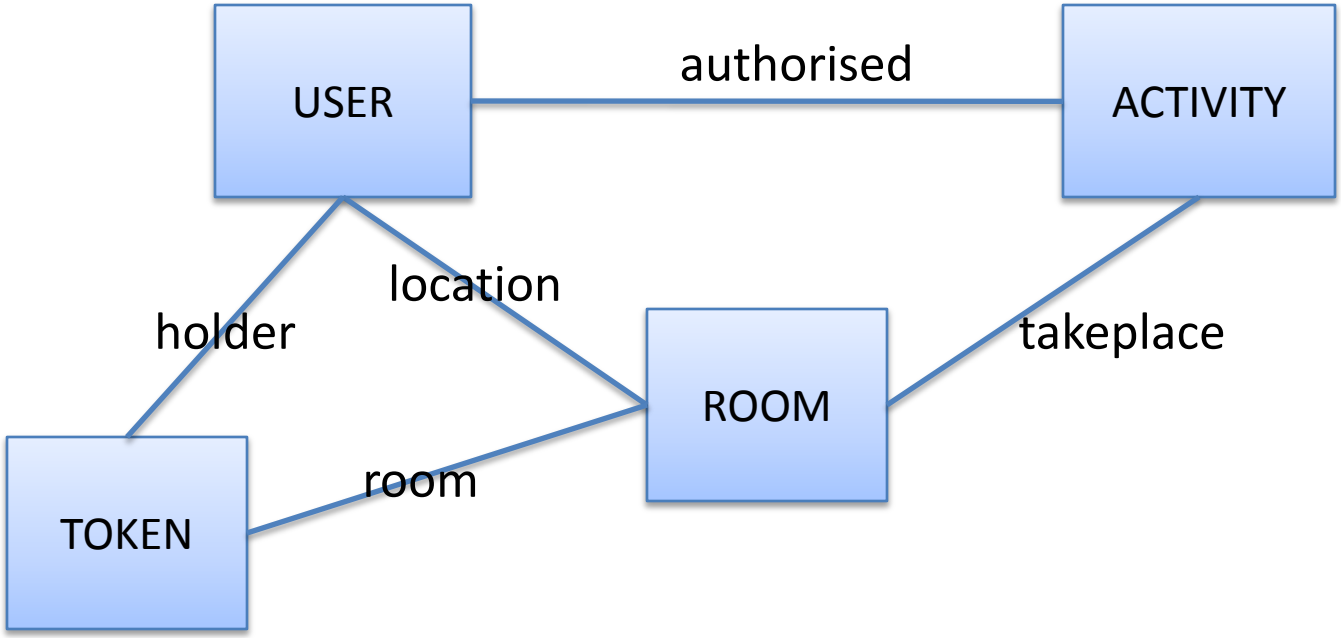
Failing proof with Rodin



Strengthen guard of *RemAuth*



Now refine



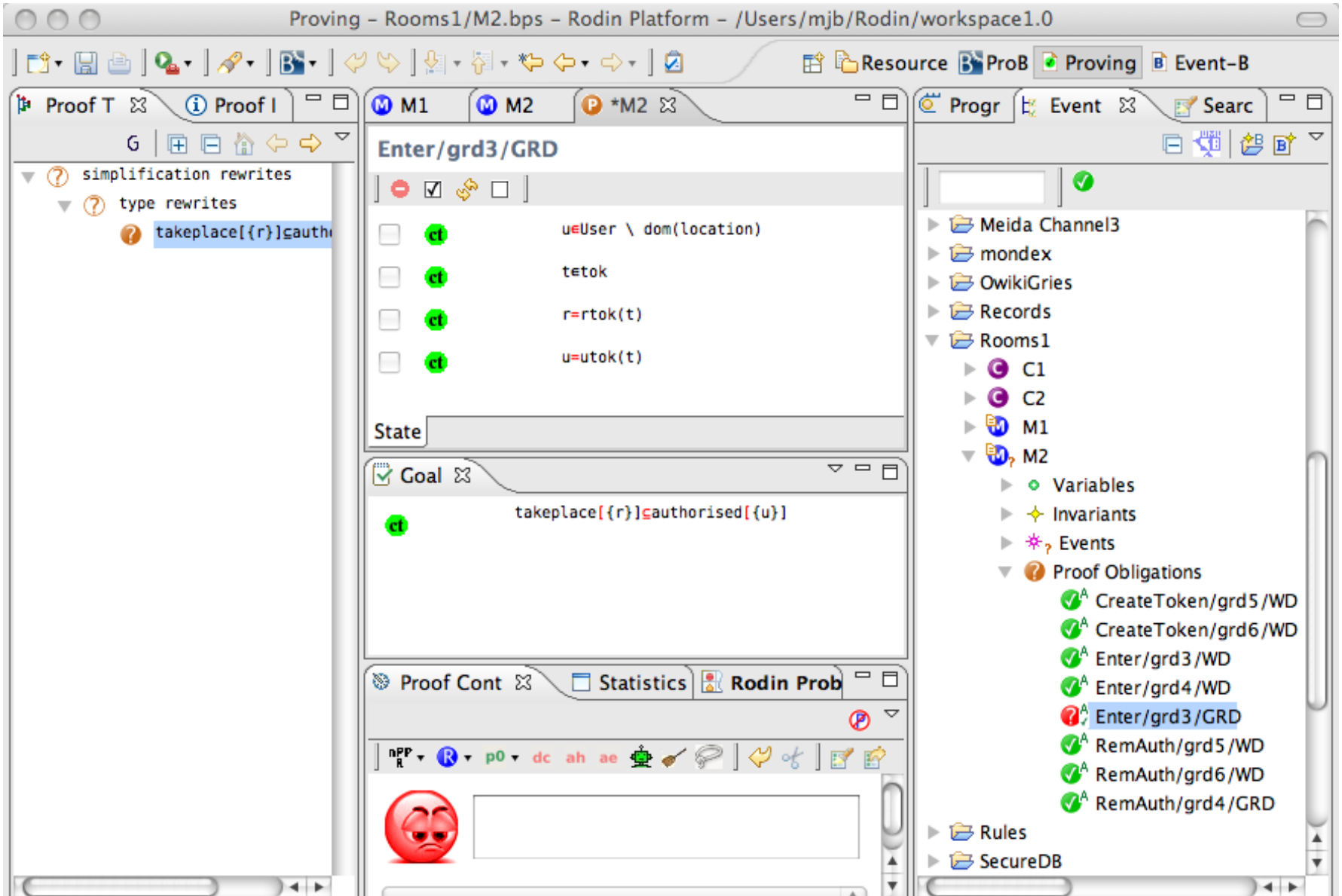
Abstract condition on a user and room for entering

takeplace[r] ⊆ authorised[u]

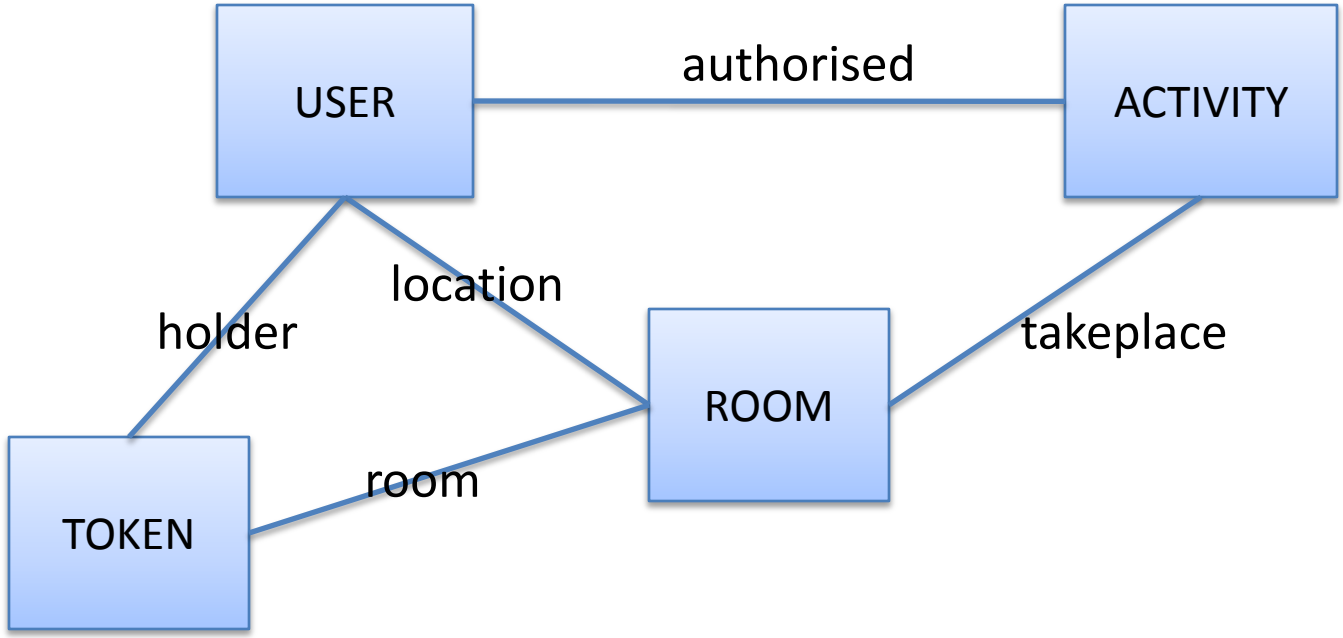
is replaced by a condition on a token

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

Failing refinement proof



Gluing invariant



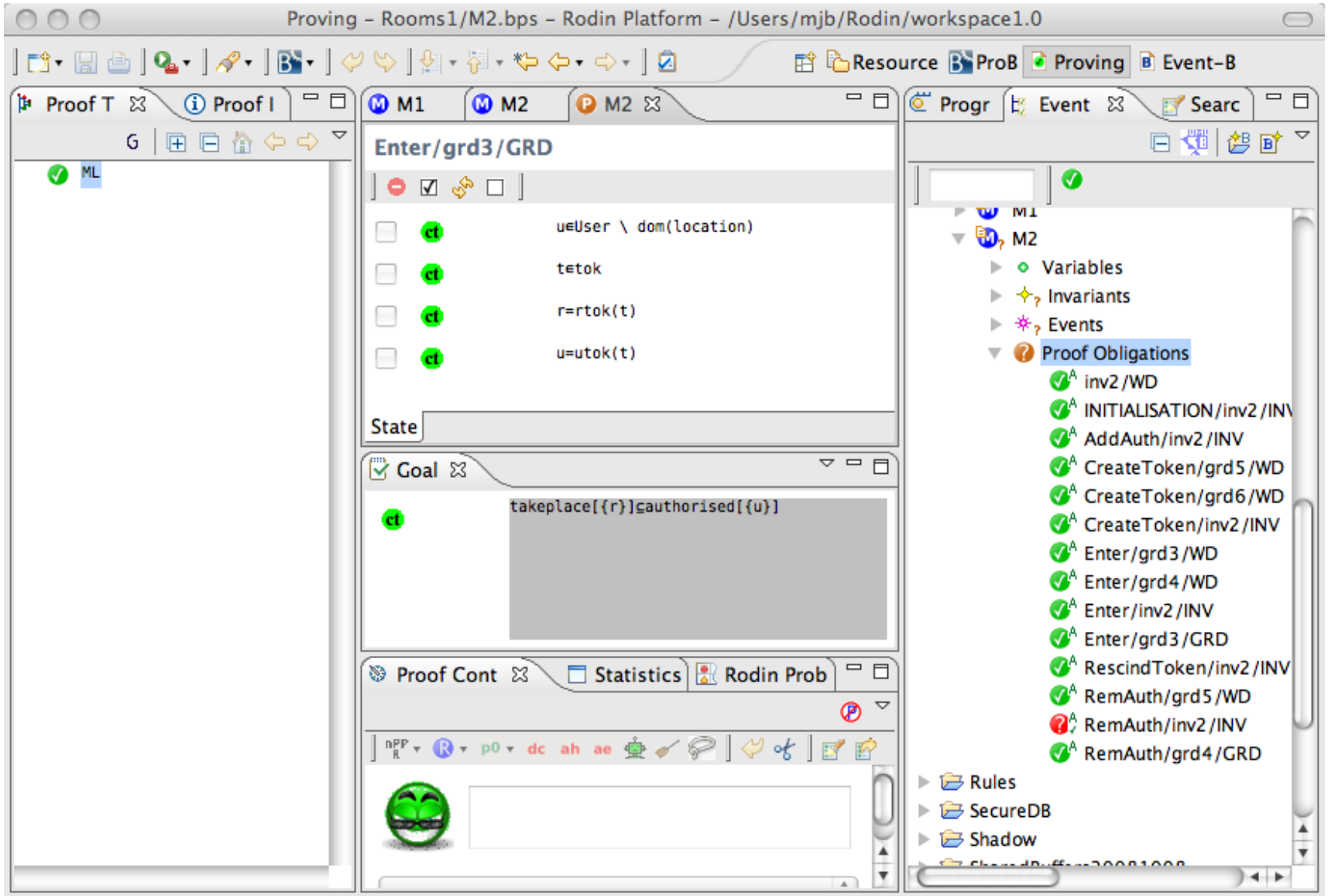
To ensure consistency of the refinement we need invariant:

$t \in valid$

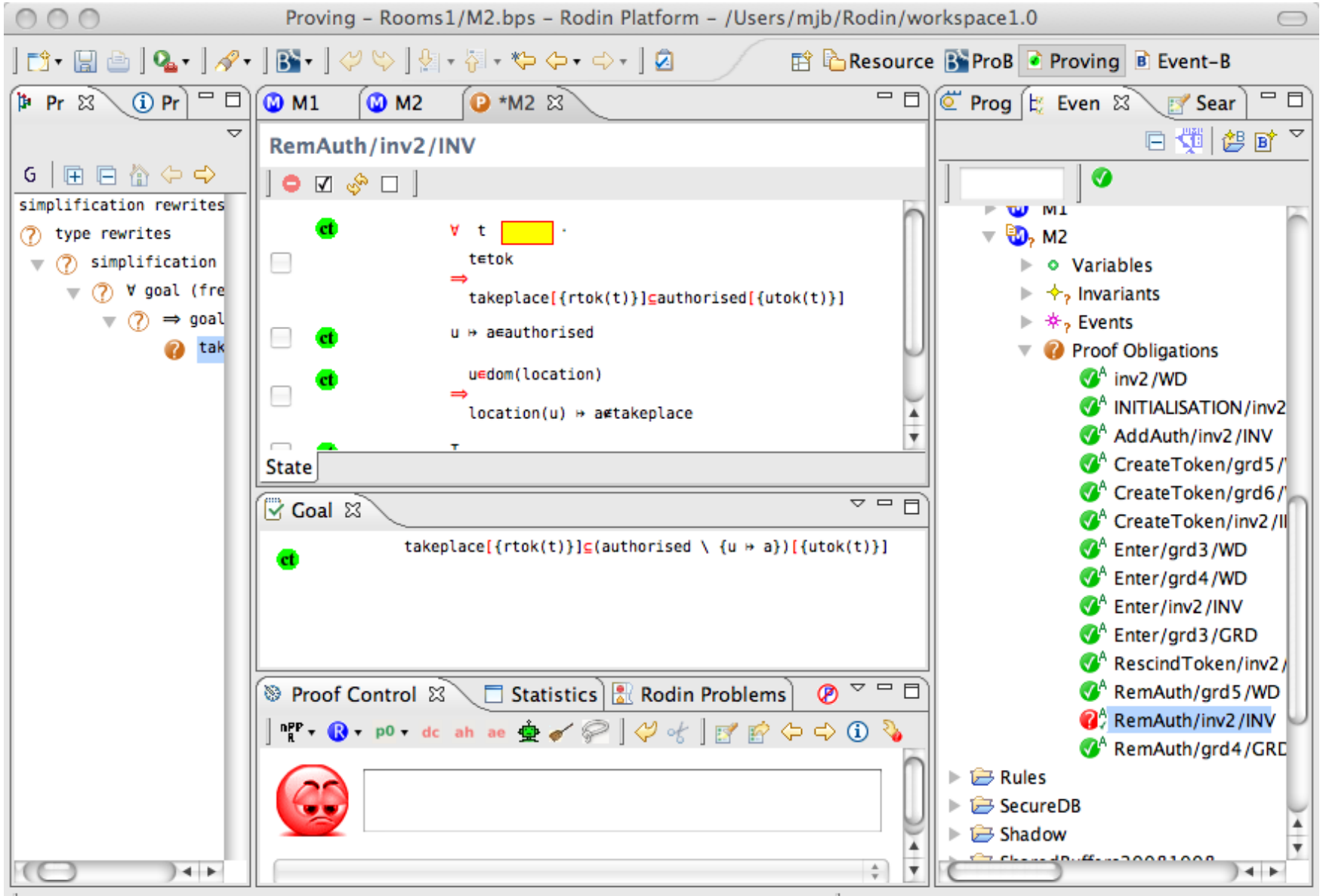
$\Rightarrow$

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

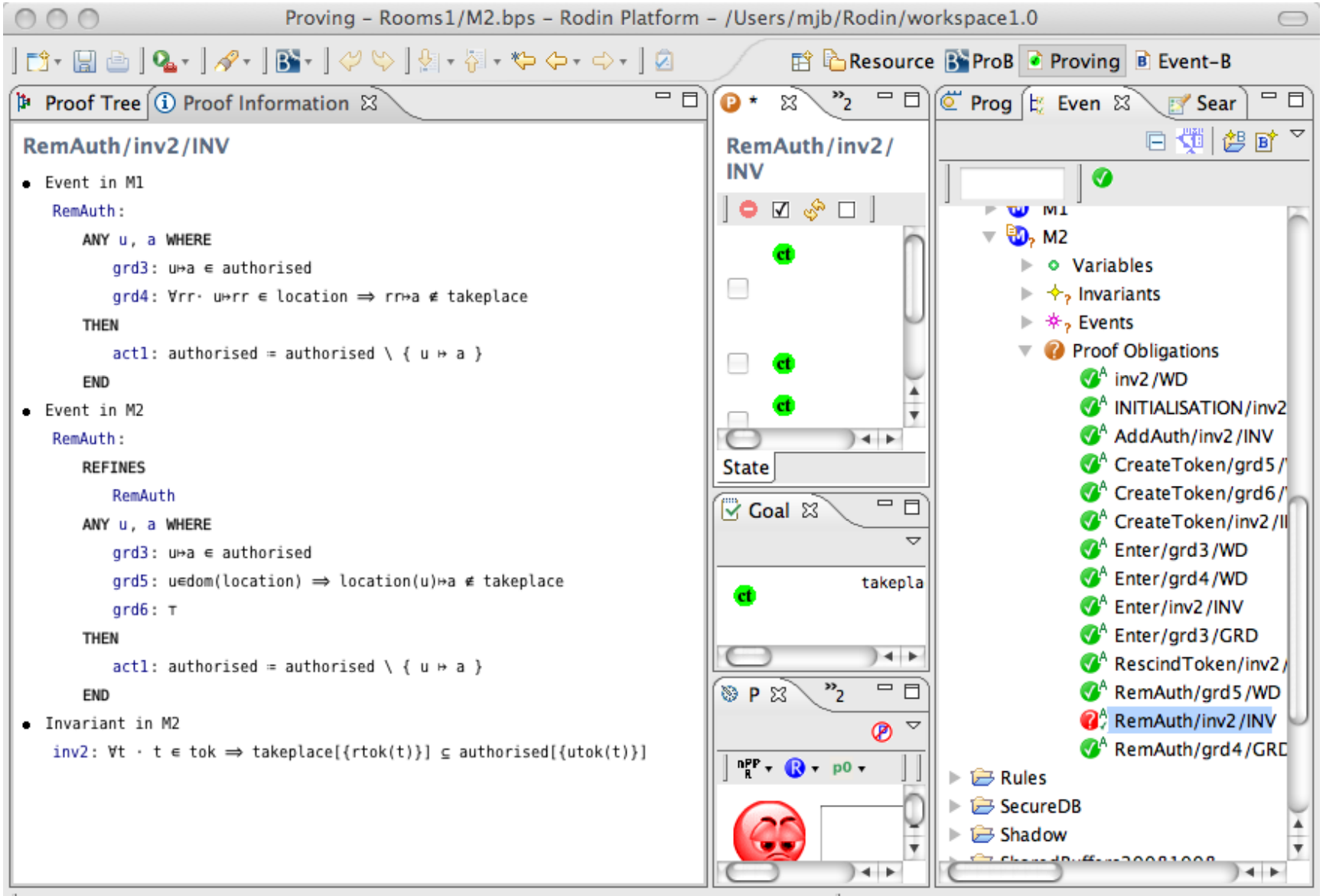
Invariant enables PO discharge



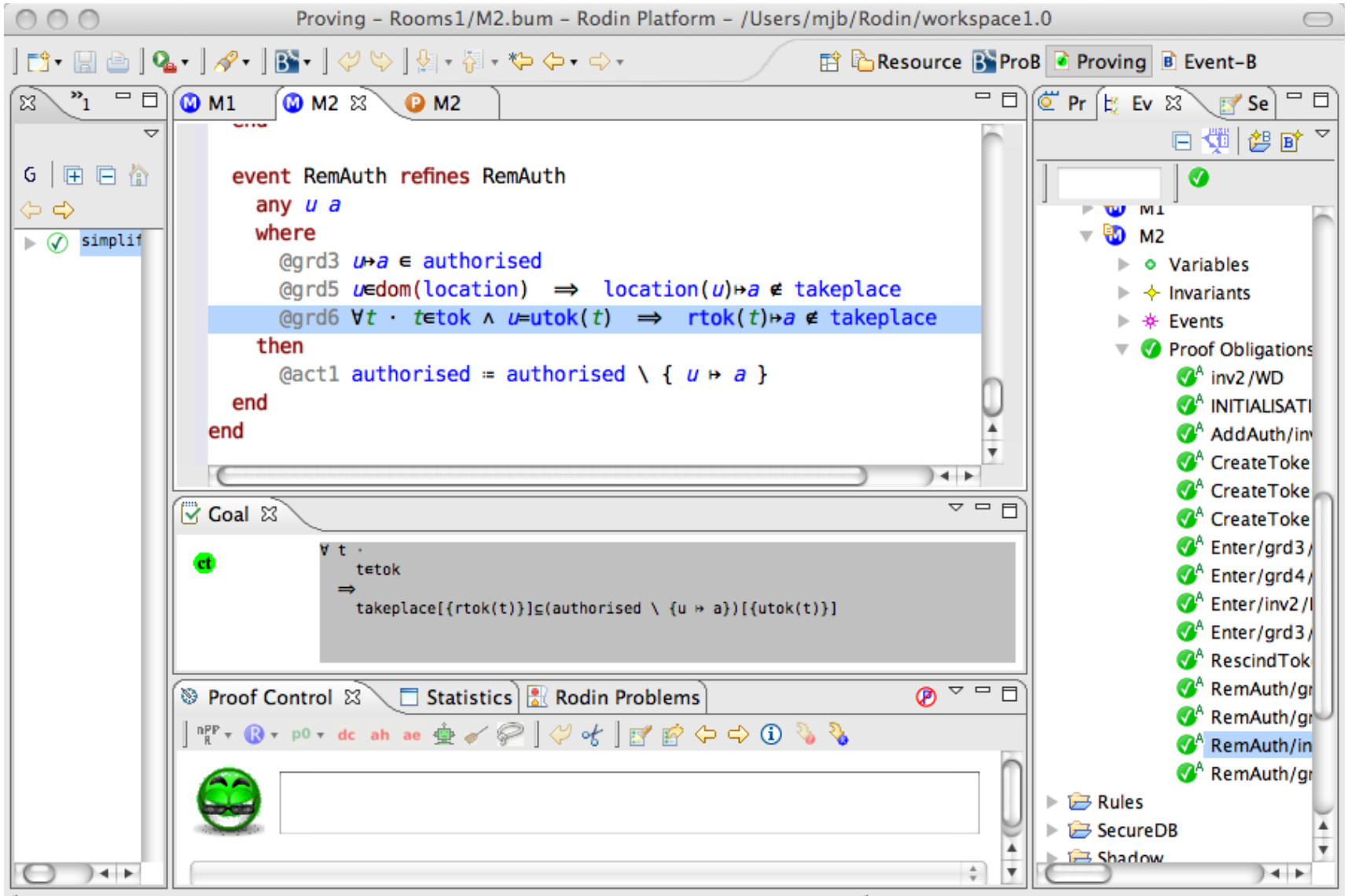
But get new failing PO



Source of failing PO



Strengthen guard of refined *RemAuth*



Rational design – what, how, why

- *What* does it achieve?

  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 that room

What, how, why written in B

- *What* does it achieve?

  location(u) = r

  ⇒ takeplace[r] ⊆ authorised[u]

- *How* does it work?

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

- *Why* does it work?

  t ∈ valid

  ⇒

  takeplace [room(t)] ⊆ authorised[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 ⇸ DATA

machine filestore
variables file, dsk
invariant
file ⊆FILE ∧
dsk∈ file → CONT

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

events

CreateFile≙…

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

ReadFile ≙ // return data in page *p* of *f*
anyf, p, d! **where**
f∈ file
p∈ dom(dsk(f))
d! = dsk(f)(p)
end

Refinement of file store

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

machinefilestore2
refinesfilestore
variables file,dsk,writing,wbuf, sdsk

invariant

writing $\subseteq$ file
wbuf$\in$ writing $\rightarrow$CONT
sdsk$\in$ writing $\rightarrow$ CONT // shadow disk

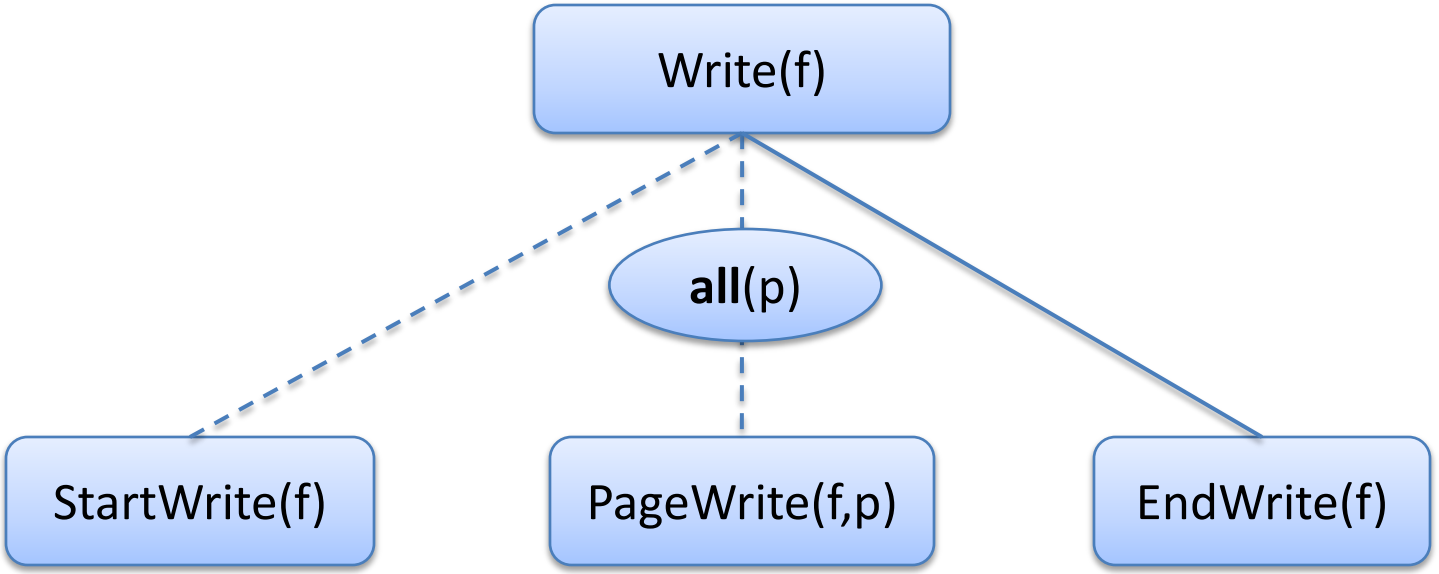
Breaking atomicity

- 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

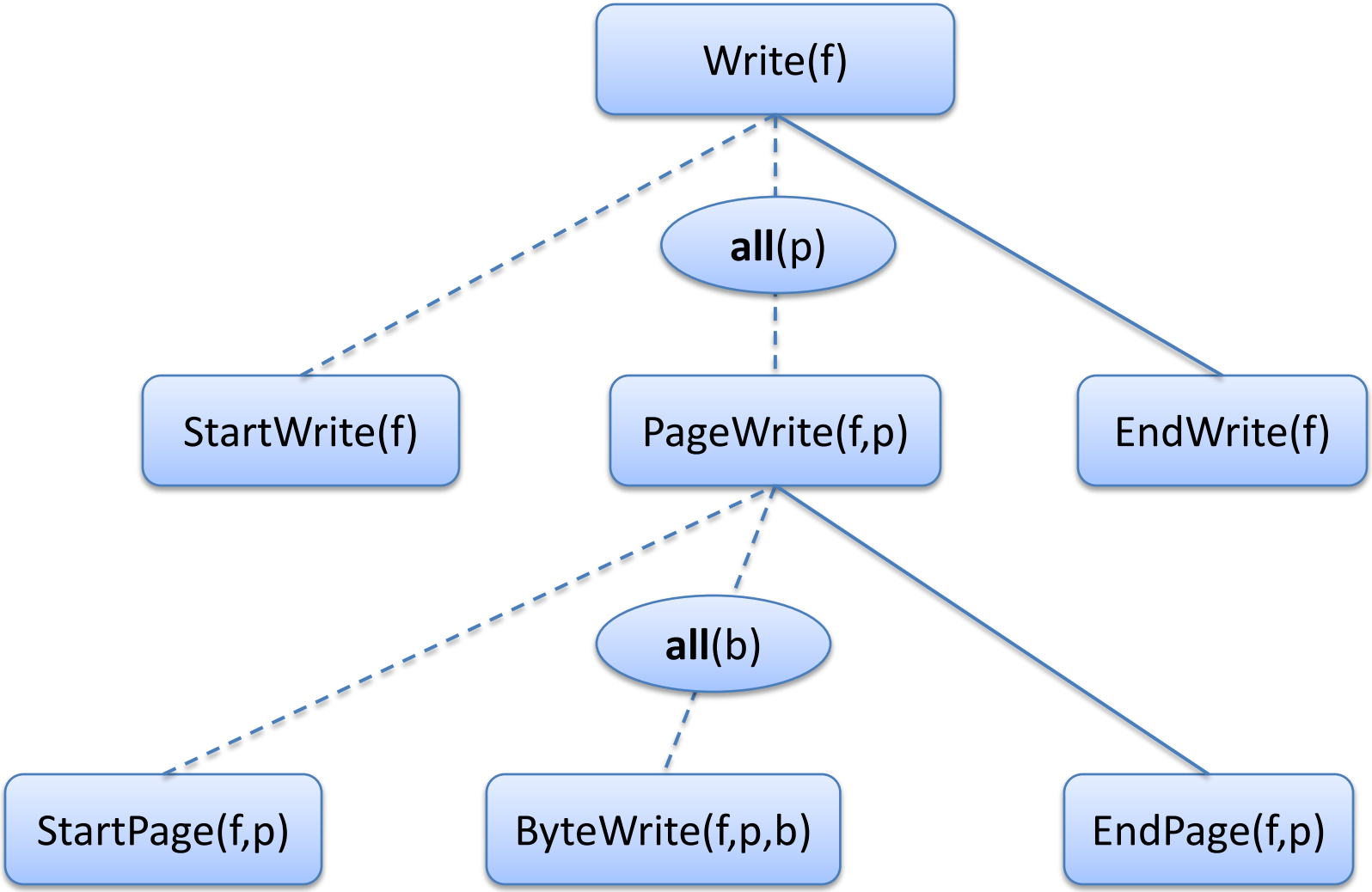
- Based on diagrammatic notation of *Jackson System Development* (JSD)
- Graphical representation of how abstract atomic events are refined
- We can exploit the hierarchical nature of JSD diagrams to represent event refinement
- Adapt JSD notation for our needs

Event refinement diagram



- Diagram represents atomicity refinement explicitly *and*
- Diagram specifies sequencing constraints on events

Hierarchical refinement



Replicated data base

- Abstract model

  db $\in$ object $\rightarrow$ DATA

Commit = /* update a set of objects *os* */
any os, update
where
os $\subseteq$ object $\wedge$
update $\in$ (os $\rightarrow$ DATA) $\rightarrow$ (os $\rightarrow$ DATA)
then
db := db **<+** update(os $\lhd$ db)
end

Refinement by replicated database

$sdb \in site \rightarrow (object \rightarrow DATA)$

Update is by two phase commit:

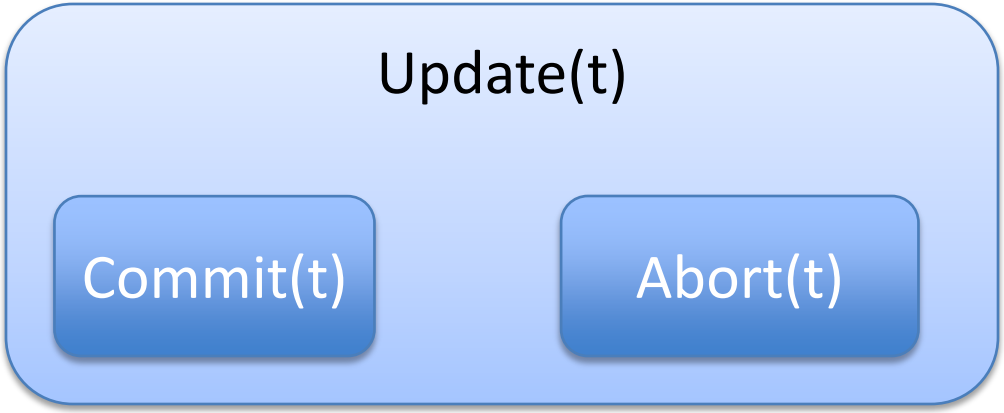
PreCommitfollowed by Commit

Global commit if all sites*pre-commit*

Global abort if at least one site aborts

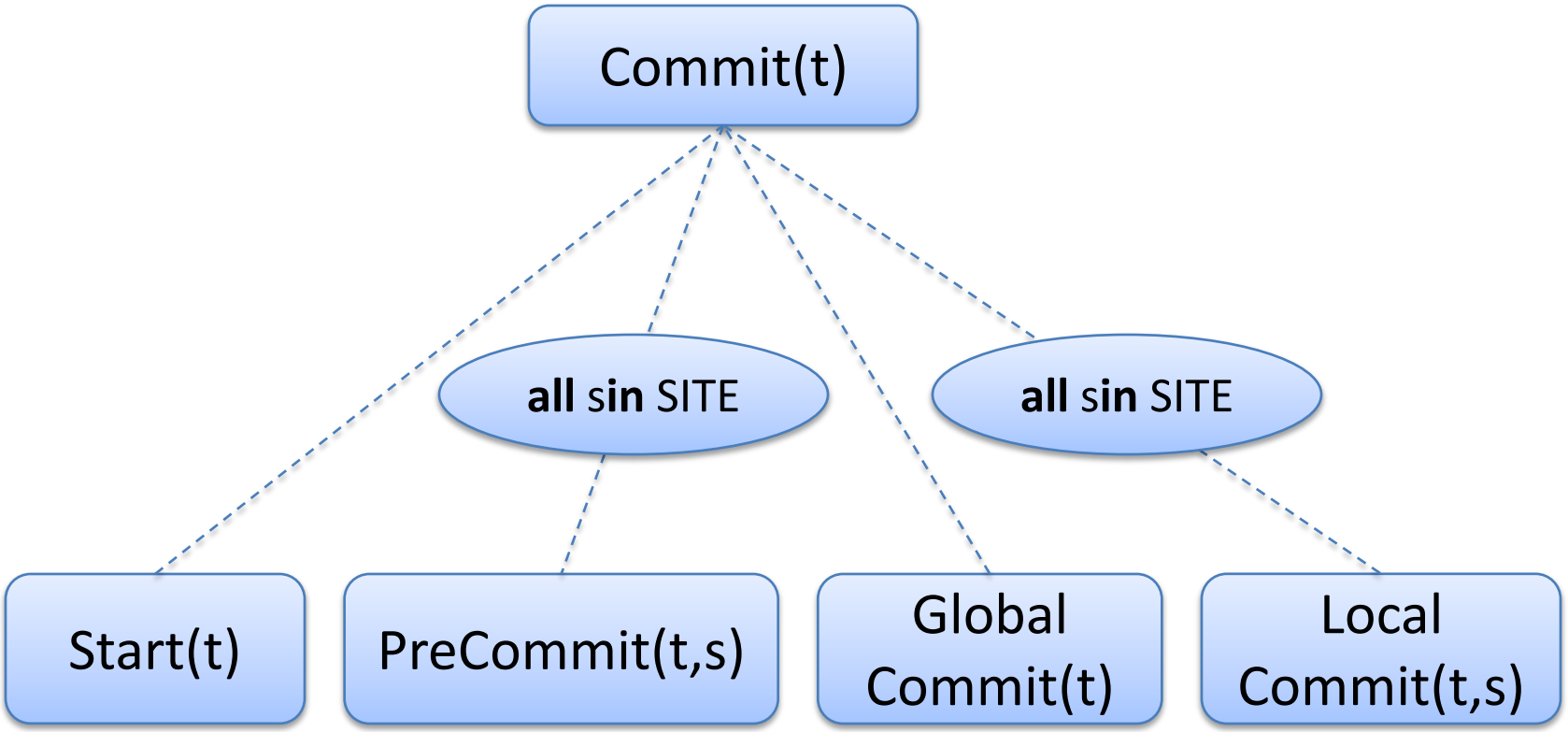
Mutual Exclusion

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



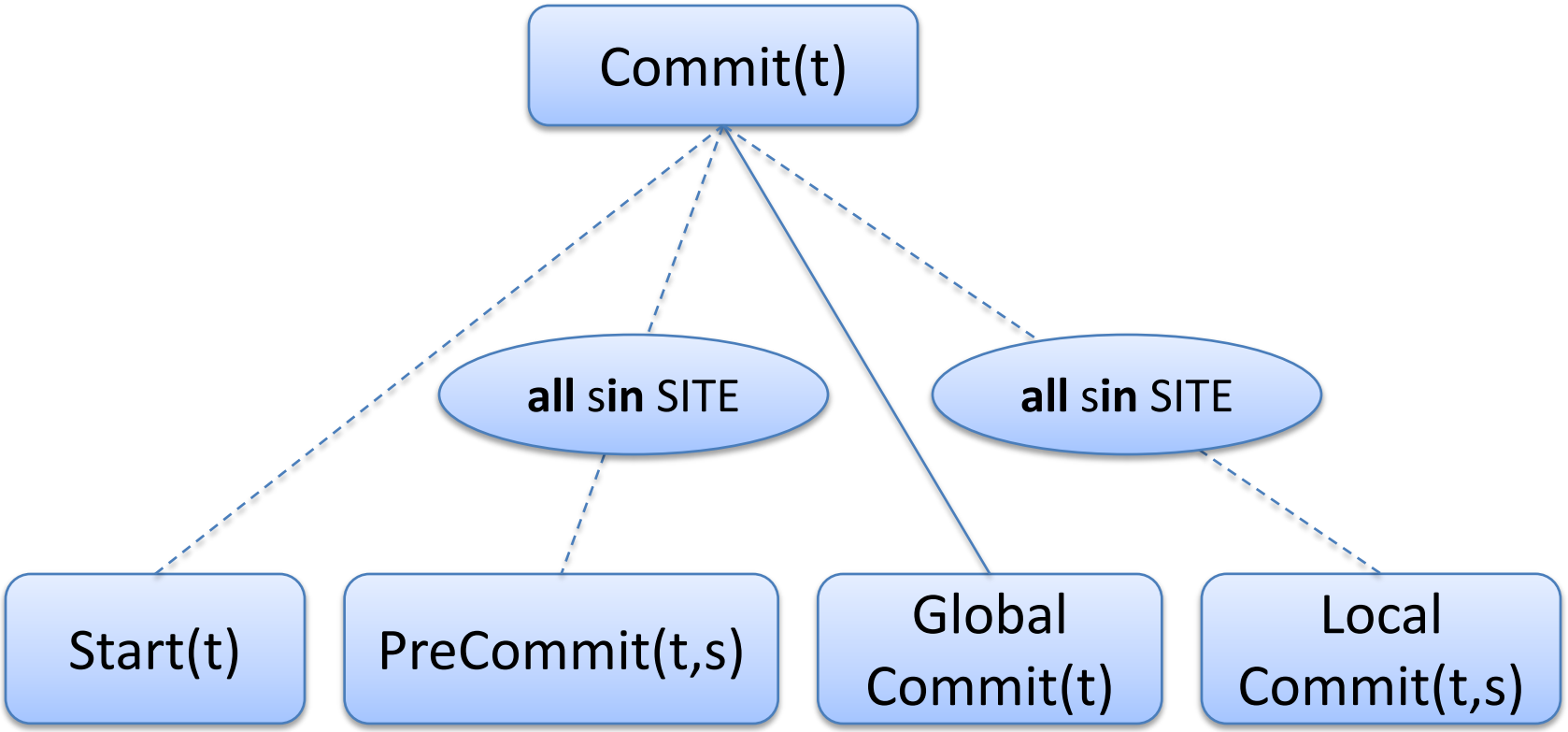
Update transaction will commit or abort *but not both*

Event refinement diagram for Commit



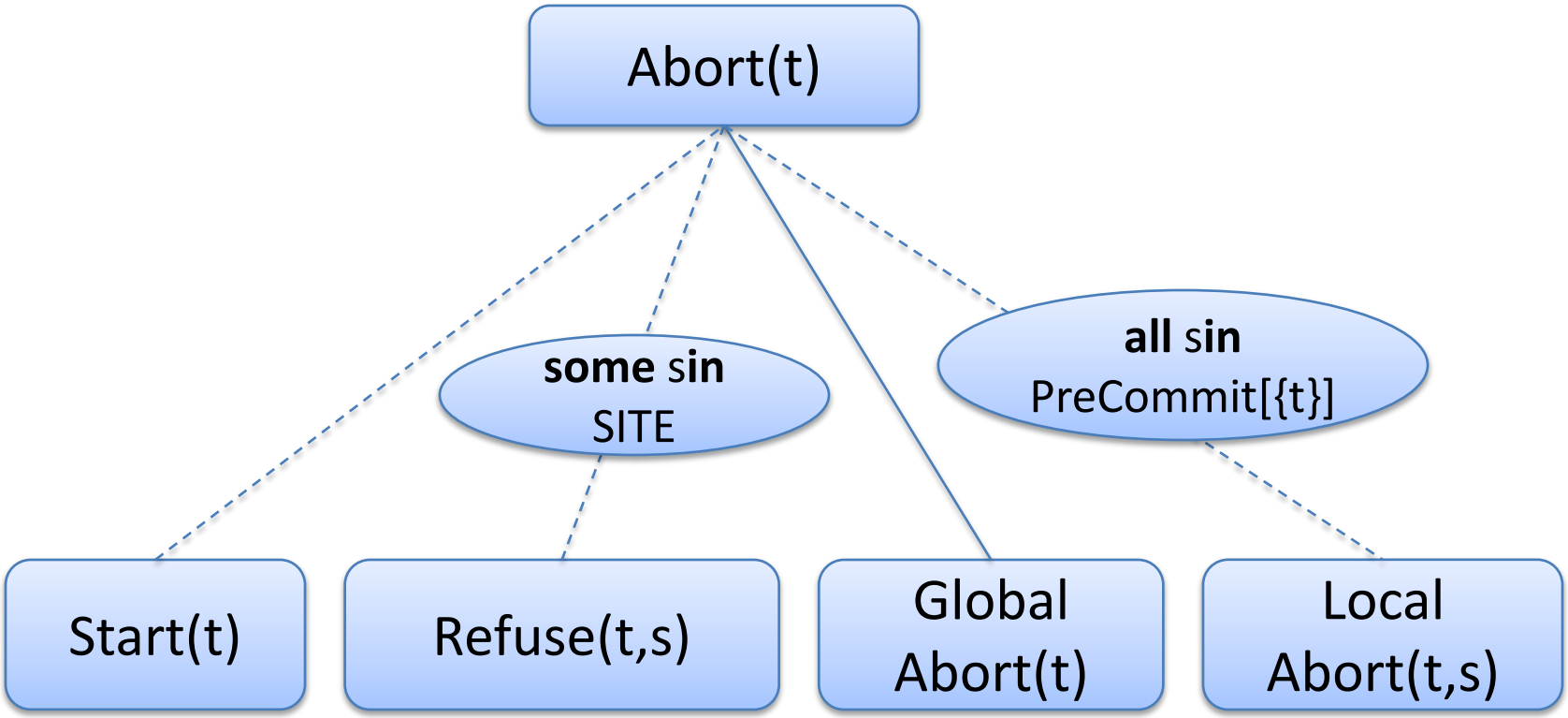
Which event refines the abstract *Commit*?

Event refinement diagram for Commit



Decision to proceed is made by *GlobalCommit*

Event refinement diagram for Abort



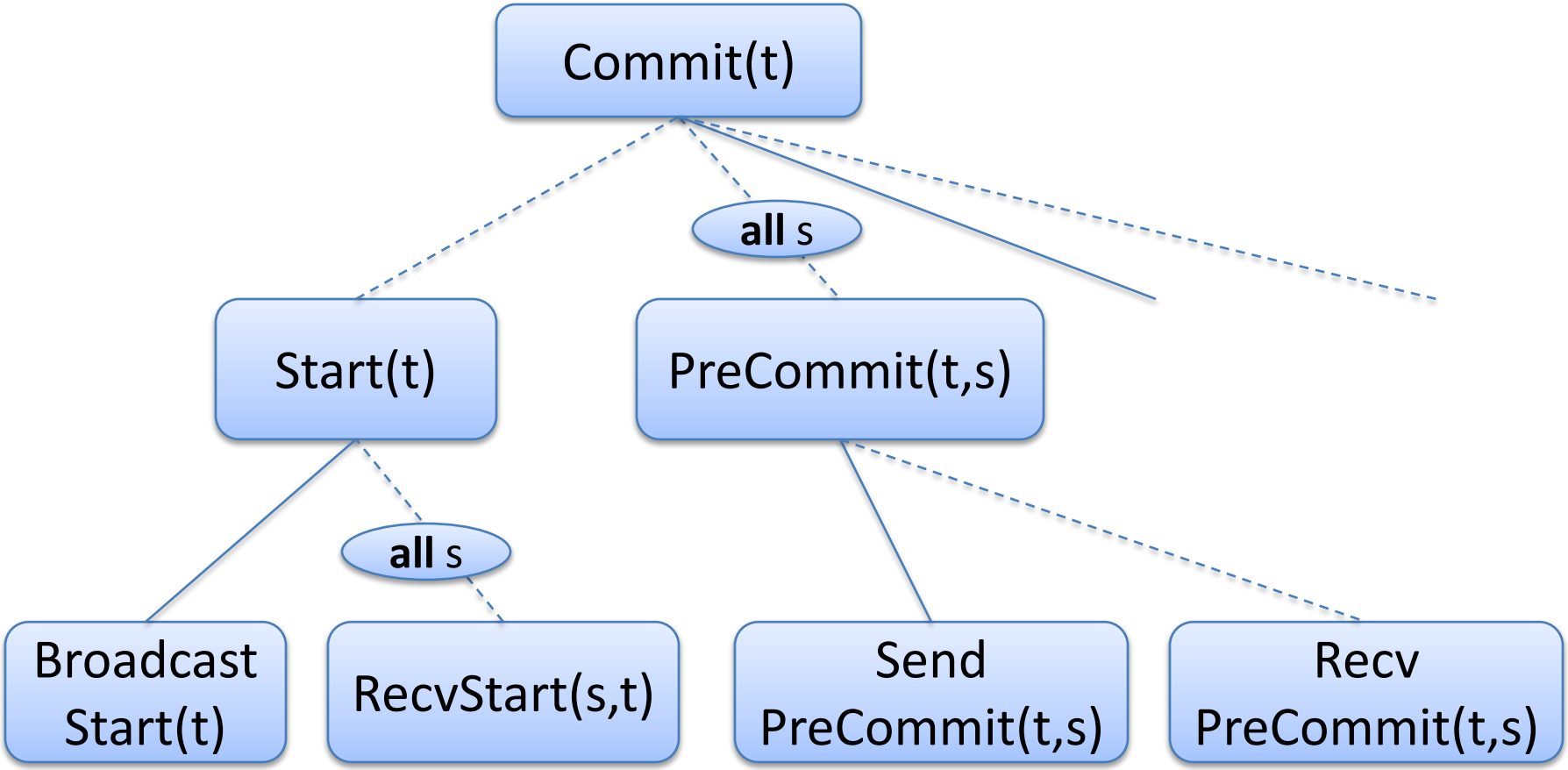
Protocol aborts transaction if some site aborts

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



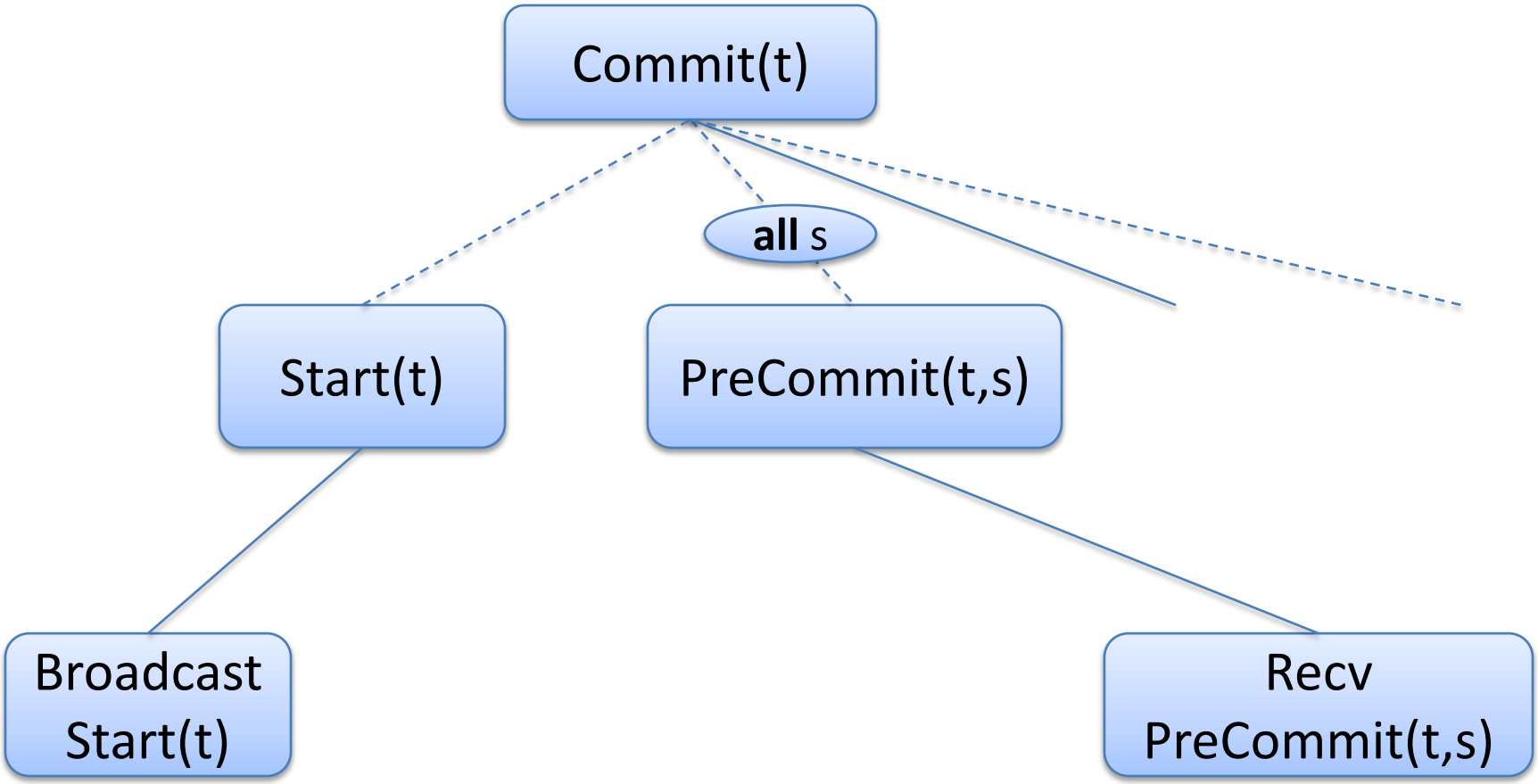
Where are we going?

- 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

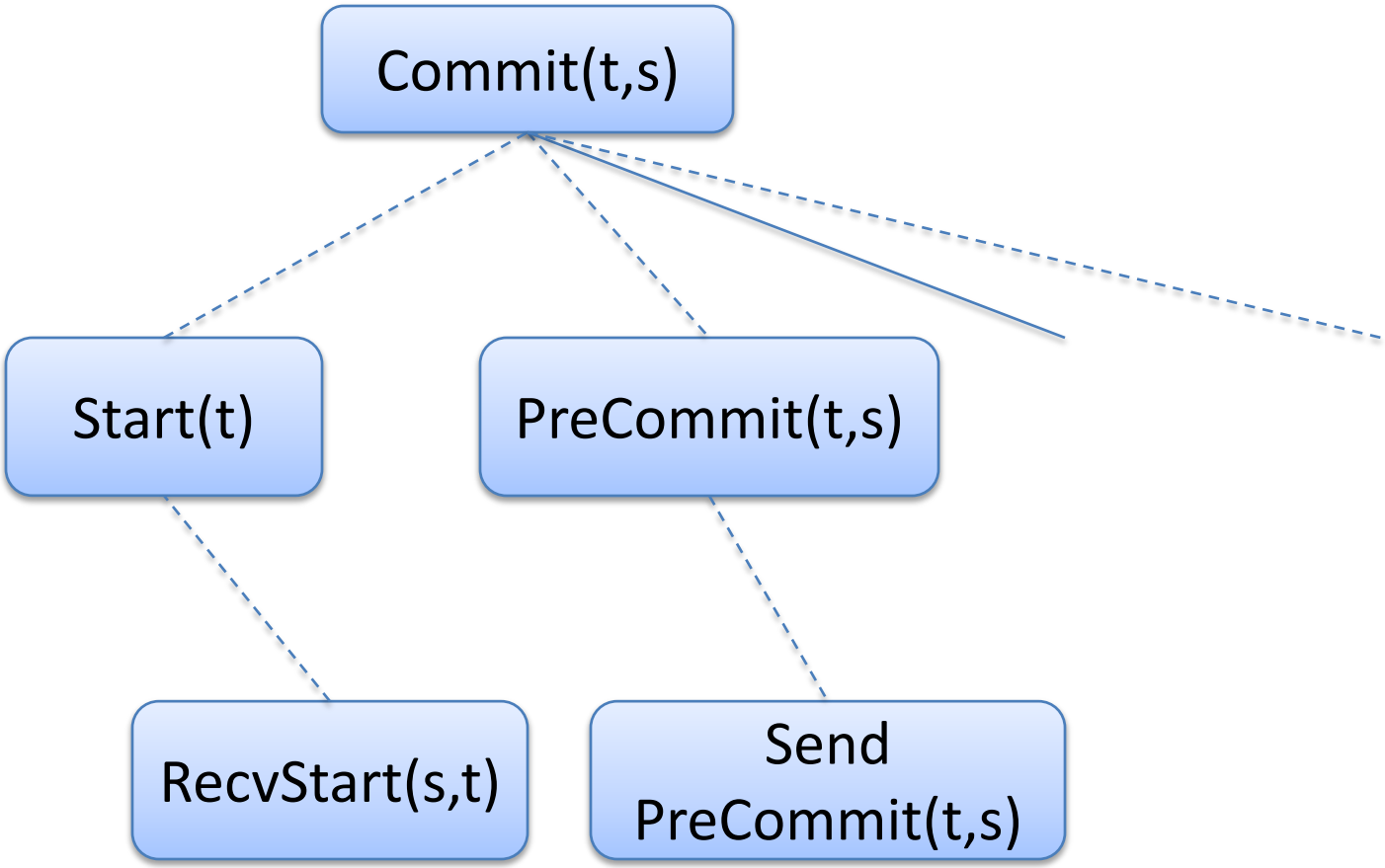
Architectural/role decomposition

- Explicit message/receive allows to separate requester/responder roles
- We do this by slicing the diagrams

Coordinator behaviour for database



Non-coordinator behaviour for database



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!