Modularisation Plugin: Parking Lot Case Study

Alexei Iliasov

Newcastle University

About this presentation

This is training material on using the modularisation plugin for structuring Event B models within the Rodin Platform. It does not show how to build Event B models and focuses on the details pertaining to the use of modularisation.

Contents

Installation

The plugin works with the platform version 1.1RC1 or higher.
The modularisation plugin is installed as follows:

- go to Install New Software
- in the software sites, select *Modularisation*
- check and click to install

Alternatively,

- click Add Site, the site url is `http://iliasov.org/modplugin`
- then proceed as above

What the plugin does

- The plugin extends the Event B modelling language with the concept of a module
- A module is a parametrised Event B development associated with a module **interface**
- An interface defines a number of **operations**
- A specification is decomposed by including a module in a machine and connecting the two using operation calls and gluing invariants

What the plugin provides

- a new type of Event B component - a module interface (editor, pretty-printer and proof obligations generator)
- new machine constructs:**IMPLEMENTS** and **USES**
- new event attributes: **group** and **final**
- the ability to write operation calls in event actions
- additional proof obligations for operation calls
- additional proof obligations for implementation machines

Parking Lot

A popular parking lot requires an access control and payment collection mechanisms. The following main requirements were identified:

1. no car may enter when there is no space left in the parking lot
2. a fare must be paid when a car leaves the parking lot
3. each time a car leaves the parking lot, the fare to be paid is determined by multiplying the total length of stay since the midnight (that is, including any previous stay(s)) by the cost of parking per unit of time
4. the amount paid in any single transaction is capped
5. at midnight, the accumulated parking time of all cars is reset to zero

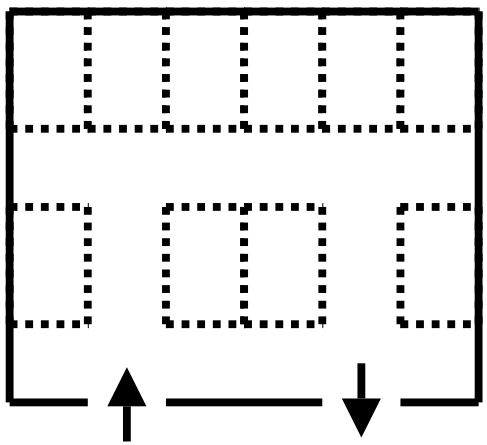
Parking Lot

Solution overview:

1. two gates are placed to control entry and exit
2. a payment collection machine is placed near to the exit gate in such a manner that a driver may use it before going through the exit gate
3. the exit gate does not open until the full payment is collected
4. the entrance gate does not open if the car park is full

Abstract model

the initial model describes the phenomena of cars entering and leaving the parking lot. It addresses the capacity restrictions although without exhibiting a concrete mechanism for controlling the number of cars entering the parking lot.



Model variables

- LOT_SIZE - the parking lot capacity (constant)
- $entered$ - the number of cars that have entered the parking lot
- $left$ - the number of cars that have left the parking lot
- hence, $left - entered$ is the current number of cars in the parking lot

INVARIANT

$$entered \in \mathbb{N}$$
$$left \in \mathbb{N}$$
$$entered - left \in 0 \ldots LOT\_SIZE$$

Model events

a new car appears:

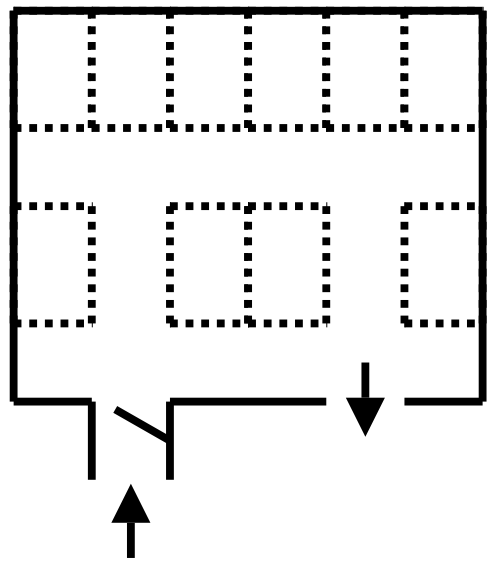
enter $=$ WHEN
$entered - left < LOT\_SIZE$
THEN
$entered := entered + 1$
END

a car leaves:

leave $=$ WHEN
$entered - left > 0$
THEN
$left := left + 1$
END

First refinement

In the first refinement the entrance is controlled by a gate. The the gate prevents a car from entering when there is no free space and also records the registration plate of an entering car.



Gate Module

The logic controlling a gate is easily decoupled from the main model. We decompose the model into the controller part and an entry gate

The first step of this decomposition is to define a gate module interface.

Gate variables

- *CAR* - car id (registration plate)
- *mcars* - the number of cars that has passed through the gate
- *current* - the id of the car in the front of the gate

INVARIANT

$mcars \in \mathbb{N}$

$current \in CAR$

Gate operations

when there is no car in front of the gate, a driver may press the gate button to try to open the gate:

$$
\begin{array}{lll}
\mathsf{carid} \leftarrow \mathsf{Button} & = & \mathrm{PRE} \\
 & & \quad current = empty \\
 & & \mathrm{POST} \\
 & & \quad current' \in CAR \setminus \{empty\} \\
 & & \quad carid' = current \\
 & & \mathrm{END}
\end{array}
$$

Gate operations

the car park controller orders the gate to open; the gate has sensors to observe whether the car has moved through the gate ($moved = \mathrm{TRUE}$) or stayed in front of the gate:

$$
\begin{array}{lll}
\mathsf{moved} \leftarrow \mathsf{OpenGate} & = & \mathrm{PRE} \\
 & & \quad current \neq empty \\
 & & \mathrm{POST} \\
 & & \quad (moved' = \mathrm{TRUE} \wedge mcars' = mcars + 1 \wedge \\
 & & \quad\quad current' = empty) \vee \\
 & & \quad (moved' = \mathrm{FALSE} \wedge mcars' = mcars \wedge \\
 & & \quad\quad current' = current) \\
 & & \mathrm{END}
\end{array}
$$

Gate operations

predicate $mcars' = mcars \wedge current' = current$ in

$$\begin{array}{l}(moved' = \mathrm{TRUE} \wedge mcars' = mcars + 1 \wedge current' = empty) \vee \\ (moved' = \mathrm{FALSE} \wedge mcars' = mcars \wedge current' = current)\end{array}$$

is necessary to indicate that *mcars* and *current* remain unchanged in the second branch of the post-condition. This is only required when a disjunction is used and not all variables are assigned new values in the disjunction branches

Operating the gate

to open the gate and let a car through it, the following has to happen:

- a driver must press the gate button (operation *Button*)
- the controller must activate the gate (operation *OpenGate*)

in our model, the main development models both driver's and controller's behaviour

First refinement machine

The first refinement imports the gate module interface. Prefix entry is used to avoid name clashes (with another gate added later on).

When a prefixed interface is imported, all its constants and sets appear prefixed in the importing context. This is not always convenient. We use type instantiation to replace the type of an imported module by a typing expression known in the importing context. We also define a property (an axiom) that equates a prefixed and unprefixed versions of constant *empty*.

USES entry : ParkingGate

TYPES

$entry\_CAR \mapsto CAR$

PROPERTIES

$entry\_empty = empty$

First refinement machine

two new variables are defined in the refinement machine. They help to link the states of the controller and the entry gate.

- *incar* - the id of an entering car
- *inmoved* - a flag indicating whether a car has passed through the (open) entry gate

INVARIANT

$$incar \in CAR$$

$$inmoved \in BOOL$$

Import invariant

it is necessary to provide an invariant relating the states of an imported module and the importing machine (import invariant)

without this, a module import does not make much sense as an overall model would be composed of two independently evolving systems

Import invariants

when there is no car at the gate, the gate car counter has the same value as the controller counter:

$$\boxed{\mathit{inmoved} = \mathrm{FALSE} \implies \mathit{entered} = \mathit{entry\_mcars}}$$

when a car is passing through the entrance gate, only the gate counter has been incremented:

$$\boxed{\mathit{inmoved} = \mathrm{TRUE} \implies \mathit{entered} + 1 = \mathit{entry\_mcars}}$$

Import invariants

when a car is passing through the gate there must be no other car at the gate:

$$inmoved = \mathrm{TRUE} \implies entry\_current = empty$$

when a car is coming through the entrance gate there is certainly free space in the parking lot:

$$inmoved = \mathrm{TRUE} \wedge entry\_current \neq empty \implies entered - left < LOT\_SIZE$$

Model events

a driver presses the gate button at the entrance gate (new event):

UserPressButton $=$ WHEN
$entered - left < LOT\_SIZE$
$entry\_current = empty$
$inmoved = FALSE$
THEN
$incar :=$ **entry_Button**
END

here **entry_Button** is a call of the *Button* operation from the *entry* module.

Model events

the parking lot controller orders the gate to open (new event):

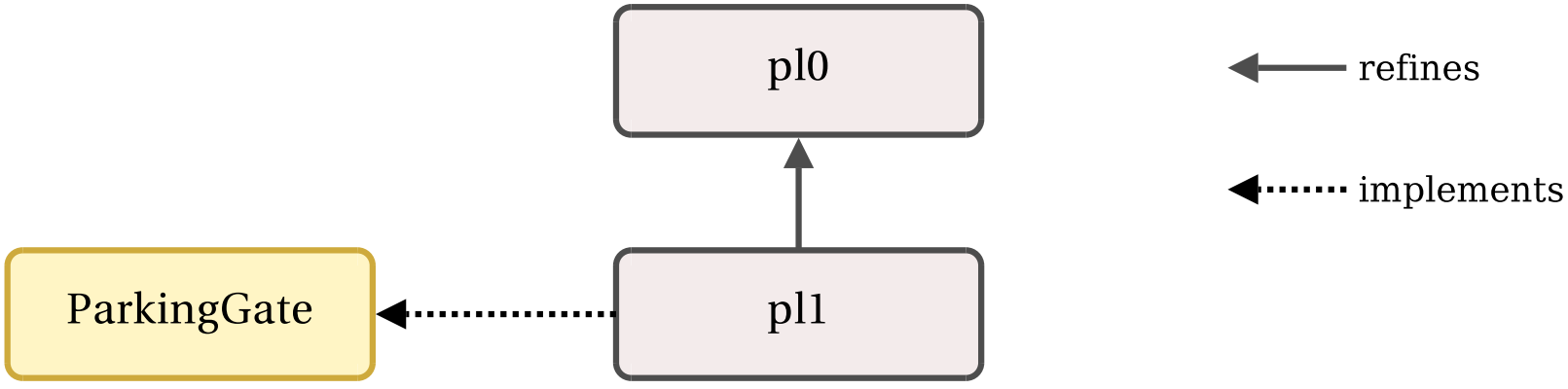
CtrlOpenGate $=$
WHEN
entry_current $\neq$ *empty* $\wedge$ *inmoved* $=$ FALSE
THEN
inmoved $:=$ **entry_OpenGate**
END

Model events

finally, the *enter* event is refined to reflect the model changes:

enter $=$
WHEN
$\quad$ *inmoved* $= \mathrm{TRUE}$
THEN
$\quad$ *entered* $:=$ *entered* $+ 1$
$\quad$ *inmoved* $:= \mathrm{FALSE}$
END

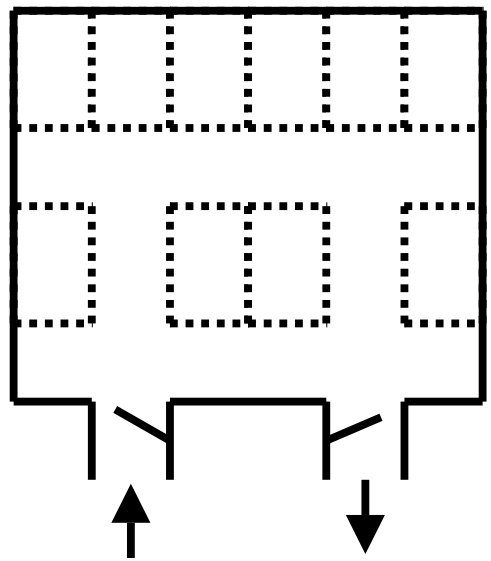
Development structure



all proof obligations are discharged automatically (18 total)

Second refinement

the second refinement is very similar: we add another gate - an exit gate. the same module is imported with a new prefix to obtain two separate modules modelling two gates.



Second refinement machine

two new variables are defined :

- *outcar* - the id of the leaving car
- *outmoved* - the flag indicating whether a leaving car has passed through the (open) exit gate

INVARIANT

$outcar \in CAR$

$outmoved \in BOOL$

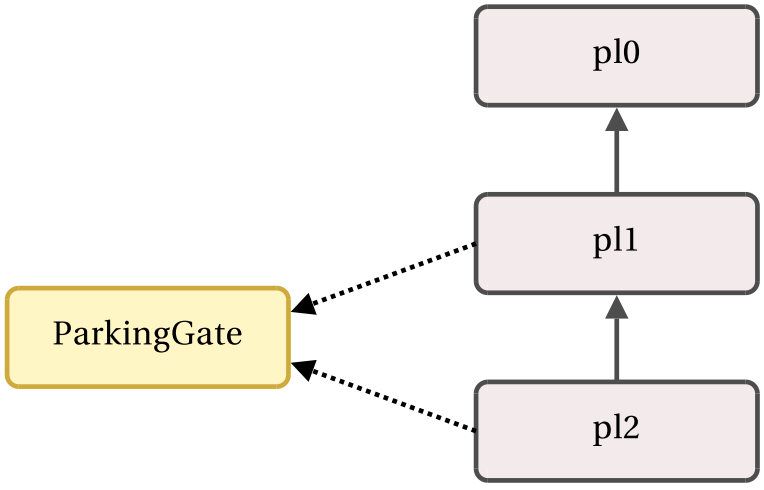
Import invariant

in addition to the conditions relating the variables of the exit gate module with the variables of the main machine (the controller) we are also able to specify a link between the states of the two gates

when the gates are closed, the number of cars entered through the entry gate minus the number of cars left via the exit gate may not be less than zero and is not greater than the parking lot capacity:

$$\begin{array}{l} \mathit{inmoved} = \mathrm{FALSE} \wedge \mathit{outmoved} = \mathrm{FALSE} \implies \\ \qquad \mathit{entry\_mcars} - \mathit{exit\_mcars} \in 0 \ldots \mathit{LOT\_SIZE} \end{array}$$

Development structure

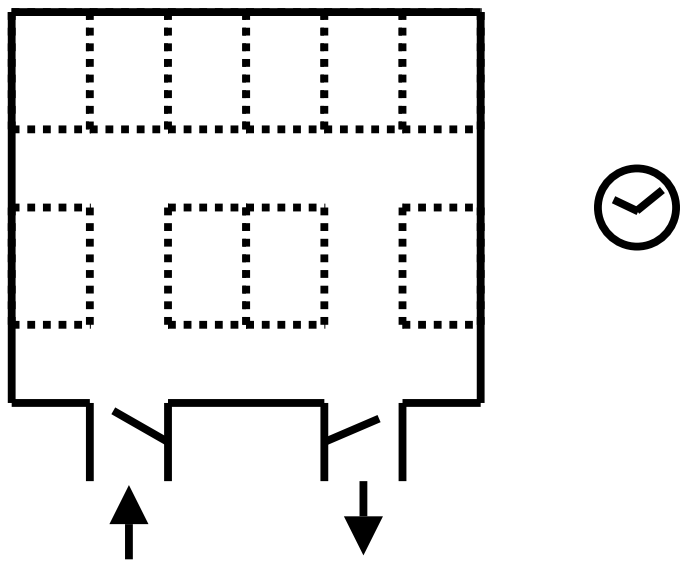


one interactive proof (17 total)

Third refinement

the third refinement step is concerned with keeping the record of car stays; this step introduces the notion of time

the definition of time will be used more than once and thus it is convenient to place in an interface



Clock interface

the clock interface models the progress of time; the following is taken as the definition of time:

- time value increase is monotonic
- time changes in discrete increments when it is observed

this reflects our modelling approach to time; at an implementation stage it may have to be mapped onto a differing concept of time progress

Clock constants and variables

- *from* - the lowest time value (constant)
- *to* - the highest time value (constant)
- *delta* - the smallest observable time increment (constant)
- *prev* - the last reading of the clock (variable)

AXIOMS

$$to \in \mathbb{N} \land from \in \mathbb{N}$$
$$to - from \in delta$$
$$delta > 0$$

INVARIANT

$$prev \in from \mathinner{..} to$$

Clock operations

the time progress is observed and the current time value is returned:

$$
\begin{array}{lll}
\mathsf{t} \leftarrow \mathsf{currentTime} & = & \text{PRE} \\
 & & \quad \text{TRUE} \\
 & & \text{POST} \\
 & & \quad \mathit{prev}' \in \mathit{from} \ldots \mathit{to} \\
 & & \quad \mathit{prev}' \geq \mathit{prev} + \mathit{delta} \wedge \mathit{prev}' = \mathit{to} \wedge \mathit{t}' = \mathit{prev}' \\
 & & \text{END}
\end{array}
$$

the clock is reset:

$$
\begin{array}{lll}
\mathsf{clockReset} & = & \text{PRE} \\
 & & \quad \text{TRUE} \\
 & & \text{POST} \\
 & & \quad \mathit{prev}' = \mathit{from} \\
 & & \text{END}
\end{array}
$$

Third refinement machine

constant definitions:

- *TOD* - the time-of-the-day type
- *DAY_START* - day start time value
- *DAY_END* - day end time value

AXIOMS

$$TOD = DAY\_START \ldots DAY\_END$$
$$DAY\_START \in \mathbb{N}$$
$$DAY\_END \in \mathbb{N}$$
$$DAY\_END > DAY\_START$$

Third refinement machine

new variables:

- *register* - function recording the time when a car enters the parking lot
- *cartime* - for a given car gives the accumulated stay time since the midnight

INVARIANT

$$register : CAR \pfun TOD$$
$$cartime : CAR \fun \mathbb{N}$$

INITIALISATION

$$register := \varnothing$$
$$cartime := CAR \times \{0\}$$

the time spent in the parking lot since the midnight is no greater than the latest registration time:

$$\forall x \cdot x \in dom(register) \implies register(x) - DAY\_START \geq cartime(x)$$

Clock module import

the clock module is imported without a prefix; the time limits are set to correspond to the *TOD* data type:

USES Clock
PROPERTIES
from = *DAY_START*
to = *DAY_END*

Import invariants

all the car registration timestamps have the time value not exceeding the current time:

$$\forall x \cdot x \in dom(register) \implies register(x) \leq prev$$

the time a car has spent in the park since the midnight is not more than the time elapsed since the midnight:

$$\forall x \cdot x \in CAR \implies cartime(x) \leq prev - DAY\_START$$

Model events

a record is made of the time when a car enters the car park:

```
enter  =  EXTENDS enter
          BEGIN
              register(incar) := currentTime
          END
```

currentTime is an operation call returning (and also advancing) the current time

Model events

when a car leaves, the registration record is removed and the total stay time is updated:

CtrlOpenGateL = EXTENDS CtrlOpenGateL
WHEN
$outcar \in dom(register)$
THEN
$cartime(outcar) := cartime(outcar) + (\mathbf{currentTime} - register(outcar))$
$register := outcar \lhd register$
END

here $\mathbf{currentTime} - register(outcar)$ is the length of the current stay of the leaving car *outcar*

Model events

according to the requirements, upon midnight, the car stay times and registration timestamps are reset:

RegisterReset $=$
WHEN
$prev = DAY\_END$
THEN
clockReset
$register := dom(register) \times \{DAY\_START\}$
$cartime := CAR \times \{0\}$
END

operation **clockReset** sets the clock reading *prev* to the start of a day time value *DAY_START*

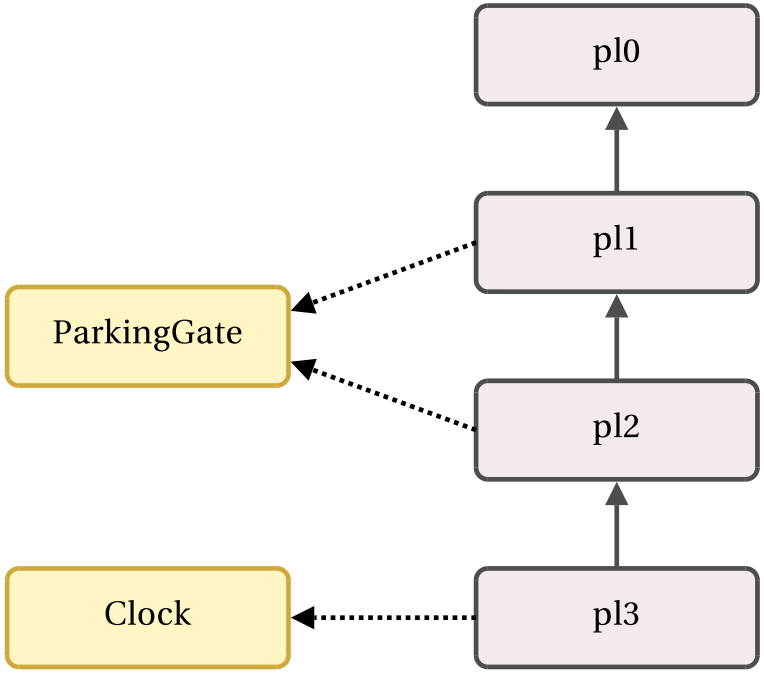
Model events

to make sure that clock and register resets happen even when there are no cars entering or leaving the parking lot, the system must actively observe time

ObserveTime	=	BEGIN **currentTime** END

this event forces the progress of time even if no other time-related activity takes place

Development structure

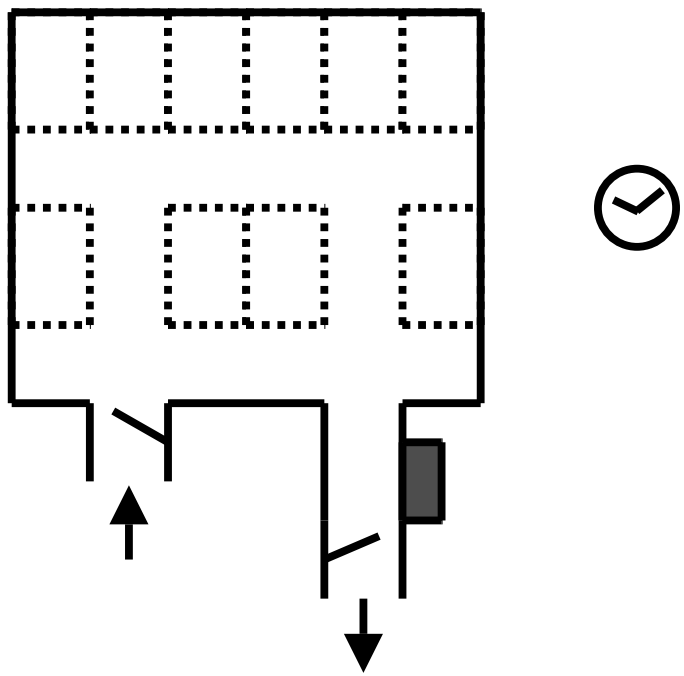


five interactive proofs (30 total)

Fourth refinement

in this step, before a car may leave, the car driver must pay the amount determined by the length of stay since the midnight

the functionality of a device collecting payment is decoupled from the controller logic and is placed in a separate module



Payment machine constants and variables

- *payPerTimeUnit* - the cost of unit of time in the parking lot (constant)
- *maxPay* - the limit on the amount paid in a single transaction (constant)
- *balance* - the outstanding balance to be paid (variable)

AXIOMS

$payPerTimeUnit \in \mathbb{N}$

$maxPay \in nat$

INVARIANT

$balance \in \mathbb{N}$

Payment machine operations

the payment machine is configured by supplying the accumulated length of stay as a parameter; the operation computes the balance to be paid:

$$
\begin{array}{lcl}
\text{Configure} & = & \text{ANY } \mathit{stay} \text{ PRE} \\
& & \quad \mathit{stay} \in \mathbb{N} \\
& & \quad \mathit{balance} = 0 \\
& & \text{POST} \\
& & \quad \mathit{balance}' = \mathit{min}(\mathit{stay}\mathit{payPerTimeUnit}, \mathit{maxPay}) \\
& & \text{END}
\end{array}
$$

the payment is taken from a driver; the amount paid is at least as large as the outstanding balance (i.e., a driver may overpay but not underpay):

$$
\begin{array}{lcl}
\text{Pay} & = & \text{PRE} \\
& & \quad \mathit{balance} \neq 0 \\
& & \text{POST} \\
& & \quad p' \in \mathbb{N} \land p' \geq \mathit{balance} \land \mathit{balance}' = 0 \\
& & \text{END}
\end{array}
$$

Fourth refinement machine

there are two new variables used to constrain the ordering of concrete events:

- *confPay* - a flag indicating the configuration phase of payment collection
- *paid* - a flag indicating that payment has been collected

INVARIANT

$$confPay \in BOOL$$
$$cpayed \in BOOL$$

Import invariants

when there is no car at the exit gate the pay machine balance is zero:

$$exit\_current = empty \vee confPay = \mathrm{TRUE} \implies pm\_balance = 0$$

during payment configuration there is always a car at the exit gate:

$$confPay = \mathrm{TRUE} \implies exit\_current \neq empty$$

Model events

the controller configures the payment machine by calling the *Configure* operation with the accumulated stay time:

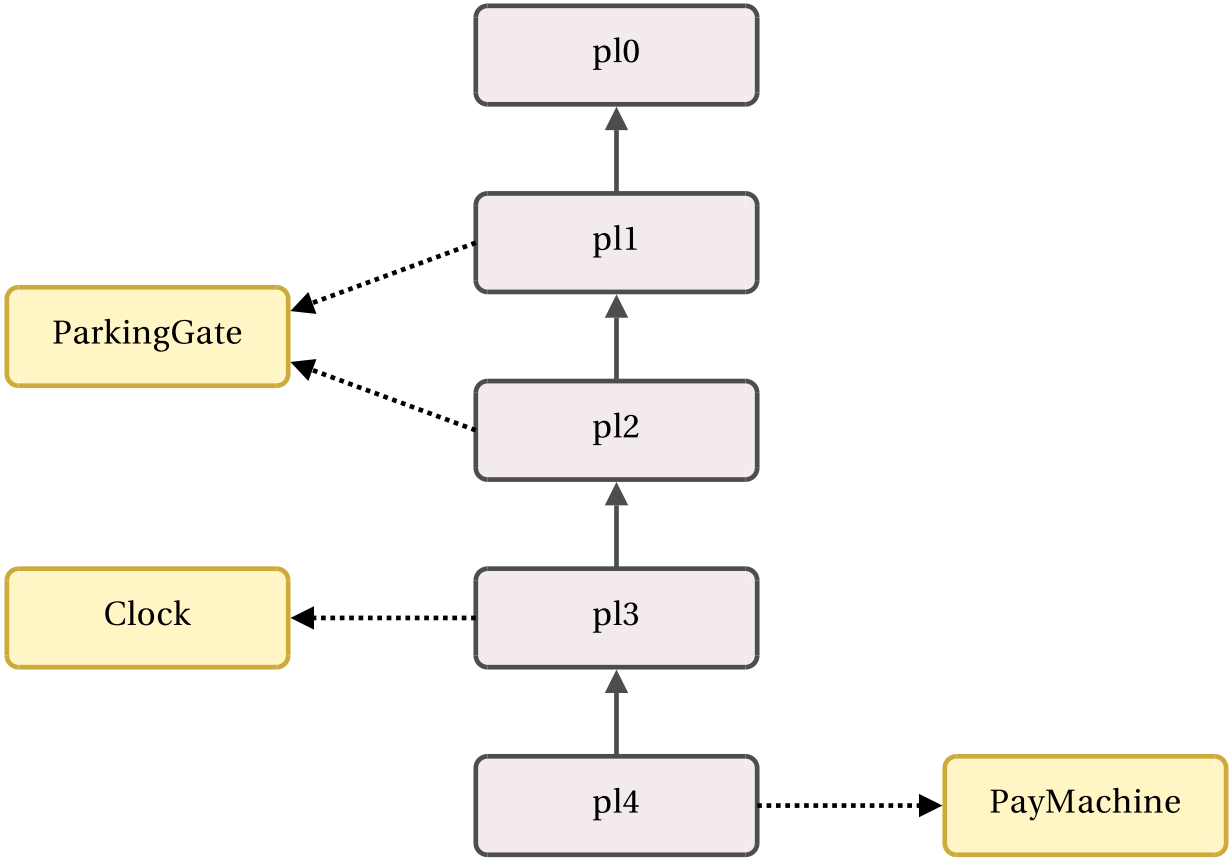
CtrlPay = WHEN
confPay = TRUE
THEN
pm_void := **pm_Configure**(*cartime*(*outcar*))
confPay := FALSE
END

Model events

a driver pays if there is an outstanding balance to be paid (always the case with the current payment machine and clock interfaces):

$$
\begin{array}{rcl}
\text{UserPay} & = & \text{WHEN} \\
 & & \quad \cdots \wedge \mathit{pm\_balance} > 0 \\
 & & \text{THEN} \\
 & & \quad \mathit{payed} := \mathit{TRUE} \\
 & & \quad \textbf{pm\_Pay} \\
 & & \text{END} \\
\text{UserNoPay} & = & \text{WHEN} \\
 & & \quad \cdots \wedge \mathit{pm\_balance} = 0 \\
 & & \text{THEN} \\
 & & \quad \mathit{payed} := \mathit{TRUE} \\
 & & \text{END}
\end{array}
$$

Development structure



all proof obligations are discharged automatically (34 total)

Implementing Modules

The development relies on three modules that are so far defined only by their interfaces. To complete the development, we will construct developments corresponding to these interfaces. One exception is the Clock interface that represents a simple time theory and cannot be usefully detalised in a module body

Implementing Modules

A machine providing the realisation of an interface is said to *implement* the interface. This is recorded by adding the interfaces into the **IMPLEMENTS** section of a machine. The fact that a machine provides a correct implementation of interfaces is established by a number of static checks and a set of proof obligations. The latter appear automatically in the list of machine proof obligations. The implementation relation is maintained during machine refinement (subject to some syntactic constraints) and thus the bulk of the module implementation activity is the normal Event B refinement process.

Event Group

The first step of implementing an interface is to provide at least one event for each interface operation. In general, an operation is realised by a set of events (an event **group**). Some events play a special role of operation termination events and are called **final** events. A final event returns the control to a caller. It must satisfy the operation post-conditions but there is no need to prove the convergence of a final event.

Implementing **ParkingGate**: abstract machine

To simplify proofs, the initial implementation is a simple machine with few events mirroring the interface operations. The machine retains interface variables *current* and *mcars* and also defines the operation return variables *Button_carid* and *OpenGate_moved*.

The names of the operation return variables are fixed for the first machine of a module implementation. In further refinements they may be replaced or removed using data refinement.

Implementing **ParkingGate**: abstract machine

The *button* event implements operation *Button* in a single atomic step. The fact that it is associated with operation *Button* is stated by GROUP Button. Being the only event in its operation group it is also a FINAL event.

MACHINE iParkingGate IMPLEMENTS
VARIABLES *current mcars Button_carid OpenGate_moved*
EVENTS
button = FINAL GROUP Button
WHEN
current = *empty*
THEN
current :∈ *CAR* \ {*empty*}
Button_carid := *current*
END

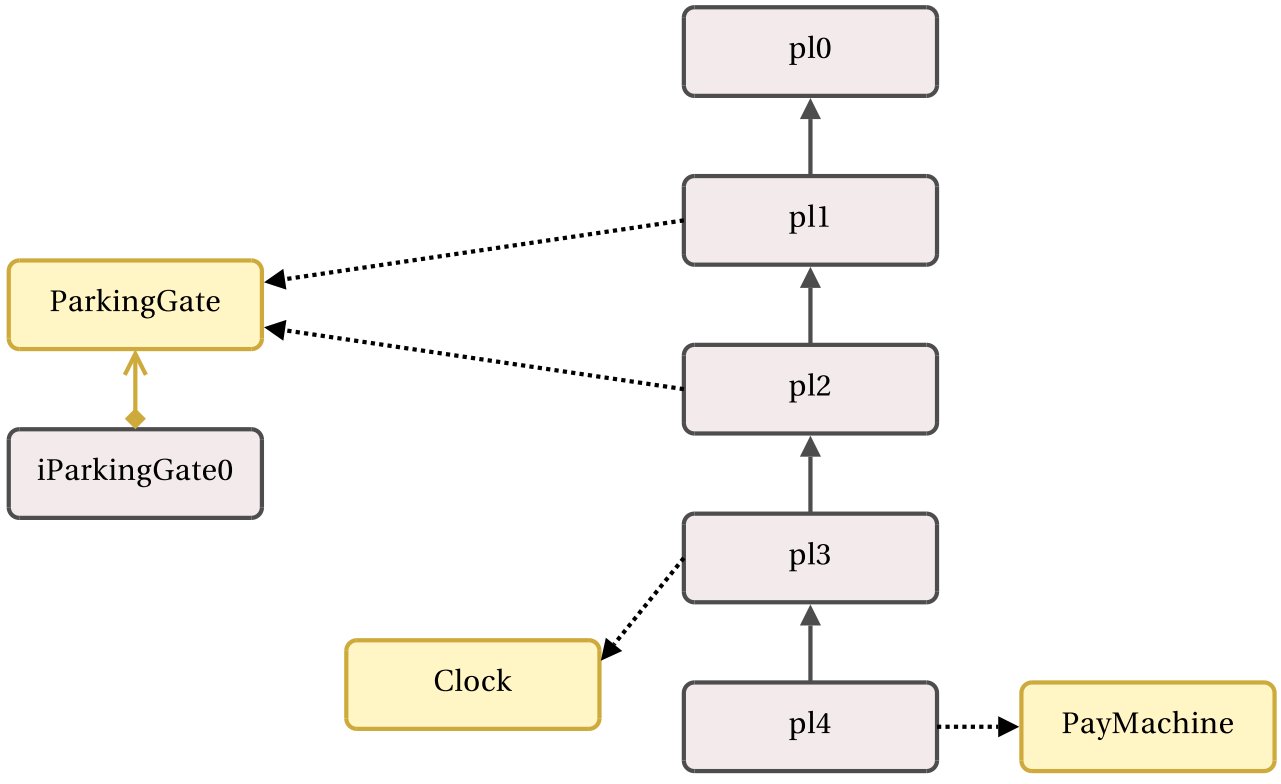
Implementing **ParkingGate**: abstract machine

The machine declares two more events, both realising the *OpenGate* operation. The events are final and each one handles one of the cases of the *OpenGate* operation post-condition.

gate_succ = FINAL GROUP OpenGate
WHEN
    $current \neq empty$
THEN
    $OpenGate\_moved := TRUE$
    $mcars := mcars + 1$
    $current := empty$
END

gate_nocar = FINAL GROUP OpenGate
WHEN
    $current \neq empty$
THEN
    $OpenGate\_moved := FALSE$
END

Development structure



one interactive proof (5 total)

Implementing **ParkingGate**: first refinement

new variables:

- *gate* - the gate state: open or closed
- *sensor* - the state of the car sensor placed; the sensor is placed on the parking lot of a gate
- *stage* - the current step of the gate operation

INVARIANT

$$gate \in GATE$$
$$sensor \in BOOL$$
$$stage \in 0 \ldots 3$$
$$stage = 1 \implies gate = OPEN$$
$$stage = 2 \implies gate = CLOSED$$

Implementing **ParkingGate**: first refinement

The refined implementation of the *OpenGate* operation includes events for opening and closing the gate.

open_gate = GROUP OpenGate
WHEN
gate = *CLOSED* ∧ *stage* = 0
THEN
gate := *OPEN*
stage := 1
END

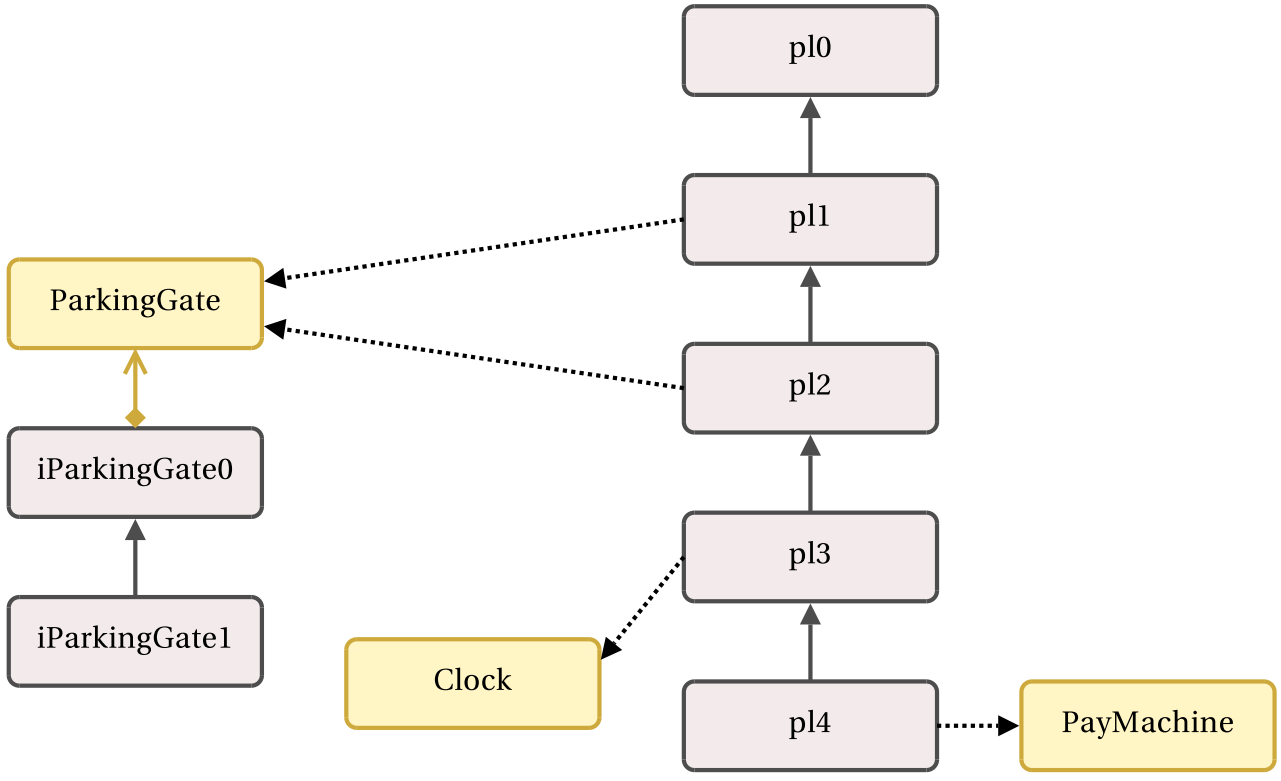
close_gate = GROUP OpenGate
WHEN
stage = 2
THEN
gate := *CLOSED*
stage := 3
END

Implementing **ParkingGate**: first refinement

the gate detects whether a car has passed through the gate while the gate was open:

readSensor $=$
GROUP OpenGate
WHEN
    $stage = 1$
THEN
    $sensor :\in BOOL$
    $stage :\in 1, 2$
END

Development structure



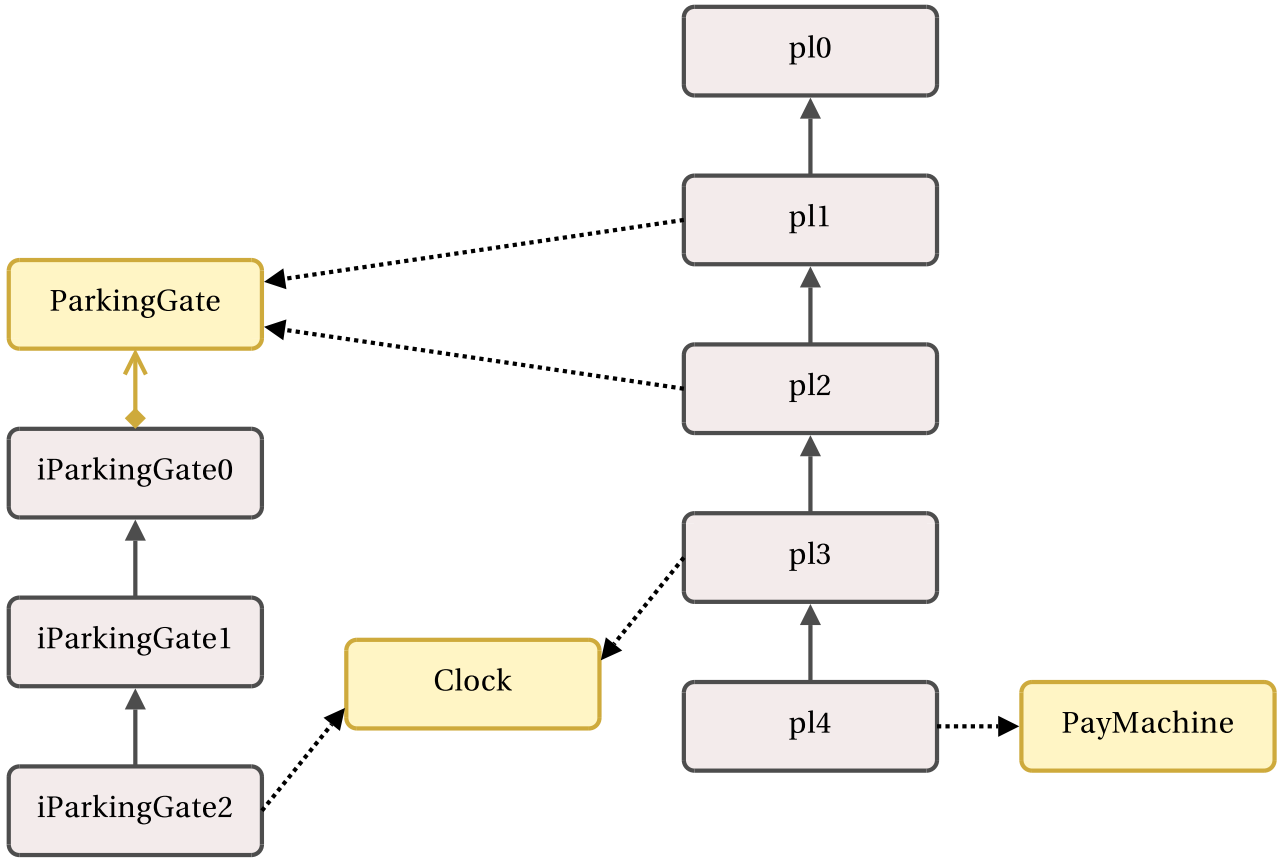
all proof obligations are discharged automatically (26 total)

Implementing **ParkingGate**: second refinement

To prove the convergence of anticipated event *readSensor*, the car sensor waits for a car for a given time interval. The time model is imported from the *Clock* interface.

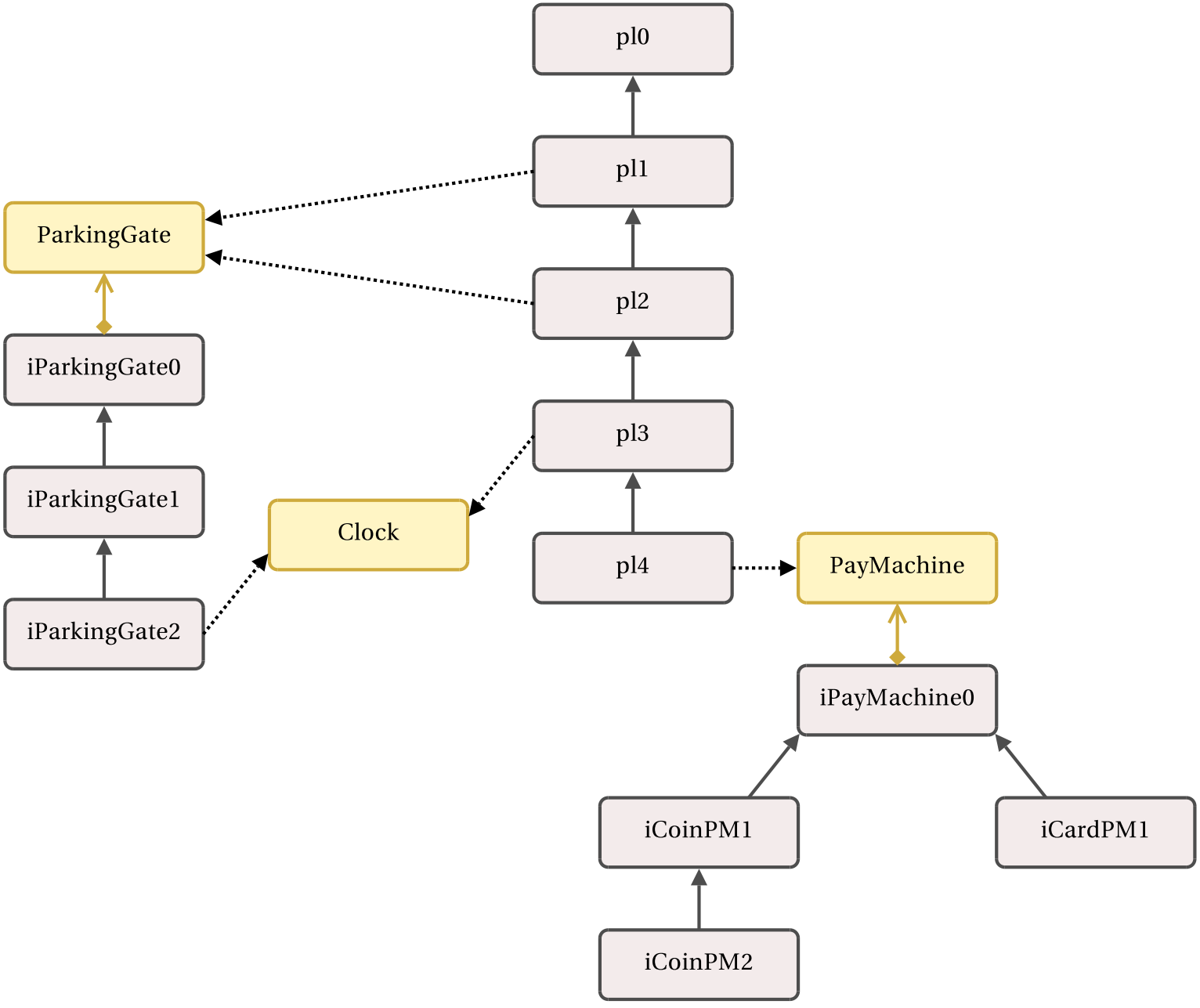
readSensor $=$
GROUP OpenGate
WHEN
$stage = 1$
$prev < delay$
THEN
$sensor, stage :| (sensor' = \mathrm{TRUE} \wedge stage' = 2) \vee (sensor' = \mathrm{FALSE} \wedge stage' = 1)$
$time :=$ **currentTime**
END

Development structure



two interactive proofs (8 total)

The overall development structure



Limitations

There are a number of limitations in the current version:

- the Rodin text editor is **not** supported: not only it is not aware of the new model elements it will also remove them when saving a model
- the Rodin builder is not aware of interface as a dependency of an importing machine: a change in an interface will not automatically trigger the rebuild of a machine importing the interface; the workaround is to manually "touch" the machine, e.g., by typing a space in a comment box (to be fixed in the next version)
- the project explorer sometimes does not show the list of POs of an interface; to see them (necessary for doing proofs) "touch" and save an interface component (to be fixed in the next version)

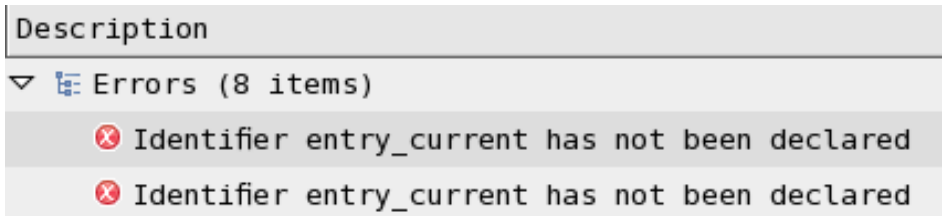
Limitations

Some verification conditions are not implemented yet:

- there is no proof obligation establishing that an operation may start whenever its post-condition is enabled. The plan is to rely on the Event B relative deadlock freeness condition when it becomes available
- module body data refinement constraint is not checked. The constraint would demonstrate that there is a functional mapping between the variables of a module body machine and the corresponding interface variables (to be added in the next version)
- there is no check that module parameters (interface constants) are not constrained in a module implementation. Writing such constraints (additional axioms) could result in developments that are not recomposable due the inability to apply the module instantiation (to be added in the next version)

Some common problems

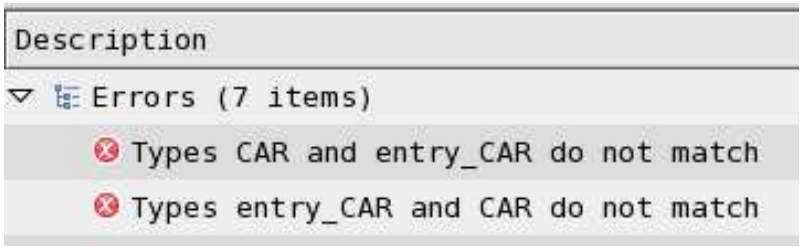
The platform *Refine* command does not copy the variables imported from modules. This results in a large number of error messages for a refinement machine:



The fix is to simply list all the variables in the **VARIABLES** section of the refinement machine.

Some common problems

Types imported from prefixed modules are automatically prefixed and are distinct from their unprefixed versions. This may be a reason for unexpected typing errors:



Some common problems

If you intend to use unprefixed types check that the module instantiation rewrites the types as necessary: