DD 2460
Software safety and security
Lecture 3

Formal Modelling. Introduction to Event-B modelling

What is a formal specification?

- A **formal specification** is the expression, in some **formal** language and at the some level of abstraction, of a collection of **properties** some **system** should satisfy.
- The formal specification depends on
  - what does "*system*" mean, i.e., where one draws the boundaries,
  - what kind of *properties* are of interest,
  - what level of abstraction is considered, and
  - what kind of *formal language* is used.

The “system” being specified may be:

- a descriptive model of the domain of interest;
- a prescriptive model of the software and its environment;
- a prescriptive model of the software alone;
- a model for the user interface;
- the software architecture;
- a model of some process to be followed;
- etc.

The “properties” under consideration may refer to:

- high-level goals;
- functional requirements;
- non-functional requirements about timing, performance, accuracy, security, etc.;
- environmental assumptions;
- services provided by architectural components;
- protocols of interaction among such components;
- and so on.

Formal specification

- “Formal” is often confused with “precise” (the former entails the latter but the reverse is not true).

- A specification is ***formal*** if it is expressed in a language made of three components:
  - rules for determining the grammatical well-formedness of sentences (the syntax);
  - rules for interpreting sentences in a precise, meaningful way within the considered domain (the semantics);
  - and rules for inferring useful information from the specification (the proof theory).
- The latter component provides the basis for automated analysis of the specification.

Why specify formally?

- Problem specifications are essential for designing, validating, documenting, communicating, reengineering, and reusing solutions.
- Formality helps in obtaining higher-quality specifications within such processes;
  - it also provides the basis for their automated support.
- The act of formalization in itself has been widely experienced to raise many questions and detect serious problems in original informal formulations.
- Besides, the semantics of the formalism being used provides precise rules of interpretation that allow many of the problems with natural language to be overcome. A language with rich structuring facilities may also produce better structured specifications.

Specify… for whom?

- Formal specifications may be of interest to different stakeholders having fairly different background, abstractions and languages:
  - clients
  - domain experts
  - users
  - architects
  - programmers
  - and tools.

Specify… when?

- There are multiple stages in the software life-cycle at which formal specifications may be useful:
  - when modeling the domain;
  - when elaborating the goals, requirements on the software, and assumptions about the environment;
  - when designing a functional model for the software;
  - when designing the software architecture;
  - or when modifying or reengineering the software.

Value of formal specification

- The cost of fixing a specification or design error is higher the later in the development that error is identified.

- ***Boehm's First Law*****:** ***Errors are more frequent during requirements and design activities and are more expensive the later they are removed.***

Specification methods

- Facilitate discovering errors at early stages of system development when they are less expensive to fix.
- Common errors introduced in the early stages of development are errors in understanding the system requirements and errors in writing the system specification.
- Without a rigorous approach to understanding requirements and constructing specifications, it can be very difficult to uncover such errors other than through testing of the software product after a lot of development has already been undertaken.

Why is it difficult?

- Lack of precision in formulating specifications resulting in ambiguities and inconsistencies that are difficult to detect.
- High complexity
  - complexity of requirements;
  - complexity of the operating environment of a system or
  - complexity of the design of a system.

The use of formal modelling

- The main aim is to overcome the problem of lack of precision.
- Formal modelling languages are supported by verification methods that support the discovery and elimination of inconsistencies in models.
- But precision does not address the problem of complex requirements and operating environments.
- Complexity cannot be eliminated but we can try to master it via **abstraction**.

Problem abstraction

- Abstraction can be viewed as a process of simplifying the problem at hand and facilitating our understanding of a system.
- Abstraction should
  - **focus** on the **intended purpose** of the system and
  - **ignore** details of **how** that purpose is achieved.

Abstraction

- If the purpose of the system is to provide some service, then
  - model what a system does from the perspective of the service user.
  - ‘user’ might be computing agents as well as humans
- If the purpose of the system is to control, monitor or protect some phenomena, then
  - the abstraction should focus on those phenomenon, considering in what way they should be monitoring, controlled or protected and should ignore the way in which this is achieved.

Refinement

- However, we do not ignore the complexity indefinitely: instead, through incremental modelling and analysis, we can gradually improve our understanding and analysis of a system and make it more detailed.
- This incremental treatment of complexity is called **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

Event-B

- It provides us with a rich modelling language, based on set theory
  - language allows precise descriptions of intended system behaviour (models) to be written in an abstract way
- Event-B uses the abstract machine notation as the basis.
- Event-B is successor of the B Method (also known as classical B).

From the B Method to Event-B

- Inventor: Jean-Raymond Abrial (his previous work is Z framework)
- Both classical B and Event-B are based on set theory
- Analyse models using proofs and additionally -- model checking, animation
- Refinement-based development
  - Verify conformance between higher-level and lower-level models
  - Chain of refinements

- Commercial tools for classical B: Atelier-B (ClearSy, France), B-Toolkit (B-Core, UK)
- Why Event-B: realisation that it is important to reason about system behaviour, not just software
- Event-B is intended for modelling and refining system behaviour

Industrial uses of Event-B

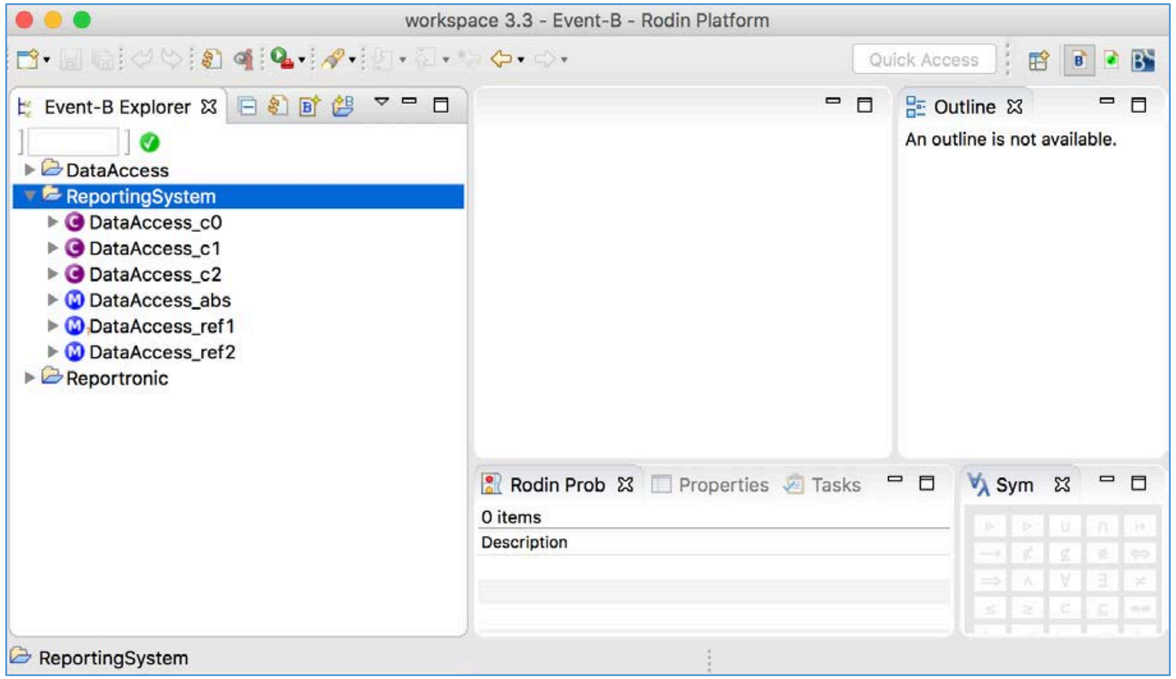
- Event-B in railway interlocking
  - Alstrom, Systerel
- Event-B in smart grids
  - Selex, Critical Software
- Event-B in a cruise control system and a start-stop system
  - Bosch
- Event-B in train control and signaling systems
  - Siemens Transportation

Rodin

- Rodin – the automated tool platform for Event-B.
- www.event-b.org
- Integrated development environment for Event-B
- Models can be created using built-in editor.
- The platform generates proof obligations that can be discharged either automatically or interactively.
- Rodin is a modular software and many extensions are available.
  - These include alternative editors, document generators, team support, and extensions (called plugins) some of which include support decomposition and records.

An Event-B project

- A **project** contains the complete mathematical development.
- Contains two kinds of components: **Contexts** and **Machines**.
- Projects and components are listed in the tool

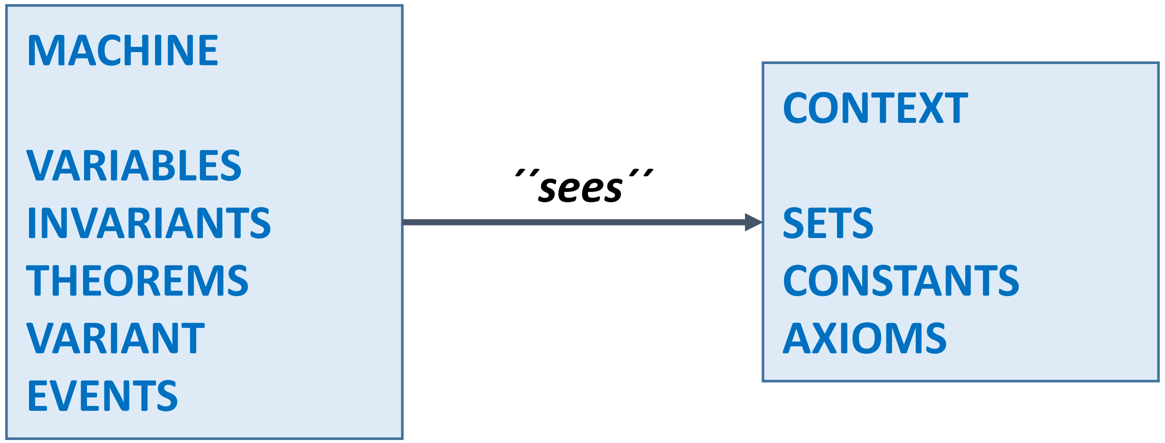


Structure of Event-B model (specification)

- An Event-B **model** is made of several components.

  A specification consists of

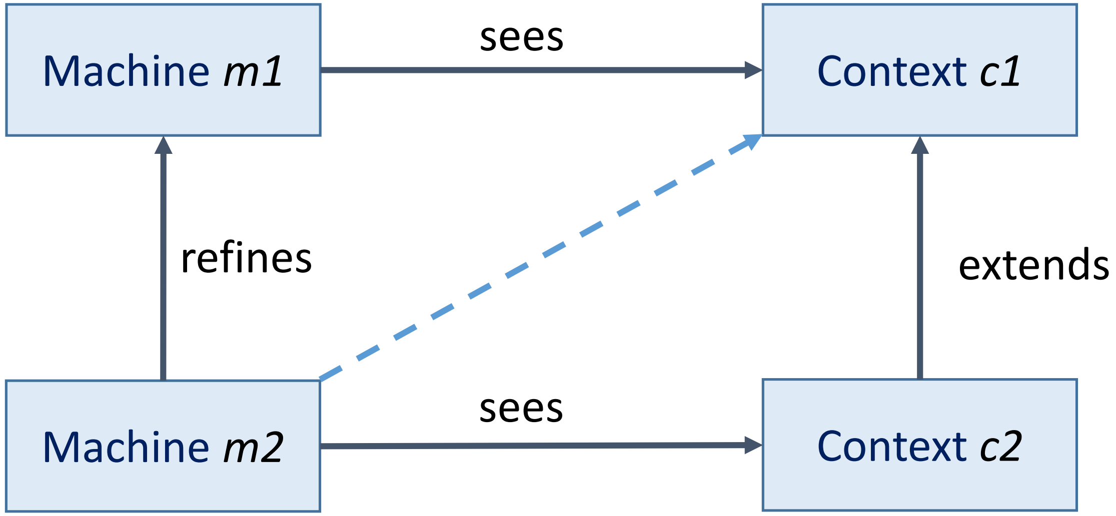
  - a *static* part, specified in a context, and
  - a *dynamic* part, specified in a machine.



Relationship between machines and contexts

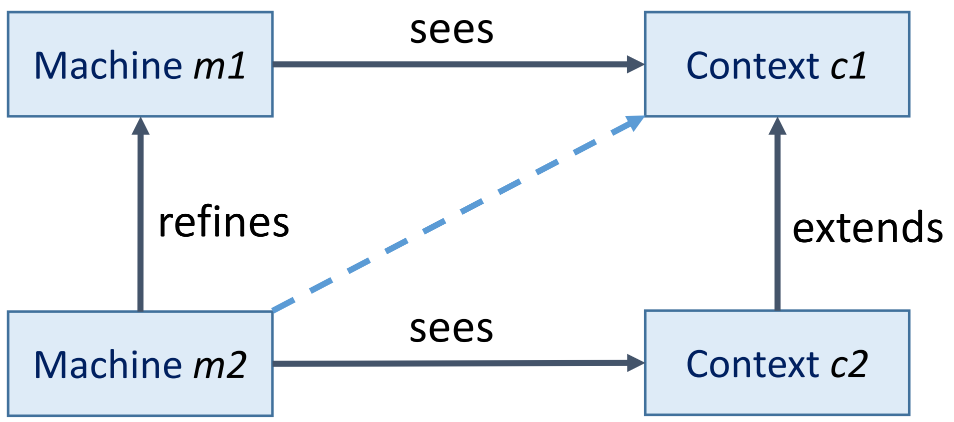
- **Contexts** contain the static structure of a discrete system (constants and axioms)
- **Machines** contain the dynamic structure of a discrete system (variables, invariants, and events)

- - *Machines see contexts*
- - *Contexts can be extended*
- - *Machines can be refined*



Visibility Rules

- A machine can see several contexts (or no context at all).
- A context may extend several contexts (or no context at all).
- A machine implicitly sees all contexts extended by a seen context.
- A machine only sees a context either explicitly or implicitly.
- A machine only refines at most one other machine.
- No cycle in the "refines" or "extends" relationships.



Context part

- A context with name **Context1** has the following form:

CONTEXT **Context1**

SETS ⟨list of carrier sets⟩

CONSTANTS ⟨list of constants⟩

AXIOMS ⟨list of labelled axioms⟩

THEOREMS* ⟨list of labelled theorems⟩

END

- the context contains the static part of a model
- the context defines sets, constants that can be used in several different machines
- different properties for the sets and constants are given in **axioms**-clause

Context part

- A context with name **Context1** has the following form:

CONTEXT **Context1**

SETS ⟨list of carrier sets⟩

CONSTANTS ⟨list of constants⟩

AXIOMS ⟨list of labelled axioms⟩

THEOREMS* ⟨list of labelled

theorems⟩

END

- A context has a unique name
- **sets**-clause contains the non-empty carrier sets
- **constants**-clause contains constants
  - they can be read but not assigned values
- **axioms**-clause lists the predicates that should hold for the constants
  - Defines types and logical properties
  - Hypotheses in all proof obligations
- **theorems**-clause lists the theorems
  - They have to be proved within the context

Context part

- A context with name **Context1** has the following form:

CONTEXT Context1 EXTENDS Context0

SETS ⟨list of carrier sets⟩

CONSTANTS ⟨list of constants⟩

AXIOMS ⟨list of labelled axioms⟩

THEOREMS* ⟨list of labelled

theorems⟩

END

- A context can extend another context
- **extends**-clause defines it

Example on context

CONTEXT *UniversityContext*

SETS STUDENTS …

CONSTANTS $max\_cource\_capacity$

AXIOMS

axm1: $max\_cource\_capacity \in \mathbb{Z}$

axm2: $max\_cource\_capacity \leq 30$

…

END

Machine part

- A machine with name **Machine1** has the following form:

```
MACHINE Machine1
        SEES* ⟨list of context names⟩
        VARIABLES ⟨list of variables⟩
        INVARIANTS ⟨list of labelled invariants⟩
        EVENTS ⟨list of events⟩
END
```

- A machine defines the ***dynamic behaviour*** of a model through events that are guarded by and act on the variables.
- **Machine**-clause gives the name of the machine
- **Sees**-clause lists the contexts that the machine can see
- **Variables**-clause gives state of the module that can be modified locally in the machine

Machine part

- A machine with name **Machine1** has the following form:

```
MACHINE Machine1
    SEES* ⟨list of context names⟩
    VARIABLES ⟨list of variables⟩
    INVARIANTS ⟨list of labelled invariants⟩
    EVENTS ⟨list of events⟩
END
```

- **Invariants**-clause lists the predicates that must hold for the variables and gluing invariants
  - The types of the variables
  - Restriction and relations between variables
- **Event**-clause contains the relevant events that change the state of the machine while preserving the invariant
  - An event describes the relationships between the state before the event takes place and just afterwards
  - Event **INITIALISATION** gives initial values to the variables and establishes the invariants

Remarks

- Machines should not be thought of as programs
  - although they might be implemented by software.
- The machine models a state and the events represent behaviour that could occur
  - the conditions that must apply if an event is to fire; and
  - the effect the event has on the state.
- All communication occurs through the state.
  - machine gives a representation of possible behaviours of some system. The system might contain non-software components.

Example on a machine

MACHINE *RegistrationSystem*
SEES *UniversityContext*

VARIABLES registered enrolled

INVARIANTS

inv1: registered ⊆ STUDENTS

inv2: enrolled ⊆ STUDENTS

inv3: enrolled ⊆ registered

…

EVENTS ⟨list of events⟩

END

An Event-B model

- A model can contain:
  - Only contexts (represents a pure mathematical structure)
  - Only machines (the model is not parametrised)
  - Both machines and contexts

- All machines and context identifiers must be distinct in the same model (project)

Event-B events

MACHINE **Machine1**

SEES* ⟨list of context names⟩

VARIABLES ⟨list of variables⟩

INVARIANTS ⟨list of labelled invariants⟩

EVENTS ⟨list of events⟩

END

- All events represents transitions
- The events modify the state of the variables
  - An event is a state transition in a discrete dynamic system.
- They are executed in one (atomic) step
  - Only one event fires at a time.
- An event essentially consists of guards and actions.

  Hence they are said to be **guarded** events.

  - **Event =** ***guard + action***

Guarded events

Event = *guard + action*

- An event essentially consists of guards and actions.
  - the guards define the necessary conditions for the event to be enabled
  - the actions define the way the variables of the machine are modified
- A guarded event can be executed only when its guard is true
  - if more than one guard is true, one of them is non-deterministically chosen
  - if no guards are true, the specification will terminate.

Event structure

```
Event1 =                                // event name
          any
                        x1  x2          // event parameters
          where
                        G1              // event guards
                        G2
                        …
          then
                        v1 := exp1      // event actions
                        v2 := exp2
                        …
          end
```

- Events are identified by unique names
- **any**-clause lists the parameters (or local variables) of the event
- **where**-clause contains the guards of the event, i.e., the conditions for the event to be enabled
- **then**-clause lists the actions of the event

Events: general form

- A machine can contain arbitrary many events
- An event can have one of the following forms:

```
Event1 =

any
    x1, x2
    where
      G1
      G2
       …
    then
      v1 := exp1
     v2 := exp2
       …
end
```

```
Event2 =
  when
    G1
    G2
    …
     then
  v1 := exp1
  v2 := exp2
   …
end
```

```
Event3 =

begin
  v1 := exp1
  v2 := exp2
                                …
end
```

Event guards

- A guards is a predicate that specifies enabling conditions under which events may occur
- Example, booking a room for the lecture

  grd1: $\boldsymbol{lec \in LECTURERS}$

  grd2: $\boldsymbol{lh} \in \boldsymbol{LHALL}$
- Guards should be strong enough to ensure invariants are maintained by the actions of an event but not too strong that they prevent desirable behaviour (invariants will be discussed later)

Event actions

- An action describes the ways one or several state variables are modified by the occurrence of an event
- An action might be either deterministic or non-deterministic
- There are three principle constructions — that Event B calls ***substitutions*** — for changing the state of a machine:
  - $\boldsymbol{x} \coloneqq \boldsymbol{e}$ // $\boldsymbol{x}$ *becomes equal to* the value of **e**
    This rule may be used multipletimes to assign to any number of variables.
  - $\boldsymbol{x}{:}\mid \boldsymbol{P}$ // $\boldsymbol{x}$ *becomes such that* it satisfies the ***before-after*** predicate **P**
  - $\boldsymbol{x}{:}\in \boldsymbol{S}$ // $\boldsymbol{x}$ *becomes in* the set **S**

- $x \coloneqq e$ and $x :\mid P$ can be extended to *multiple assignment*:

$$x, y \coloneqq e1, e2 \qquad \text{and} \qquad x, y :\mid P,$$

- and recursively to many variables.

- The variables must be distinct! $x, x, z \cancel{\coloneqq} e1, e2, e2$

- <u>*Note*</u>: all assignments can be written in the form: $x, y :\mid P$

Deterministic action (example)

- Example of deterministic actions on variables x, y and z:

Event example

...

then

act1: $x \coloneqq x + y$

act2: $y \coloneqq y + x - z$

...

- Notice that variables x and y should be distinct.
- Actions are supposed to be ´´performed´´ in parallel.
- Variables x and y are assigned to $x + y$ and $y + x - z$, respectively.
- Variable z is used but not modified by these actions

Non-deterministic actions

- A non-deterministic actions of the form

$$x :| P$$

// x *becomes such that* it satisfies the ***before-after*** predicate P

- The ***before-after-predicate*** gives the condition that holds just before the action takes place. It may contain all variables of the machine.

Non-deterministic actions (example)

$$\boldsymbol{x}, \boldsymbol{y} :| \ \boldsymbol{x}' > \boldsymbol{x} \wedge \boldsymbol{y}' < \boldsymbol{x}'$$

- On the LHS of operator **:|**, we have two distinct variables
- On the RHS, we have a ***before-after predicate***
- The RHS contains occurrences of $\boldsymbol{x}$ and $\boldsymbol{y}$ (<u>before</u> values) and primed occurrences $\boldsymbol{x}'$ and $\boldsymbol{y}'$ (<u>after</u> values)
- As a result (in this example):
  - $\boldsymbol{x}$ is assigned a value greater than its previous value
  - $\boldsymbol{y}$ is assigned a value smaller than that, $\boldsymbol{x}'$, assigned to $\boldsymbol{x}$

Non-deterministic action (cont.)

$$\boldsymbol{x} :\in \boldsymbol{S}$$

- The variable is assigned an arbitrary element of the set $\boldsymbol{S}$.
- This form is a special form of the previous one.

- Example 1:

$$\textbf{act1:}\quad \boldsymbol{x} :\in \{\boldsymbol{x}+\mathbf{1}, \boldsymbol{y}-\mathbf{2}, \boldsymbol{z}+\mathbf{3}\}$$

Here $\boldsymbol{x}$ is assigned any value from the set $\{\boldsymbol{x}+\mathbf{1}, \boldsymbol{y}-\mathbf{2}, \boldsymbol{z}+\mathbf{3}\}$

- Example 2:

$$\textbf{act2:}\quad \boldsymbol{st} :\in \boldsymbol{STUDENTS}$$

Here $\boldsymbol{st}$ is assigned any value from the set $\boldsymbol{STUDENTS}$

Non-deterministic action (more examples)

act1: $\boldsymbol{x} :\in \{\boldsymbol{a} \mid \mathbf{0} < \boldsymbol{a} \land \boldsymbol{a} < \mathbf{50}\}$

Here $\boldsymbol{x}$ is assigned any arbitrary number between 1 and 49.

CoffeeClub

The elementary description of machines will be illustrated with a simple running example of a coffee club.

- We will start with a machine that will be used to model some requirements of the coffee club.

Moneybank requirements

- For the coffee club we require a moneybank that stores money used by the coffee club.
- **REQ1:** a money bank is used for storing and reclaiming finite, non-negative funds for a coffee club;
- **REQ2**: there is an operation for adding money to the money bank;
- **REQ3:** there is an operation for removing money from the money bank; it cannot remove more money than the amount of money which bank contains.

Moneybank model

MACHINE CoffeeClub

VARIABLES *moneybank*

// The machine state is represented by the variable, *moneybank*, denoting the money bank for the coffee club.

INVARIANTS

inv1: *moneybank* $\in \mathbb{N}$ // **REQ1**: moneybank must not be negative.

// here $\in$ set membership

// $\mathbb{N}$ - the set of natural numbers

…

Model events

```
...
EVENTS
INITIALISATION ≙
        then
                act1: moneybank:= 0                 // we initialise moneybank to 0
        end

FEEDBANK ≙                      // REQ2: adding to moneybank
        any  amount
        where
                grd1: amount ∈ ℕ1 // ℕ1 is used rather than ℕ to prevent the event firing uselessly if amount = 0
        then
                act1: moneybank := moneybank + amount
        end
```

Model events (cont.)

```
…
ROBBANK ≙                    // REQ3: removing money from moneybank
        any amount
        where
                grd1: amount ∈ 1 .. moneybank
                        // The amount must not exceed the contents of money-
                                bank and we don't need to uselessly remove an amount of 0
        then
                act1: moneybank := moneybank - amount
        end
END
```

CoffeeClub model

Machine **CoffeeClub**
Variables *moneybank*
inv1: $moneybank \in \mathbb{N}$
Events
INITIALISATION $\triangleq$
then
act1: $moneybank := 0$
end
FEEDBANK $\triangleq$
any *amount*
where
grd1: $amount \in \mathbb{N}_1$
then
act1: $moneybank := moneybank + amount$
end

ROBBANK $\triangleq$
any *amount*
where
grd1: $amount \in 1 \,..\, moneybank$
then
act1: $moneybank := moneybank - amount$
end
END

Rodin platform

- Rodin is an open tool platform for the cost effective rigorous development of dependable complex software systems and services.
- This platform is based on the Event-B formal method and provides natural support for refinement and mathematical proof.
- This platform contributes to the Eclipse framework and is extensible using the Eclipse plug-in mechanism.
- Rodin builds and tries to solve proof obligations associated to abstract machines and their refinements.

Coffee club model in Rodin Editor

```
MACHINE
  CoffeeClub
VARIABLES
  moneybank
INVARIANTS
  inv1   :   moneybank ∈ ℕ
EVENTS
  INITIALISATION   ≙
  STATUS
   ordinary
  BEGIN
   act1   :   moneybank ≔ 0
  END
 FeedBank   ≙
 STATUS
  ordinary
 ANY
  amount
 WHERE
  grd1   :   amount ≥ 0
 THEN
  act1   :   moneybank ≔ moneybank + amount
 END
  RobBank   ≙
  STATUS
   ordinary
  ANY
   amount
  WHERE
   grd1   :   amount ∈ 1‥moneybank
  THEN
   act1   :   moneybank ≔ moneybank − amount
  END

END
```

Correctness of specifications

- Modelling in Event B is based on mathematical (logical) proofs
- In predicate logic we have:

FALSE ⇒ any statement

anything can be proved from false assumptions

- Hence, a contradictive specification can be implemented by any program
  - if the invariant evaluates to FALSE (it is unsatisfiable) then anything can be proved;
  - the Event-B specification is trivially ”correct”.
- In order to prohibit this, Event-B forces the developers to check that the first specification is correct and consistent.

Conditions of specification correctness

- In order to check the correctness of a specification, Event-B generates the following proof obligations
- **Conditions for the context**
  - checks that the axioms and theorems are well defined
- **Conditions for the invariant**
  - checks that the invariant and theorems are well defined
- **Conditions for the initialisation**
  - proves that the assignment statements in the initialisation establish a state that satisfies the invariant(s)
- **Conditions for the events**
  - proves that every model event preserves the invariant(s)

Proof Obligations: Sequent representation

- Sequent representation of PO:
  - **hypotheses**
  - $\vdash$
  - **goal**
- The truth of the hypotheses leads to the truth of the goal.
- The symbol $\vdash$ is sometimes called *stile* or *turnstile*.
- If any of the hypotheses is $\bot$(FALSE) then any goal is trivially established.
- If the hypotheses are identically $\top$ (TRUE) then the hypotheses will be omitted.

Consistency of Contexts

- In order to be consistent the Event-B context must satisfy the following properties:

1. All its axioms must be well defined (**axm/WD**)
2. All its theorems must be well defined (**thm/WD**)
3. All its theorems must be proved (**thm/THM**)

Well Defined Expressions

- All mathematical operators (functions) are not defined for all inputs of the type they accept.
  - Type-checking is not sufficient to check well-formedness
  - Two examples are ”/” and ”mod” which are not defined if their second argument is zero
- The Rodin tool generates proof obligations to check that all expressions are well defined.
- We denote that an expression (or a predicate) **E** is well defined by **WD(E)**

Well Defined Expressions

- Well definedness is defined by induction of the structure of the expression (predicate)
- $WD(P \wedge Q) = WD(P) \wedge (P \implies WD(Q))$
- $WD\big(F(E)\big) = WD(F) \wedge WD(E) \wedge E \in dom(F) \wedge F \in A$+-->B
- $WD(E\ /\ F) = WD(F) \wedge WD(E) \wedge F \neq 0$

- ***Example***:
- $WD(n > 0 \wedge 1/n) = WD(n > 0) \wedge (n > 0 \implies WD(1/n))$

Consistency of a Machine

- In order to be consistent a machine must satisfy the following conditions:

1. All it invariants and theorems must be well defined (**inv/WD** and **thm/WD**)
2. All its event guards and actions must be well defined (**grd/WD** and **act/WD**)
3. All its nondeterministic events must be feasible (**evt/act/FIS**)
4. All its theorems must be proved (**thm/THM**)
5. All invariants must be established by the initialisation (**INIT/inv/INV**)
6. All invariants must be preserved by all events (**evt/inv/INV**)

Well definedness

- Well definedness for machines are defined in the same way as for contexts
- Also for machines the theorems have to be proved

Feasibility of non-determinism

- *Feasibility* states that the action of an event is always feasible whenever the event is enabled.
- In other words, there are always possible after values for the variables satisfying the before-after predicate.
- The feasibility proof obligation **evt/act/FIS** is as follows:

- Axioms and theorems
- Invariants and theorems
- Guards of the event
- $\vdash$
- $\exists v'.$ Before-after predicate

Invariant preservation

- An essential feature of an Event-B machine M is its invariants:
  - They show properties that hold in every reachable state of the machine.
- The invariant preservation proof obligation **evt/inv/INV** is as follows:

Axioms and theorems
Invariants and theorems
Guards of the event
Before-after predicate of the event
$\vdash$
Modified specific invariant

Invariant preservation: example

- Assume that we have an event **EVENT** and an invariant $y \in N$.
- The event is given as
- **EVENT** =
- **any** x
- **where**

  $x \in N$

  $y < z$
- **then** $y := z+x$
- **end**
- and the invariant is given as $y \in N$
- ▪ BA(**EVENT**) = ∃v. Guards ∧ BA(S)

$y \in N$
$x \in N$
$y < z$
$(\exists x. x \in N \wedge y < z \wedge y' = z+x)$
$\vdash$
$y' \in N$

Invariant preservation: example (cont.)

INST_HYP

EQ_LR

$y \in N$ $x \in N$ $y < z$ $(\exists x. x \in N \land y < z \land y' = z + x)$ $\vdash$ $y' \in N$	$y \in N$ $x \in N$ $y < z$ $y' = z + x$ $\vdash$ $y' \in N$	$y \in N$ $x \in N$ $y < z$ $y' = z + x$ $\vdash$ $z + x \in N$

Invariant establishment

- *Invariant establishment* states that any possible state after initialisation given by the after predicate must satisfy the invariant *I*.
- The invariant establishment proof obligation **INIT/inv/INV** is as follows:

Axioms and theorems
Guards of the event
Before-after predicate of the event
$\vdash$
Modified specific invariant

Contradicting events

- An event can be contradicting (its correctness cannot be proved), if
- its guard is too weak
- the guard or the action is incorrect
- the invariant is too strong (too precise) (states properties that are not maintained by the events)
- the invariant is too weak (does not give enough assumptions for the proof to succeed)
- the invariant simply is wrong

Consistency of a Machine

- In order to be consistent a machine must satisfy the following conditions

1. All it invariants and theorems must be well defined (**inv/WD** and **thm/WD**)
2. All its event guards and actions must be well defined (**grd/WD** and **act/WD**)
3. All its nondeterministic events must be feasible (**evt/act/FIS**)
4. All its theorems must be proved (**thm/ THM**)
5. All invariants must be established by the initialisation (**INIT/inv/INV**)
6. All invariants must be preserved by all events (**evt/inv/INV**)
7. **<u>Deadlock freeness (DLF)</u>**

Deadlock freeness

- For non-terminating systems we need to guarantee that there is always an enabled event, i.e. that the system is deadlock free
- The deadlock freeness proof obligation **DLF** is as follows:

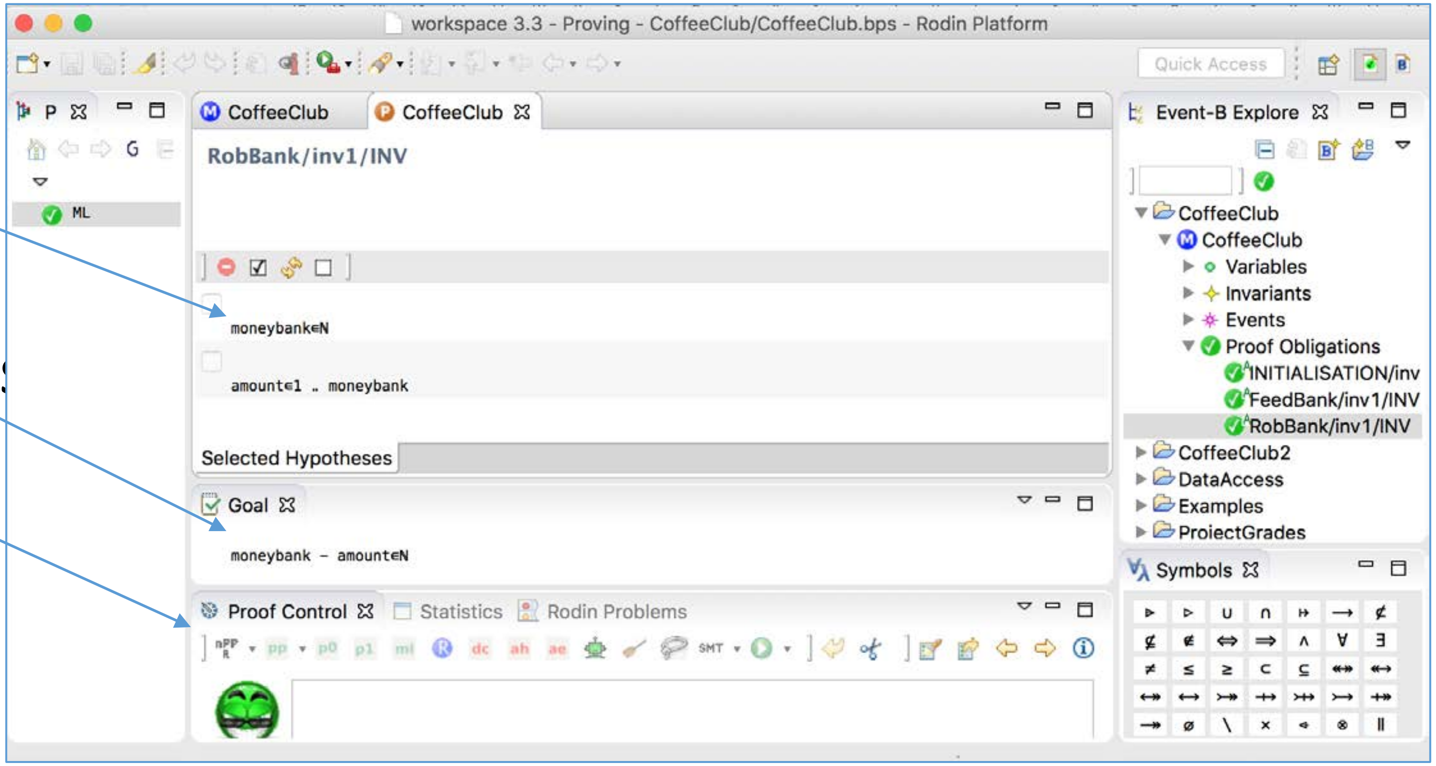
Axioms and theorems
Invariants and theorems
$\vdash$
Disjunction of the guards of the events

Proof in Rodin

- When discharging a proof in Rodin, three views are available:
- hypotheses,
- goal and
- proof control that provid

access to a range of provers

tactics for manual proof.



Labelling of proof obligations

- Proof obligations are labelled by Rodin as follows:
  - **event-name/predicate id/proof type id**
- For example:

- Green color indicates - a proof obligation to show that the **INITIALISATION** event satisfies the invariant labelled with inv1.

- Red (brown) color indicates - a proof obligation to show that the **RobBank** event does not satisfy the invariant labelled with inv1.

More examples on POs

CoffeeClub model (from Lecture 8)

MACHINE **CoffeeClub**

VARIABLES *moneybank*

inv1: *moneybank* $\in \mathbb{N}$

EVENTS

INITIALISATION ≜

then

act1: *moneybank*:= 0

end

FEEDBANK ≜

any *amount*

where

grd1: *amount* $\in \mathbb{N}_1$

then

act1: *moneybank* := *moneybank* + *amount*

end

ROBBANK ≜

any *amount*

where

grd1: *amount* ∈ *1 .. moneybank*

then

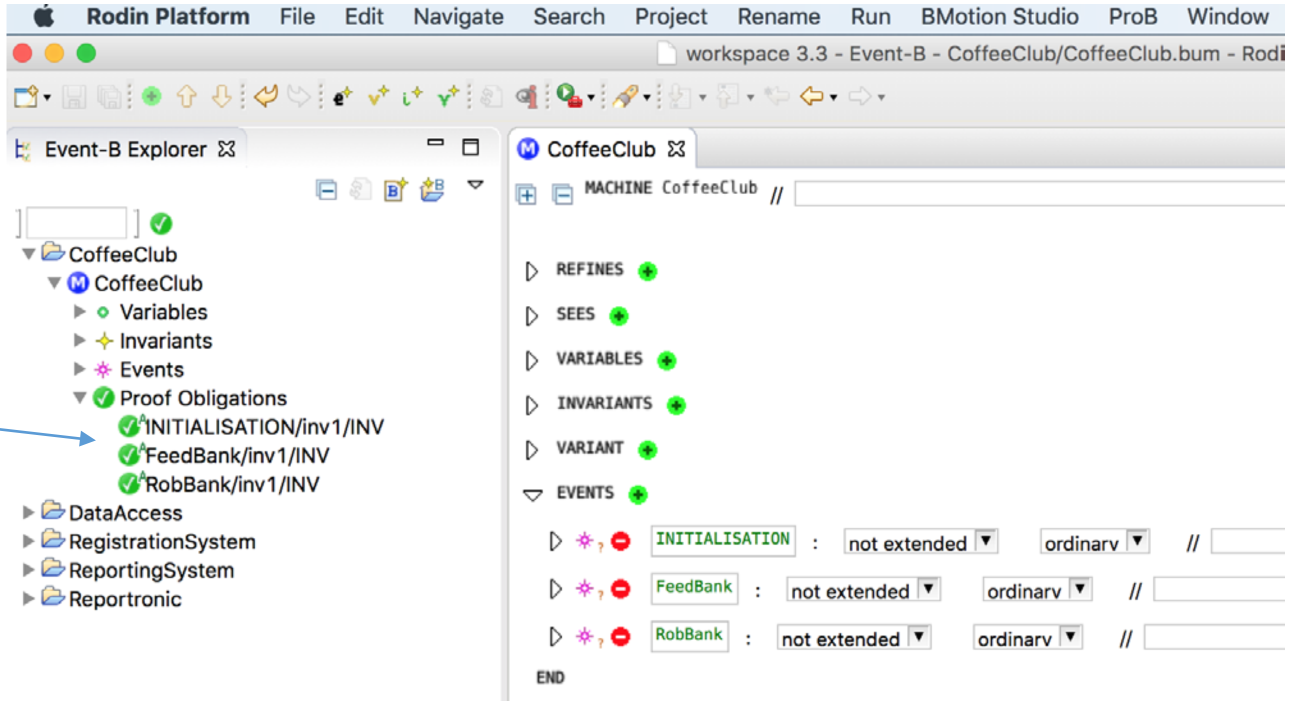
act1: *moneybank* := *moneybank* - *amount*

end

END

Proof Obligations for CoffeeClub

- CoffeeClub is a very simple model and the POs are correspondingly simple.
- As a consequence the POs are easily discharged automatically by the provers in the Rodin tool.
- The following POs are generated for the above machine.
- Notice that all POs concerned with maintenance of INV 1 are verifying that REQ1 is satisfied.



Proving prove obligations

- **INITIALISATION/inv1/INV:**
- $\top$
- $\vdash$
- $0 \in \mathbb{N}$

Invariants
inv1: *moneybank* $\in \mathbb{N}$
…
INITIALISATION $\triangleq$
then
act1: *moneybank* := 0
end

- Assuming nothing, ($\top$ or true), show that 0 is an element of the set of natural numbers. Clearly, it is true.
- Verifying that the **INITIALISATION**, moneybank := 0, establishes the invariant *moneybank* $\in \mathbb{N}$.

Proving prove obligations

- **FEEDBANK/inv1/INV:**
- $moneybank \in \mathbb{N}$
- $amount \in \mathbb{N}_1$
- $\vdash$
- $moneybank + amount \in \mathbb{N}$

Invariants

inv1: $moneybank \in \mathbb{N}$

…

FEEDBANK ≙

any *amount*

where

grd1: $amount \in \mathbb{N}_1$

then

act1: $moneybank := moneybank + amount$

end

- Verifying that the actions of **FEEDBANK**, $moneybank := moneybank + amount$, maintains the invariant, $moneybank \in \mathbb{N}$.
  This is clearly true as both *moneybank* and *amount* are natural numbers.

Proving prove obligations

- **ROBBANK/inv1/INV:**
- $moneybank \in \mathbb{N}$
- $amount \in 1..\ moneybank$
- $\vdash$
- $moneybank - amount \in \mathbb{N}$

Invariants

inv1: $moneybank \in \mathbb{N}$

…

RODBANK ≙

any *amount*

where

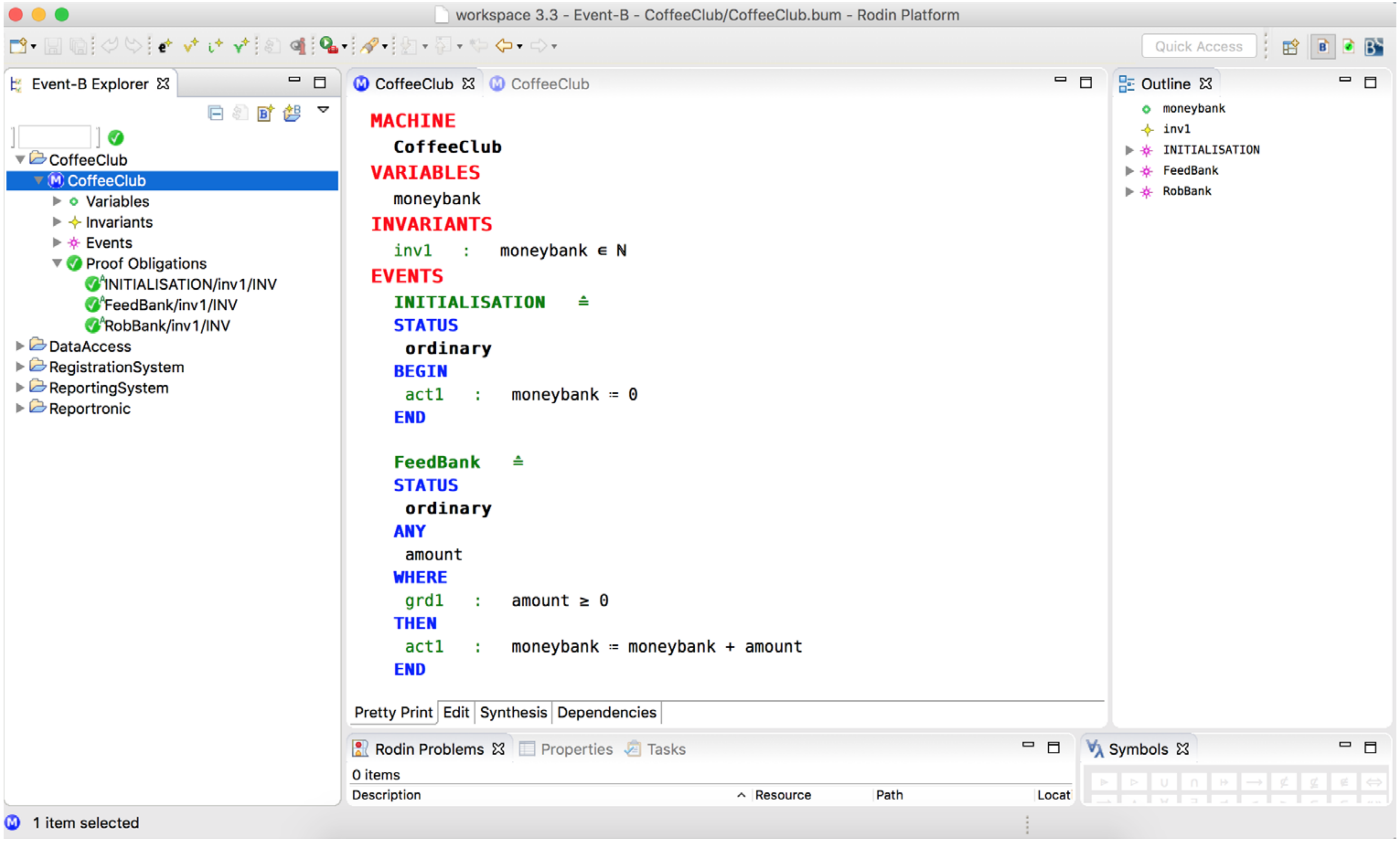
grd1: $amount \in 1..\ moneybank$

then

act1: $moneybank := moneybank - amount$

end

- Similar to the preceding POs, but this time verifying that $moneybank \in N$ is maintained by the action $moneybank := moneybank - amount$.
- This is not quite so simple, but the guard $amount \in 1\ ..\ moneybank$ ensures that amount ≤ moneybank and hence the invariant is maintained.


workspace 3.3 - Event-B - CoffeeClub/CoffeeClub.bum - Rodin Platform
Quick Access
Event-B Explorer
CoffeeClub
CoffeeClub
Variables
Invariants
Events
Proof Obligations
INITIALISATION/inv1/INV
FeedBank/inv1/INV
RobBank/inv1/INV
DataAccess
RegistrationSystem
ReportingSystem
Reportronic
CoffeeClub
CoffeeClub
MACHINE
CoffeeClub
VARIABLES
moneybank
INVARIANTS
inv1 : moneybank ∈ ℕ
EVENTS
INITIALISATION ≙
STATUS
ordinary
BEGIN
act1 : moneybank ≔ 0
END
FeedBank ≙
STATUS
ordinary
ANY
amount
WHERE
grd1 : amount ≥ 0
THEN
act1 : moneybank ≔ moneybank + amount
END
Pretty Print
Edit
Synthesis
Dependencies
Outline
moneybank
inv1
INITIALISATION
FeedBank
RobBank
Rodin Problems
Properties
Tasks
0 items
Description
Resource
Path
Locat
Symbols
1 item selected