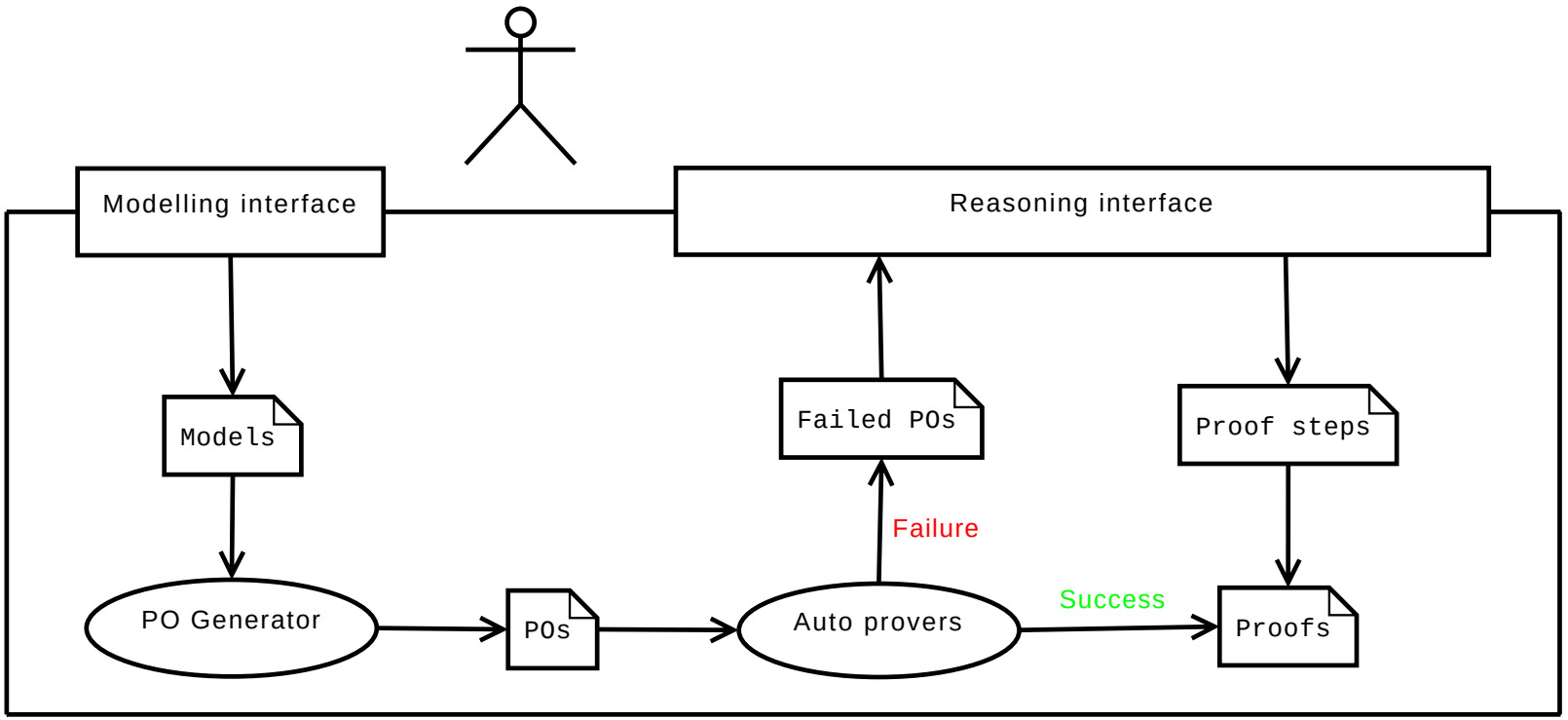
SEAR: Systems Evolution via Animation and Reasoning

Andrew Ireland, Maria Teresa Llano and Rob Pooley

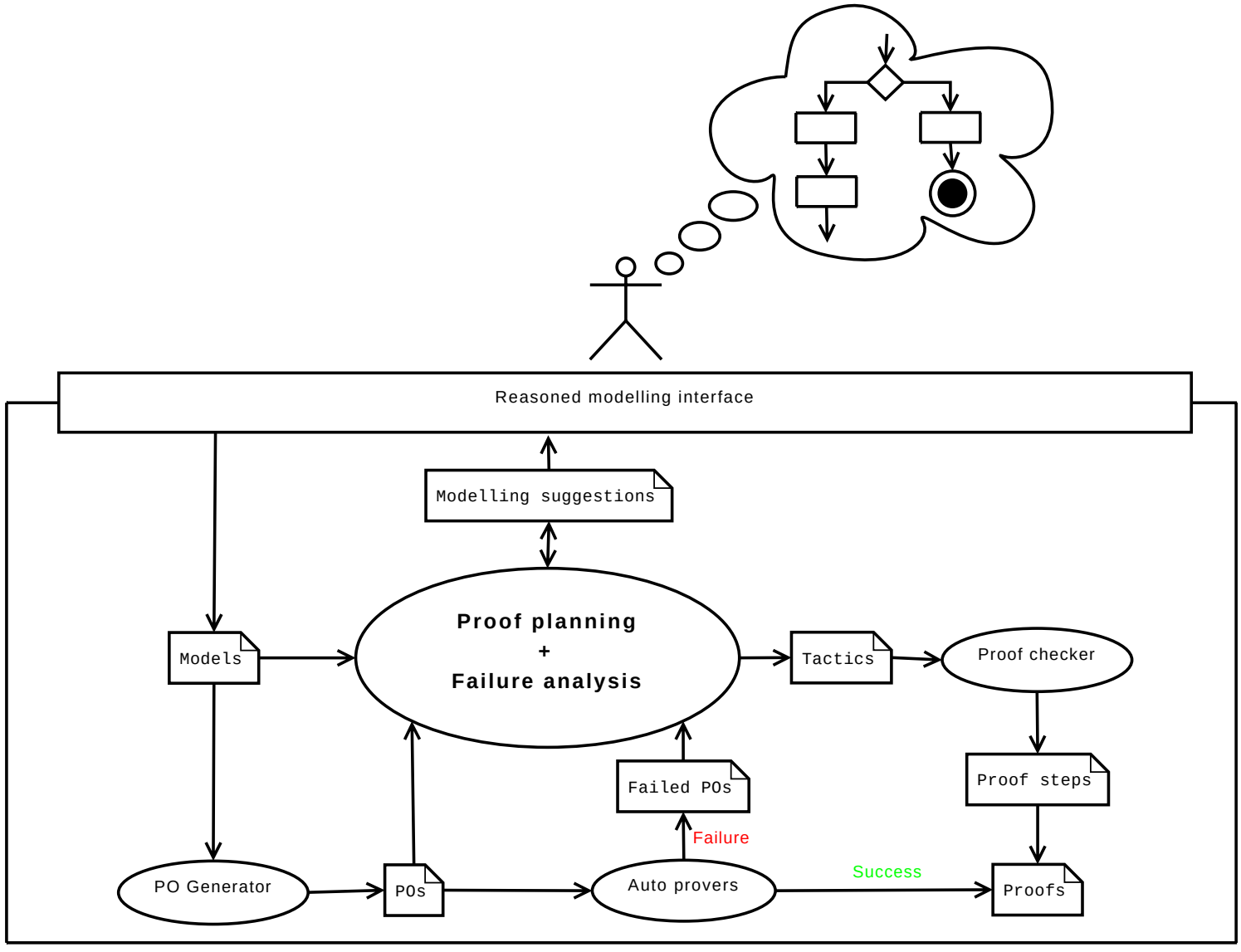
Dependable Systems Group
School of Mathematical and Computer Sciences
Heriot-Watt University

Rodin Workshop, July, 2009

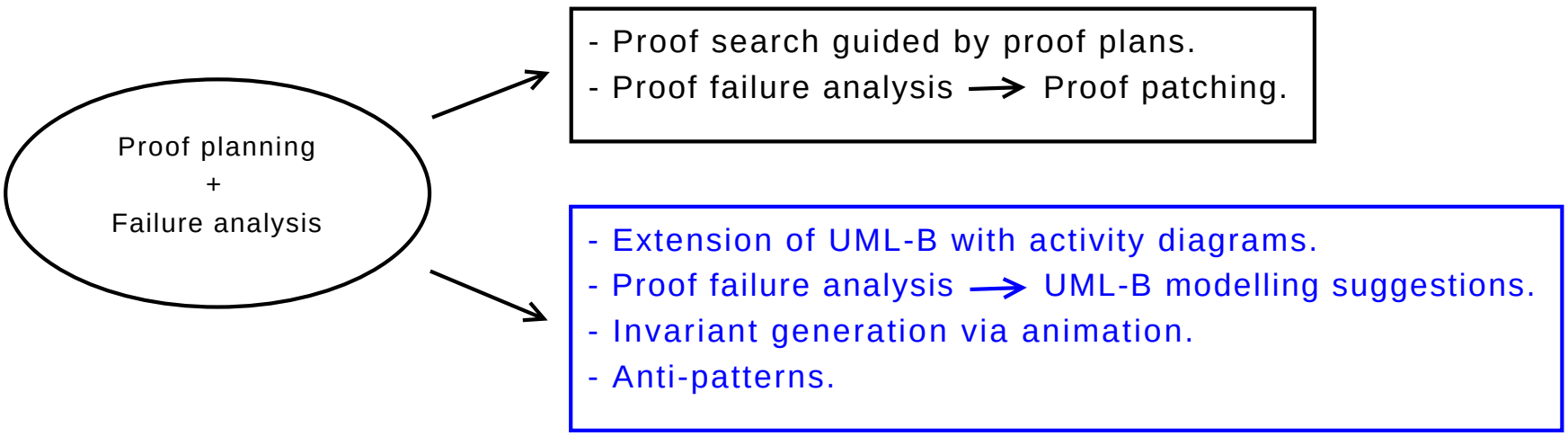
Currently


Modelling interface
Reasoning interface
Models
Failed POs
Proof steps
Failure
PO Generator
POs
Auto provers
Success
Proofs

Our objective

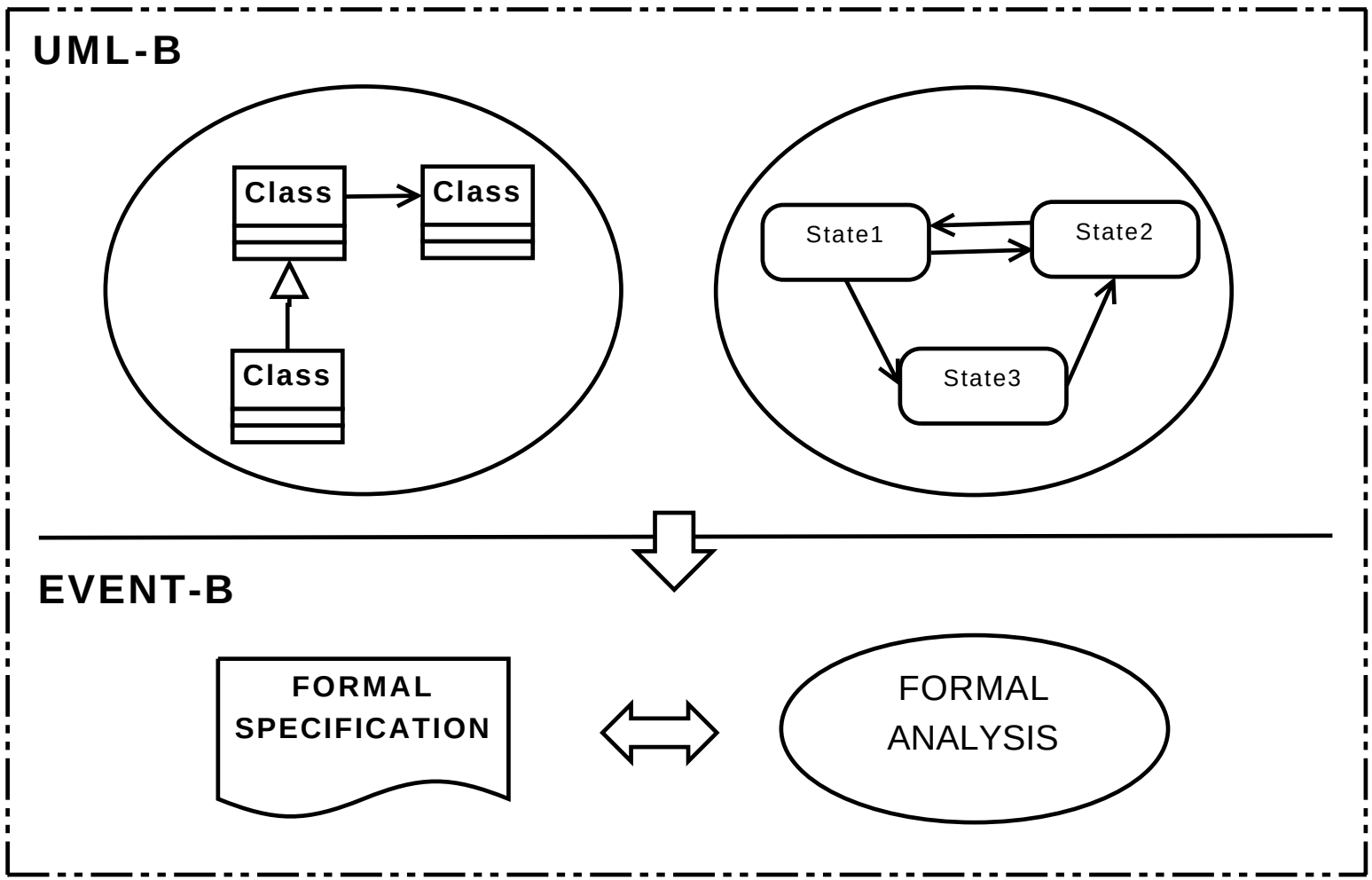


Specific objectives

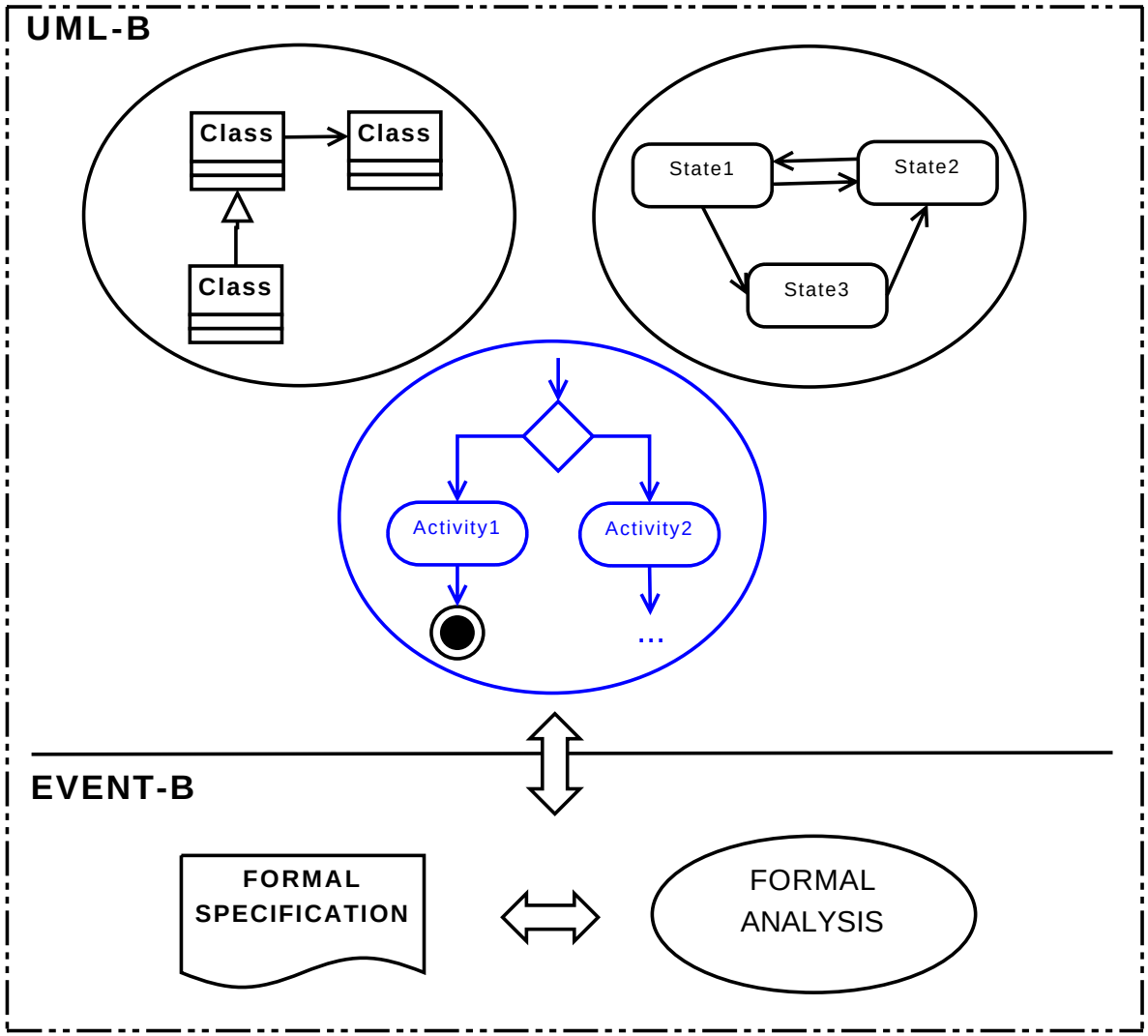


UML-B extension proposal

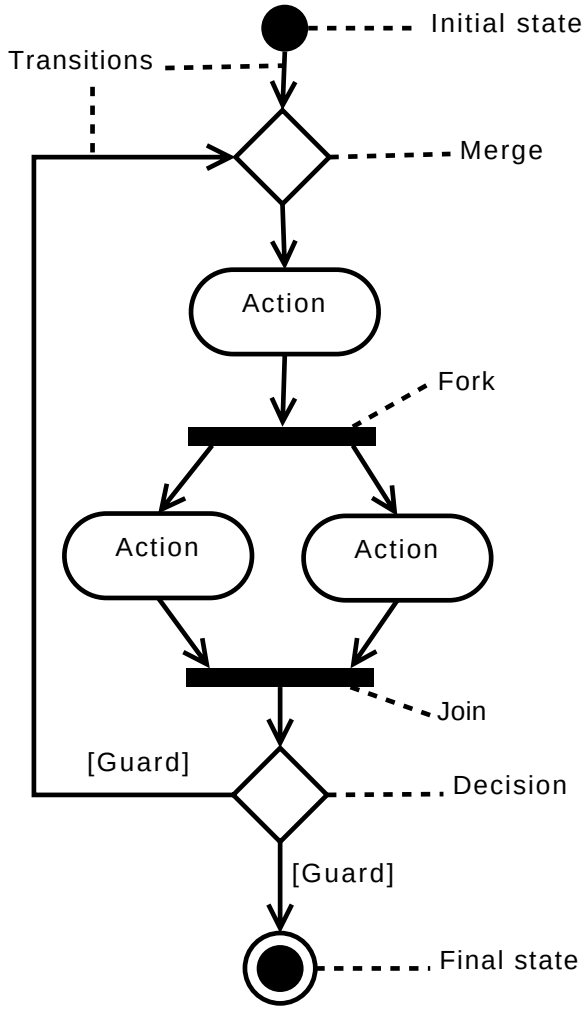
Extending UML-B



Extending UML-B with activity diagrams



Activity diagram



Why activity diagrams?

- Allow the representation of the flow of actions and the interactions between the elements of a system.
- Contain more detailed information about the behaviour of the system.
- It is possible the modelling of concurrent behaviour.
- Modelling suggestions in the form of activity diagrams.
- Anti-patterns have already been analysed with activity diagrams in UML. [1]

[1] M.T. Llano and R. Pooley. UML specification and correction of object-oriented anti-patterns. ICSEA 2009.

Analysis of anti-patterns in UML-B

Anti-patterns

Anti-patterns are design patterns whose purpose is to document common bad practices in software design and to suggest solutions to improve them.

Anti-patterns are not mistakes! they are models that produce bad consequences like:

- Slower execution times.
- Unnecessary consumption of resources
- Violation of good design principles.

Our purpose: To identify anti-patterns by reasoning about UML-B designs.

Advantages of analysing anti-patterns in UML-B

- It is possible to identify ineffective or potentially harmful models.
- They suggest a refactored suitable solution for the problem.
- Having a catalogue of UML-B anti-patterns would equip UML-B users with knowledge about patterns of models they should avoid.
- Anti-patterns can be analysed in the design stage.

Modelling suggestions through proof failure analysis

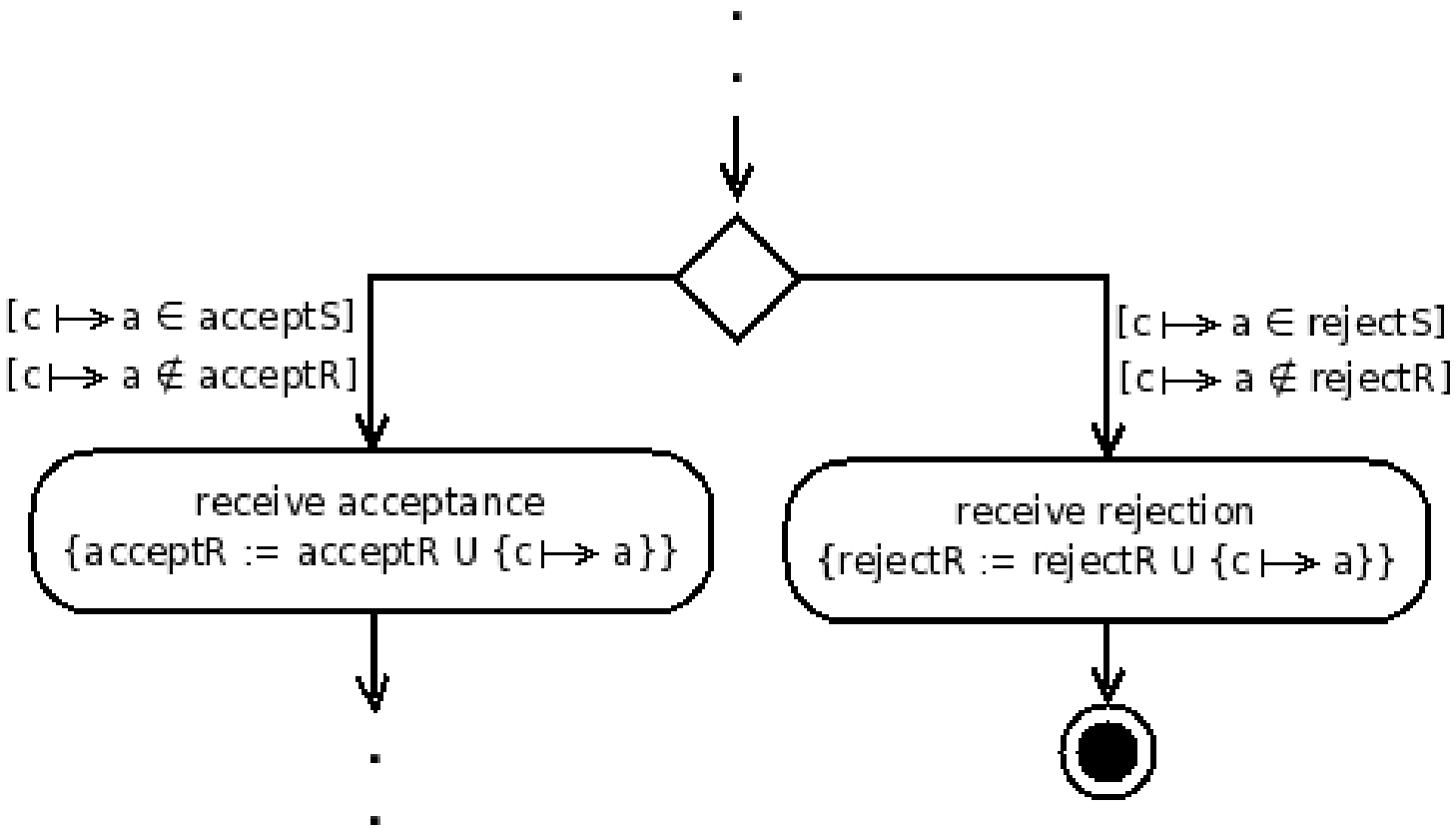
Proof failure analysis

- Analysis of failed proof obligations.
- Generation of modelling suggestions.
- Modelling suggestions translated into UML-B diagrams.
- Feedback to the user given in the form of UML-B designs rather than in the form of failed proof obligations or Event-B code.

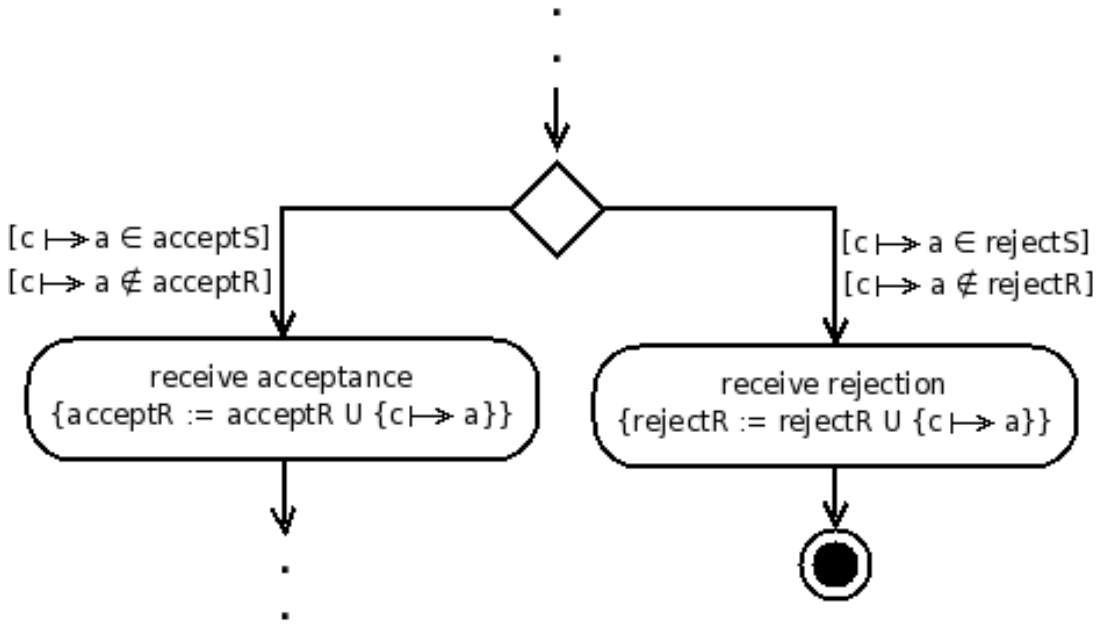
Example: The contract net protocol

- Protocol of distributed negotiation process.
- An agent (the initiator) needs to find an agent or groups of agents (participants) to be in charge of completing a task.
- The initiator calls for proposals from the participants.
- The best proposals are chosen.
- The participants are informed about their rejection or acceptance.
- The protocol finishes when the task is completed by the participants.

Acceptance and rejection of messages *buggy* model

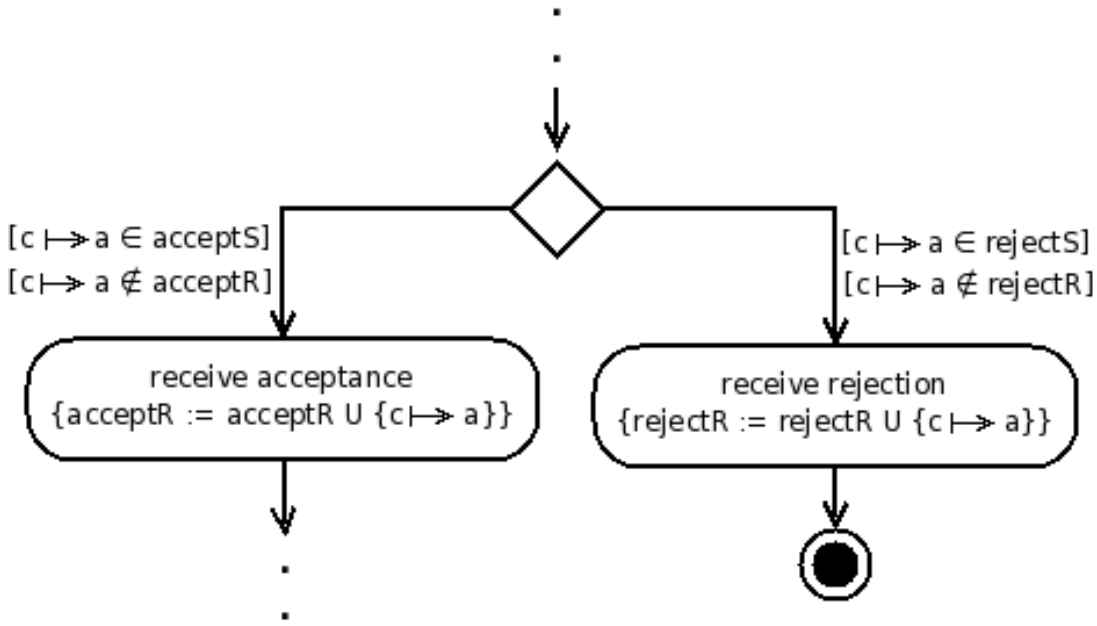


Translation to Event-B



Event *receiveAcceptance* $\widehat{=}$ **Any** c, a **Where** $c \longmapsto a \in acceptS$ $c \longmapsto a \notin acceptR$ **Then** $acceptR := acceptR \cup \{ c \longmapsto a \}$ **End**	**Event** *receiveRejection* $\widehat{=}$ **Any** c, a **Where** $c \longmapsto a \in rejectS$ $c \longmapsto a \notin rejectR$ **Then** $rejectR := rejectR \cup \{ c \longmapsto a \}$ **End**

Proof obligations



Invariant: $\text{acceptR} \cap \text{rejectR} = \emptyset$.

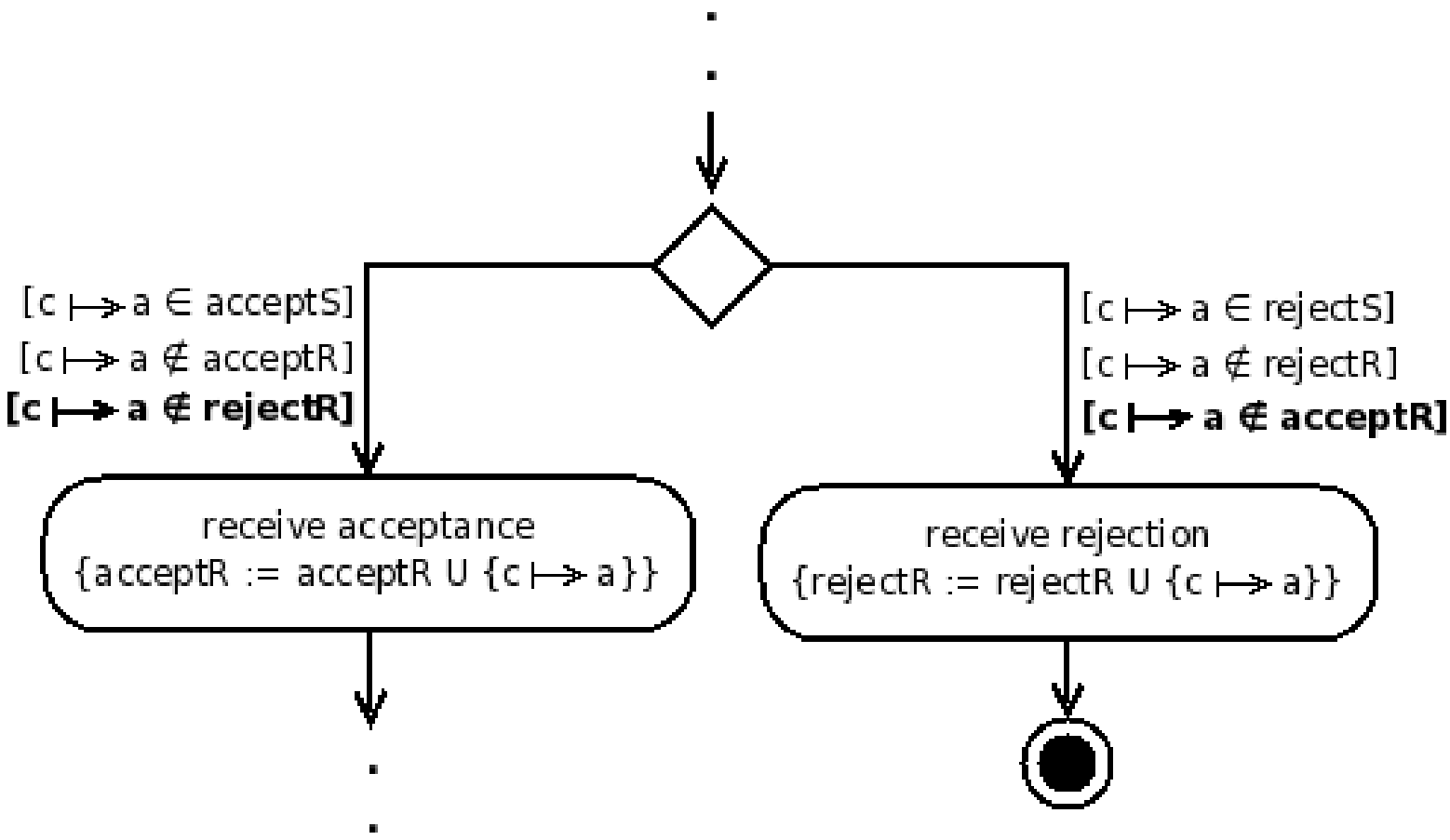
Proof obligations	
Hypothesis $\text{acceptR} \cap \text{rejectR} = \emptyset$ $c \longmapsto a \in \text{acceptS}$ $c \longmapsto a \notin \text{acceptR}$ **Goal** $(\text{acceptR} \cup c \longmapsto a) \cap \text{rejectR} = \emptyset$	**Hypothesis** $\text{acceptR} \cap \text{rejectR} = \emptyset$ $c \longmapsto a \in \text{rejectS}$ $c \longmapsto a \notin \text{rejectR}$ **Goal** $\text{acceptR} \cap (\text{rejectR} \cup c \longmapsto a) = \emptyset$

Reasoned modelling

$acceptR \cap rejectR = \emptyset$
$c \longmapsto a \in acceptS$
$c \longmapsto a \notin acceptR$
$\Rightarrow$
$(acceptR \cup c \longmapsto a) \cap rejectR = \emptyset$

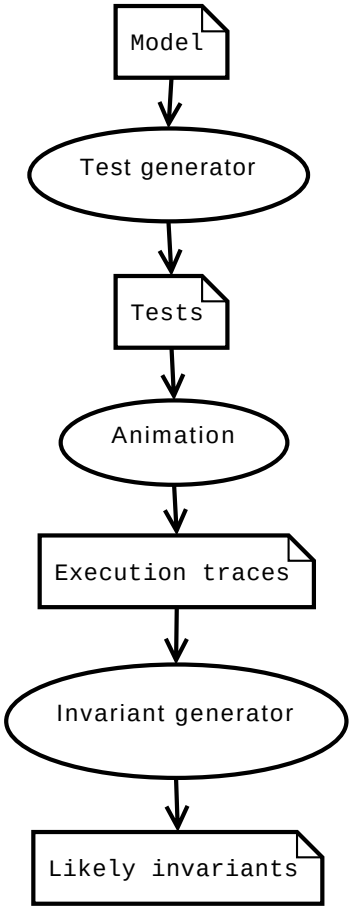
CASE SPLIT	
$acceptR \cap rejectR = \emptyset$ $c \longmapsto a \in acceptS$ $c \longmapsto a \notin acceptR$ $\mathbf{c \longmapsto a \notin rejectR}$  $\Rightarrow$ $(acceptR \cup c \longmapsto a) \cap rejectR = \emptyset$	$acceptR \cap rejectR = \emptyset$ $c \longmapsto a \in acceptS$ $c \longmapsto a \notin acceptR$ $\mathbf{c \longmapsto a \in rejectR}$  $\Rightarrow$ $(acceptR \cup c \longmapsto a) \cap rejectR = \emptyset$ However if we prove the negation of the goal: $acceptR \cap rejectR = \emptyset$ $c \longmapsto a \in acceptS$ $c \longmapsto a \notin acceptR$ $\mathbf{c \longmapsto a \in rejectR}$  $\Rightarrow$ $\mathbf{(acceptR \cup c \longmapsto a) \cap rejectR \neq \emptyset}$

Feedback to the user



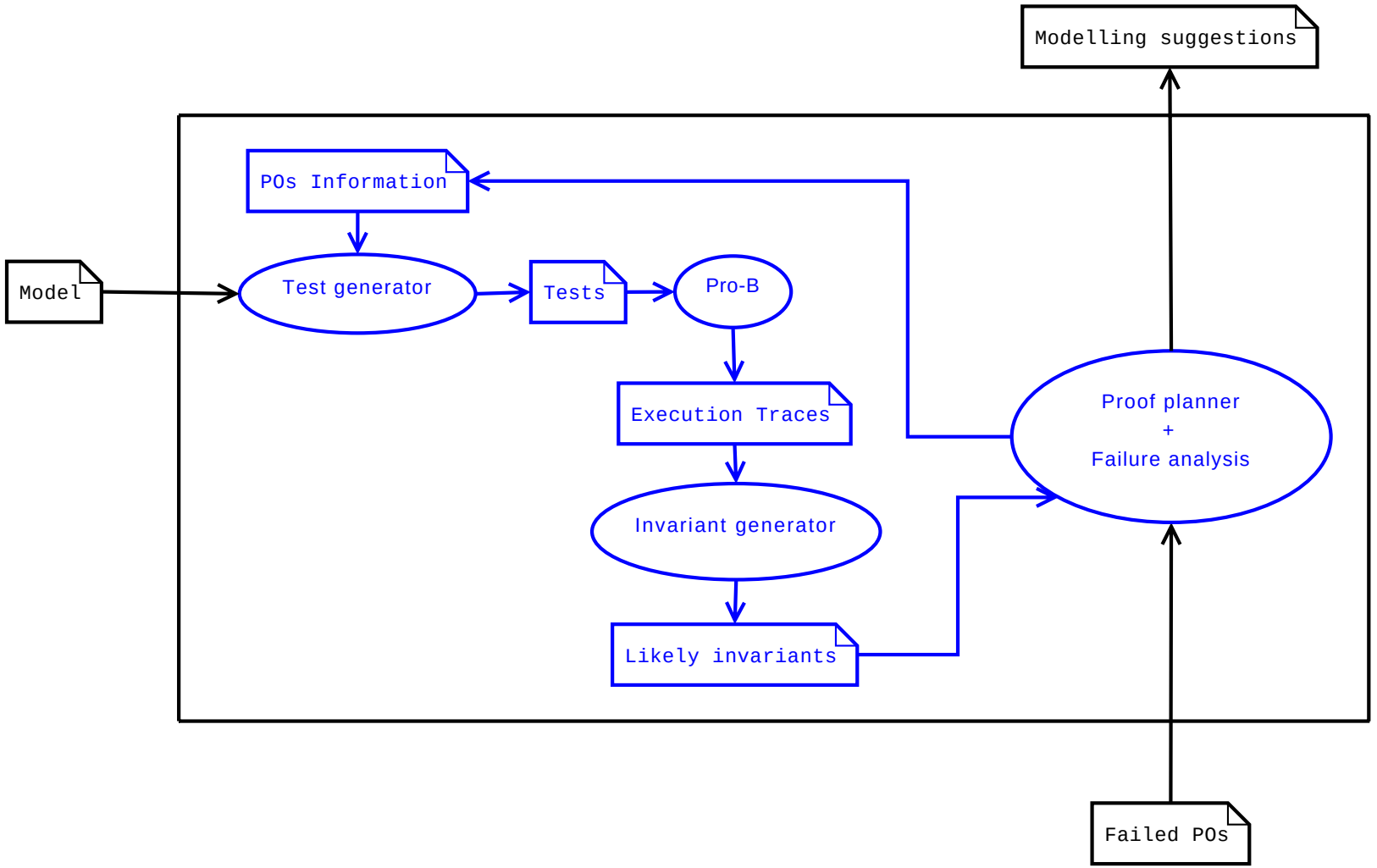
Automatic invariant generation

Invariant generation via animation



We are currently studying *Daikon* as a possible invariant generator to integrate with Rodin.

Invariant generator integrated with Rodin



Summary

Our main purpose is to help UML-B users by suggesting changes to their models trough:

- Proof failure analysis.
- Automatic invariant generation.
- Analysis of anti-patterns.