DD2460 Lecture 6.
More examples of specifications and refinement

Elena Troubitsyna

Lecture outline

Examples of using relations and functions in specifications of

- access control

- seat registration

Example of refinement in Event-B

Example: printer access for students

The system tracks the permissions that students of a system have with regard to the printers attached to the system.

- A system should support adding a permission for a student in order to get an access to a particular printer and removing a permission.
- A system should support removing a student’s access to all printers at once.
- A system should support giving the combined permissions of any two students to both of them.

Printer access

- Permissions are naturally expressed as a *relation* between students and printers, so the machine makes use of a variable whose type is relation.
- Since the machine will have to keep track of changing permissions, it will make use of a *variable* ***access*** whose type is a *relation* between *STUDENTS* and *PRINTERS*.
- As permissions are added or removed, the variable will be updated to reflect the information.

Printer access: context

CONTEXT *PrinterAccess_c0*
SETS *STUDENTS*
PRINTERS
AXIOMS
axm1: *finite(STUDENTS)*
axm2: *finite(PRINTERS)*
axm3: *STUDENTS* $\neq \emptyset$
axm4: *PRINTERS* $\neq \emptyset$
END

Printer access: machine

MACHINE *PrinterAccess_m0*
SEES *PrinterAccess_c0*
VARIABLES *access*
INVARIANTS
    inv1: *access* $\in$ *STUDENTS* $\leftrightarrow$ *PRINTERS*
EVENTS
  INITIALISATION $\triangleq$
    begin
      act1: *access* := $\emptyset$
    end
…

Model events

```
ADD ≙
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
    then
        act1: access:=access ∪ {st ↦ pr}
    end
BLOCK ≙
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
        grd3: st ↦ pr ∈ access
    then
        act1: access:=access \{st ↦ pr}
    end
```

Model events

```
BAN ≜
    any st
    where
        grd1: st ∈ STUDENTS
    then
        act1: access:={st} ⩤ access /use of domain subtraction
    end
UNIFY ≜
    any st1  st2
    where
        grd1: st1 ∈ STUDENTS
        grd2: st2 ∈ STUDENTS
    then
        act1: access:= access ∪ ({st1} × access[{st2}]) ∪ ({st2} × access[{st1}])
    end
END
```

Relational image

Printer access rules

- Assume that we want to restrict the number of printers that a student can have access to.

For example, a student can use no more than 3 printers.

We have to reflect this new functionality into our model.

Model events: modification of ADD event

ADD ≙

any *st pr*

where

grd1: *st* ∈ STUDENTS

grd2: *pr* ∈ PRINTERS

***grd3*: ??? // we have to specify new condition here**

then

act1: *access:=access* ∪ $\{st \mapsto pr\}$

end

Model events: modification of ADD event

ADD $\triangleq$

any *st pr*

where

grd1: $st \in STUDENTS$

grd2: $pr \in PRINTERS$

grd3: $card(\{st\} \lhd access) < 3$ *// new guard*

then

act1: $access := access \cup \{st \mapsto pr\}$

end

Use of domain restriction

*// We restrict a domain of **access** relation by a set containing one element student **st**, i.e.,* $\{st\} \lhd access$. As a result of this operation we get a set of pairs, whose the first element is ***st***. Then by ***card*** operator we count a number of such pairs. Thus, we get a number of printers that this particular student ***st*** has access to.

Model events: modification of UNIFY event

Similarly, we have to modify the event **UNIFY.**

However, the new guard here will be rather complex

- *Informally:* we have to check, if, after the Unify operation, two students still will have access to no more than 3 printers.

This means that the following property should be defined as a model invariant (and, consequently preserved during events execution):

$$\forall st.\ st \in \boldsymbol{dom}(\boldsymbol{access}) \Rightarrow \boldsymbol{card}(\{st\} \lhd \boldsymbol{access}) \leq 3$$

More examples

- ***Every person is either a student or a lecturer. But no person can be a student and a lecturer at the same time.***

$STUDENTS \subseteq PERSONS, LECTURERS \subseteq PERSONS$

$LECTURERS \cup STUDENTS = PERSONS$

$LECTURERS \cap STUDENTS = \emptyset$

- ***Only lecturer can teach course***

$e.g.,\ CourseLecturer\ \in COURSES \leftrightarrow LECTURERS$

More examples

- ***Every* course is given by *at most one* lecturer**

$CourseLecturer \in COURSES \longrightarrow LECTURERS$ // total function

- ***A lecturer has to teach at least one course and at most three courses***

$CourseLecturer \in COURSES \longrightarrow LECTURERS \wedge \boldsymbol{ran}(CourseLecturer) = LECTURERS$

$\wedge (\forall\, l.\ \boldsymbol{card}(CourseLecturer \rhd \{l\}) \leq 3))$

Comment on Initialisation event

MACHINE *CoursesRegistration_m0*
SEES *CoursesRegistration_m0*
VARIABLES *access*
INVARIANTS
    inv1: $CourseLecturer \in COURSES \longrightarrow LECTURERS$
    ….
EVENTS
  INITIALISATION ≙
    begin
        act1: $CourseLecturer := \emptyset$ *// wrong! Since* $CourseLecturer$ *defined as a total function*
    end

inv1 invariant should be preserved upon **INITIALISATION** event.

BUT Rodin prover will fail to prove that since upon substitution $CourseLecturer$ by ∅, it will have to prove that $\emptyset \in COURSES \longrightarrow LECTURERS$. But it is wrong!

Simple example: seat booking system

The system allows a person to make a seat booking. Specifically:

- A system should support booking a seat by only one person;
- A system should support cancelling of a booking.

Modelling seat booking system in Event-B

- In the static part of our Event-B model – context - we will introduce required sets: *SEATS* and *PERSONS* as well as required axioms.
- In the dynamic part of the model – machine – we will define (specify) operations by events **BOOK** and **CANCEL**, correspondingly.
- We introduce a variable ***booked_seats*** whose type is a *partial function* on the sets *SEATS* and *PERSONS.*
- ***booked_seats*** keeps track on booked seats and persons make their booking.
- Since booking of a seat can be done or cancelled, the variable ***booked_seats*** will be updated by the events **BOOK** or **CANCEL** to reflect this.

Seat booking system

We define a context **BookingSeats_c0** as follows

CONTEXT
  BookingSeats_c0
SETS
  PERSONS
  SEATS
AXIOMS
  axm1: finite(SEATS)
  axm2**:** finite(PERSONS)
  axm3: SEATS$\neq \emptyset$
  axm4: PERSONS$\neq \emptyset$
END

Machine **BookingSeats_m0**

```
MACHINE  BookingSeats_m0
SEES BookingSeats_c0
VARIABLES
	booked_seat
INVARIANTS
	inv1: booked_seat ∈ SEATS ⇸ PERSONS
// this variable is defined as a partial function (every seat can be
occupied by only one person, but not every seat from the set SEATS
is booked yet)
EVENTS
INITIALISATION ≙
	then
		act1: booked_seat := ∅  // empty set
	end
BOOK ≙		//booking a seat
	any person  seat
	where
		grd1: person ∈ PERSONS	// take any person
		grd2: seat ∈ SEATS		// we take any seat ...
		grd3: seat ∉ dom(booked_seat)  // ... that is free
	then
		act1: booked_seat := booked_seat ∪ {seat ↦ person}
	end

CANCEL ≙		// cancelation of booking
	any person seat
	where
		grd1: seat ↦ person ∈ booked_seat	// any pair
							from booked_seat
	then
		act1: booked_seat := booked_seat \ {seat ↦ person}
		// delete this pair from booked_seat
	end
END
```

Model development with Event-B

- Event-B allows models to be developed gradually via mechanisms such as ***context extension*** and ***machine refinement***.
- These techniques enable users to develop target systems from their abstract specifications, and subsequently introduce more implementation details.
- More importantly, properties that are proved at the abstract level are maintained through refinement, and hence are also guaranteed to be satisfied by later refinements.
- As a result, correctness proofs of systems are broken down and distributed amongst different levels of abstraction, which is easier to manage.

A course management system: Requirements

- A club has some fixed *members*; amongst them are *instructors* and *participants*.
- A member can be both an instructor and a participant.

REQ1	**Instructors are members of the club.**

REQ2	**Participants are members of the club.**

A course management system (cont.)

- There are predefined *courses* that can be offered by a club.
- Each course is associated with exactly one fixed instructor.

REQ3	**There are predefined courses.**

REQ4	**Each course is assigned to one fixed instructor.**

A course management system (cont.)

A course is either *opened* or *closed* and is managed by the system.

REQ5	**A course is either *opened* or *closed*.**

REQ6	**The system allows a closed course to be opened.**

REQ7	**The system allows an opened course to be closed.**

A course management system (cont.)

The number of opened courses is limited.

REQ8	**The number of opened courses cannot exceed a given limit.**

Only when a course is opened, can participants *register* for the course. An important constraint for registration is that an instructor cannot attend his own courses.

REQ9	**Participants can only register for an opened course.**

REQ10	**Instructors cannot attend their own courses.**

A course management system: development with Event-B

- Next, we will develop a formal model based on the above requirements document:
  - we will refer to the above requirements in order to justify how they are formalised in the Event-B model.
- In the initial model, we focus on opening and closing of courses by the system.
- We start our modelling with defining a context **Courses_c0.**

A course management system: Context

CONTEXT **Courses_c0**
SETS *COURSES* // a carrier set *COURCES* denoting the set of courses that can be offered by the club (REQ3)
CONSTANTS *m* // REQ8: the maximum number of courses that the club can open
AXIOMS

axm0_1: **finite**(*COURSES*)
axm0_2: $m \in \mathbb{N}$
axm0_3: $m > 0$
axm0_4: **card**(*COURSES*) $\geq m$ // number of all possible courses is no less than *m*

END

A course management system: Machine

- We develop machine **Courses_m0** of the initial model, focusing on courses opening and closing.
  - This machine sees context **Courses_c0** developed before.
- We model the set of opened courses by a variable *courses*

```
MACHINE Courses_m0
SEES Courses_c0
VARIABLES  courses   // The machine state is represented by the variable, courses,
                                          denoting the open courses
 INVARIANTS
          inv0_1: courses ⊆ COURCES  // open courses is a subset of all available courses
          inv0_2: card(courses) ≤ m
EVENTS
INITIALISATION ≙
          then
             act1: courses:= ∅  // Initially, all courses are closed
                                   (so, the set of opened courses is set to the empty set);
         end
…
```

A course management system: Machine

- We model the opening and closing of courses using two events **OPENCOURSE** and **CLOSECOURSE:**

OPENCOURSE $\triangleq$ // **REQ6:** The system allows a closed course to be opened.

    any *crs*

    where

        grd1: $\mathbf{card}(courses) < m$ // the current number of opened courses has not yet reached the limit

        grd2: $crs \notin courses$ // a course *crs* is not opened yet

    then

        act1: $courses := courses \cup \{crs\}$ // add *crs* course to the set *courses*

    end

CLOSECOURSE $\triangleq$ // **REQ7:** The system allows an opened course to be closed.

    any *crs*

    where

        grd1: $crs \in courses$ // the course crs has been opened before

    then

        act1: $courses := courses \setminus \{crs\}$ // remove *crs* from the set *courses*

    end

Course Management System: refinement

We extend context **Courses_c0** by the context **Members_c1**

CONTEXT **Members_c1** **EXTENDS** **Courses_c0**
SETS *MEMBERS* // a carrier set *MEMBERS* represents the set of club members

CONSTANTS *PARTICIPANTS* // constant *PARTICIPANTS* denotes the set of participants
INSTRUCTORS // constant *INSTRUCTORS* denotes the set of instructors
courseInstructor // constant models a relationship between courses and instructors
AXIOMS
axm1_1: **finite**(*MEMBERS*)
axm1_2: $PARTICIPANTS \subseteq MEMBERS$ // participants must be members of the club
axm1_3: $INSTRUCTORS \subseteq MEMBERS$ // instructors must be members of the club
axm1_4: $courseInstructor \in COURSES \rightarrow INSTRUCTORS$ // a total function from *COURSES* to *INSTRUCTORS* (*thus we* formalise REQ4)
END

Machine Refinement

- *Machine refinement* is a mechanism for introducing details about the dynamic properties of a model
  - When speaking about machine N refining another machine M, we refer to M as the *abstract* machine and to N as the *concrete* machine.
- Two kinds of refinement: ***superposition refinement*** and ***data refinement***
  - In superposition refinement, the abstract variables of M are retained in the concrete machine N, with possibly some additional concrete variables.
  - In data refinement, the abstract variables *v* are replaced by concrete variables *w* and, subsequently, the connections between M and N are represented by the relationship between *v* and *w*.
  - Often, Event- B refinement is a mixture of both superposition and data refinement: some of the abstract variables are retained, while others are replaced by new concrete variables.

Superposition Refinement

- In superposition refinement, variables *v* of the abstract machine M are kept in the refinement, i.e. as part of the state of N.
- N can have some additional variables *w*.
- The concrete invariants ***J*(*v*,*w*)** specify the relationship between the old and new variables.
- Each abstract event e is refined by a concrete event f
- Assume that the abstract event e and the concrete event f are as follows:

e = any *x* where *G*(*x*, *v*) then *Q*(*x*, *v*) end

f = any *x* where *H*(*x*,*v*,*w*) then *R*(*x*,*v*,*w*) end

- **f refines *e*** if the guard of f is stronger than that of e (*guard strengthening*), concrete invariants *J* are maintained by f, and abstract action *Q* simulates the concrete action *R* (*simulation*).

Superposition Refinement

- In the course of refinement, *new events* are often introduced into a model.
- Lets go back to our *Course Management System*…

Refinement of a machine **Courses_m0**

MACHINE Courses_m0
SEES Courses_c0
VARIABLES *courses*
INVARIANTS
inv0_1: $courses \subseteq COURCES$
inv0_2: $\mathbf{card}(courses) \leq m$
EVENTS
INITIALISATION ≙ …
OPENCOURSE ≙ …
CLOSECOURSE ≙ …

we had before ….

Courses_m0 machine is refined by a machine **Members_m1**

MACHINE Members_m1
REFINES Courses_m0
SEES Members_c1
VARIABLES *courses participants*
INVARIANTS
inv1_1: $participants \in courses \leftrightarrow PARTICIPANTS$
inv1_2: $\forall c.\ c \in courses \Rightarrow courseInstructor(c) \notin participants[\{c\}]$
EVENTS
INITIALISATION ≙
then
…
act2: $participants := \emptyset$ *// The variable is initialised to the empty set.*
end

- New variable ***participants*** representing information about course participants (modelled as a relation between the sets of open courses ***courses*** *and the set PARTICIPANTS*)
- Invariant ***inv1_2:*** $\forall c.\ c \in courses \Rightarrow courseInstructor(c) \notin participants[\{c\}]$ states that "*for every opened course c, the instructor of this course is not amongst its participants* " ***(REQ10)***

Modelling machine **Members_m1**

The original abstract event **OPENCOURSE** stays unchanged in this refinement, while an additional assignment is added to **CLOSECOURSE** to update ***participants*** by removing the information about a closing course *crs* from it.

OPENCOURSE refines **OPENCOURSE** ≜ // no changes in this event
  any *crs*
  where
    grd1: **card**(*courses*) < *m*
    grd2: $crs \notin courses$
  then
    act1: $courses := courses \cup \{crs\}$
  end
CLOSECOURSE refines **CLOSECOURSE** ≜ // we add in to the event an additional action
  any *crs*
  where
    grd1: $crs \in courses$
  then
    act1: $courses := courses \setminus \{crs\}$
    act2: $participants := \{crs\} \lhd participants$ // *removing all the relationships between this course and its participants.*
end

Machine **Members_m1**

- A new event **REGISTER** is added. It models the registration of a participant *p* for an opened course *c*.
- The guard of the event ensures that *p* is not the instructor of the course (***grd1_3***) and is not yet registered for the course (***grd1_4***).
- The action of the event updates ***participants*** accordingly by adding the mapping $\boldsymbol{c \mapsto p}$ to it.

```
REGISTER ≙   //  the registration of a participant p for an opened course c
        any  p  c
        where
                grd1_1: c ∈ courses
                grd1_2: p ∈ PARTICIPANTS
                grd1_3: p≠CourseInstructor(c)      // p is not the instructor of the course
                grd1_4: c ↦p ∉ participants        // p is not yet registered for the course
        then
                act1: participants  := participants  ∪ {c ↦p}  // adding all the relationships between this
                                                                  course and its participants.
        end
```

Data Refinement

- In data refinement, abstract variables *v* are removed and replaced by concrete variables *w*.
- The states of abstract machine M are related to the states of concrete machine N by ***gluing invariants*** *J*(*v*, *w*).
- In Event-B, the gluing invariants *J* are declared as invariants of N and also contain the *local* concrete invariants, i.e., those constraining only concrete variables *w*.

- Coming back to the *Course Management System*…

Data refinement of **Members_m1** machine

- We perform a data refinement by replacing abstract variables ***courses*** and ***participants*** by a new concrete variable ***attendants***:

inv2_1: $attendants \in COURSES \nrightarrow \mathbb{P}(PATICIPANTS)$

- is a *partial function* from *COURSES* to some set of participants.

The following invariants at as gluing invariants, linking abstract variables ***courses*** and ***participants*** with concrete variable ***attendants***

inv2_2: *courses* = **dom**(*attendants*)

inv2_3: $\forall c.\ c \in courses \Longrightarrow participants[\{c\}] = attendants(c)$ // for every opened course *c*, the set of participants attending that course represented abstractly as *participants*[{*c*}] is the same as *attendants(c)*.

Members_m2 machine

MACHINE **Members_m2**
REFINES **Members_m1**
SEES **Members_c1**
VARIABLES *attendants*
INVARIANTS
inv2_1: $attendants \in COURSES \rightarrowtail\!\!\!\!\!\!\!+\ \mathbb{P}(PATICIPANTS)$
inv2_2: $courses = \mathbf{dom}(attendants)$
inv2_3: $\forall c.\ c \in courses \Longrightarrow participants[\{c\}] = attendants(c)$
EVENTS
…

Refinement of **OPENCOURSE** event

MACHINE Members_m1 REFINES Courses_m0	MACHINE Members_m2 REFINES Members_m1
….	…
OPENCOURSE refines **OPENCOURSE** ≙	**OPENCOURSE_new** refines **OPENCOURSE** ≙
any *crs*	**any** *crs*
where	**where**
grd1: **card**(*courses*) $<$ *m*	***grd2_1***: *crs* ∉ *dom(attendants)*
grd2: *crs* ∉ *courses*	***grd2_2***: **card**(*attendants*) ≠ *m*
then	**then**
act1: *courses* := *courses* ∪ {*crs*}	***act1***: *attendants(crc)* := ∅
end	**end**

we had before ….

now

- The concrete guards ensure that *crs* is a closed course and the number of opened courses (**card(***attendants***)**) has not reached the limit *m*.
- The action of **OPENCOURSE_new** sets the initial participants for the newly opened course *crs* to be the empty set.

Refinement of **CLOSECOURSE** event

- Abstract event **CLOSECOURSE** is refined by concrete event **CLOSECOURSE_new**, where one course ***crs*** is closed at a time. The guard and action of concrete event **CLOSECOURSE_new** are as expected:

MACHINE Members_m1 **REFINES** Courses_m0
….
CLOSECOURSE refines **CLOSECOURSE** ≙
    any *crs*
    where
        grd1: $crs \in courses$
    then
        act1: $courses := courses \setminus \{crs\}$
        act2: $participants := \{crs\} \lhd participants$
    end

we had before ….

MACHINE Members_m2 **REFINES** Members_m1
….
CLOSECOURSE_new refines **CLOSECOURSE** ≙
    any *crs*
    where
        grd1: $crs \in dom(attendants)$
    then
        act1: $attendants := \{crs\} \lhd attendants$
    end

now

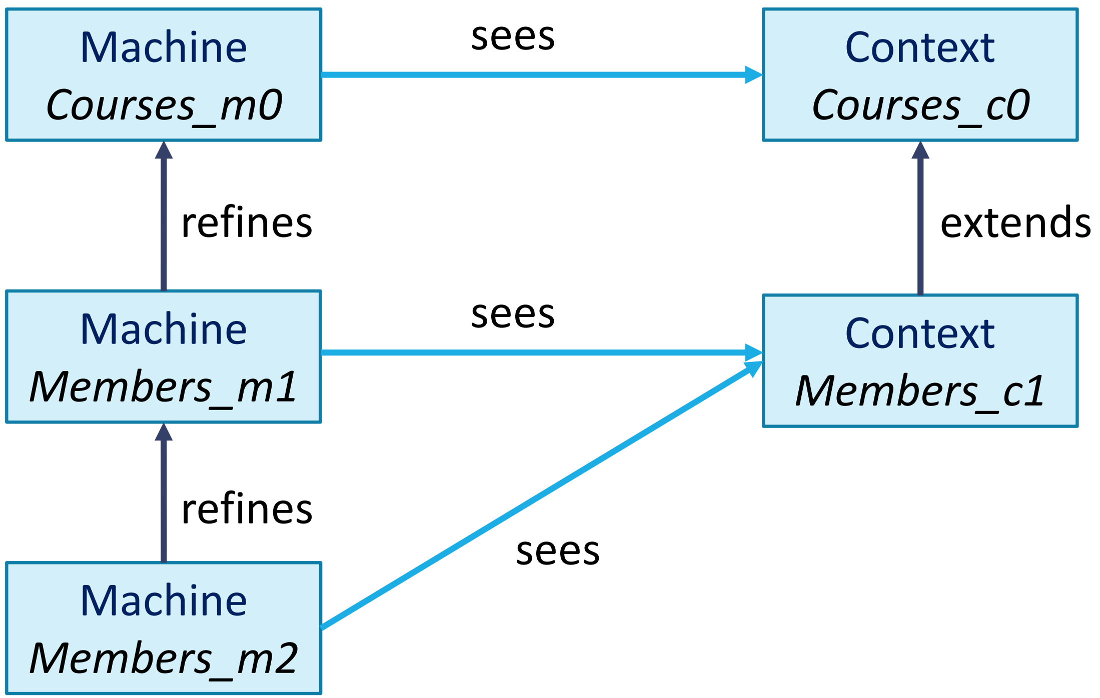
Refinement of **REGISTER** event

MACHINE Members_m1 **REFINES** Courses_m0
…
REGISTER ≙
    any p c
    where
        grd1_1: $c \in courses$
        grd1_2: $p \in PARTICIPANTS$
        grd1_3: $p \neq CourseInstructor(c)$
        grd1_4: $c \mapsto p \notin participants$
    then
        act1: $participants := participants \cup \{c \mapsto p\}$
end

MACHINE Members_m2 **REFINES** Members_m1
…
REGISTER_new refines **REGISTER** ≙
    any p c
    where
        grd2_1: $c \in dom(attendants)$
        grd2_2: $p \in PARTICIPANTS$
        grd2_3: $p \neq CourseInstructor(c)$
        grd2_4: $p \notin attendants(c)$
    then
        act1: $attendants(c) := attendants(c) \cup \{p\}$
end

Summary of the development

The hierarchy of the development:



Requirements tracing:

REQ id	Models
REQ1	*Members_c1*
REQ2	*Members_c1*
REQ3	*Courses_c0*
REQ4	*Members_c1*
REQ5	*Courses_m0*
REQ6	*Courses_m0*
REQ7	*Courses_m0*
REQ8	*Courses_m0*
REQ9	*Members_m1*
REQ10	*Members_m1*

Summary

We studied how to use different mathematical concepts (sets, functions, relations) and operations over them to specify behaviour of safety-critical systems and systems that require modelling some access rights

The main verification technique was proof of the invariant preservation

This is important for the verification of safety and preservation of access control restrictions

However, dealing with liveness (progress) properties is harder in Event-B while model checking is great in this.