Modularisation in Event-B

Alexei Iliasov, Alexander Romanovsky
Elena Troubitsyna, Linas Laibinis

Newcastle University
Åbo Academy

Contents

Installation

The plugin works with the platform version 2.0 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:

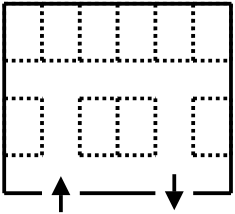
1. no car may enter when there is no space left in the parking lot
2. a fare must be paid when a car leaves the parking lot
3. each time a car leaves the parking lot, the fare to be paid is determined by multiplying the total length of stay since the midnight (that is, including any previous stay(s)) 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

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:

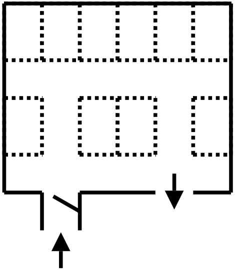
$$\text{enter} \quad = \quad \text{WHEN} \quad \mathit{entered} - \mathit{left} < \mathit{LOT\_SIZE} \quad \text{THEN} \quad \mathit{entered} := \mathit{entered} + 1 \quad \text{END}$$

a car leaves:

$$\text{leave} \quad = \quad \text{WHEN} \quad \mathit{entered} - \mathit{left} > 0 \quad \text{THEN} \quad \mathit{left} := \mathit{left} + 1 \quad \text{END}$$

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:

$$\text{carid} \leftarrow \text{Button} \quad = \quad \begin{array}{l} \text{PRE} \\ \quad \textit{current} = \textit{empty} \\ \text{POST} \\ \quad \textit{current}' \in \textit{CAR} \setminus \{\textit{empty}\} \\ \quad \textit{carid}' = \textit{current} \\ \text{END} \end{array}$$

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:

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

Gate operations

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

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

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:

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

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

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

Import invariants

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

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

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

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

Model events

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

$$
\begin{array}{lcl}
\text{UserPressButton} & = & \textsc{when} \\
 & & \quad \mathit{entered} - \mathit{left} < \mathit{LOT\_SIZE} \\
 & & \quad \mathit{entry\_current} = \mathit{empty} \\
 & & \quad \mathit{inmoved} = \mathit{FALSE} \\
 & & \textsc{then} \\
 & & \quad \mathit{incar} := \textbf{entry\_Button} \\
 & & \textsc{end}
\end{array}
$$

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):

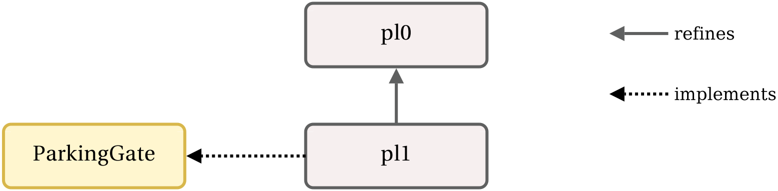
$$\text{CtrlOpenGate} \;=\; \begin{array}{l} \textsc{when} \\ \quad \textit{entry\_current} \neq \textit{empty} \wedge \textit{inmoved} = \mathrm{FALSE} \\ \textsc{then} \\ \quad \textit{inmoved} := \textbf{entry\_OpenGate} \\ \textsc{end} \end{array}$$

Model events

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

$$
\begin{array}{lcl}
\mathsf{enter} & = & \text{WHEN} \\
 & & \quad \mathit{inmoved} = \text{TRUE} \\
 & & \text{THEN} \\
 & & \quad \mathit{entered} := \mathit{entered} + 1 \\
 & & \quad \mathit{inmoved} := \text{FALSE} \\
 & & \text{END}
\end{array}
$$

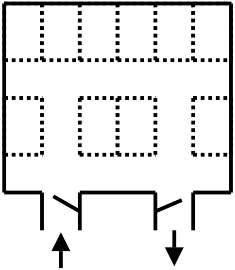
Development structure



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
$outcar \in CAR$
$outmoved \in BOOL$

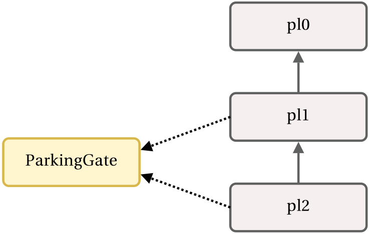
Import invariant

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:

$$\begin{aligned} &\mathit{inmoved} = \mathrm{FALSE} \land \mathit{outmoved} = \mathrm{FALSE} \implies \\ &\qquad \mathit{entry\_mcars} - \mathit{exit\_mcars} \in 0 \ldots \mathit{LOT\_SIZE} \end{aligned}$$

Development structure

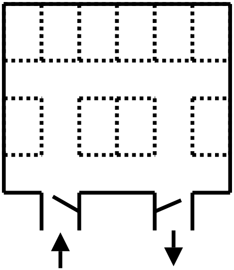


one interactive proof (17 total)

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

$$prev \in from \ldots to$$

Clock operations

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

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

the clock is reset:

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

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
$register : CAR \nrightarrow TOD$
$cartime : CAR \rightarrow \mathbb{N}$

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:

$$\mathsf{enter} \;=\; \textsc{extends}\ \mathsf{enter}\ \ \textsc{begin}\ \ \mathit{register}(\mathit{incar}) := \mathbf{currentTime}\ \ \textsc{end}$$

currentTime is an operation call returning (and also advancing) the current time when a car leaves, the registration record is removed and the total stay time is updated:

$$
\begin{array}{lcl}
\mathsf{CtrlOpenGateL} & = & \textsc{extends}\ \mathsf{CtrlOpenGateL} \\
& & \textsc{when} \\
& & \quad \mathit{outcar} \in \mathit{dom}(\mathit{register}) \\
& & \textsc{then} \\
& & \quad \mathit{cartime}(\mathit{outcar}) := \mathit{cartime}(\mathit{outcar}) + \\
& & \qquad\qquad (\mathbf{currentTime} - \mathit{register}(\mathit{outcar})) \\
& & \quad \mathit{register} := \mathit{outcar} \lhd \mathit{register} \\
& & \textsc{end}
\end{array}
$$

here $\mathbf{currentTime} - \mathit{register}(\mathit{outcar})$ 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:

$$
\begin{array}{lll}
\text{RegisterReset} & = & \textsc{when} \\
 & & \quad \mathit{prev} = \mathit{DAY\_END} \\
 & & \textsc{then} \\
 & & \quad \textbf{clockReset} \\
 & & \quad \mathit{register} := \mathit{dom}(\mathit{register}) \times \{\mathit{DAY\_START}\} \\
 & & \quad \mathit{cartime} := \mathit{CAR} \times \{0\} \\
 & & \textsc{end}
\end{array}
$$

operation **clockReset** sets the clock reading *prev* to the start of a day time value $\mathit{DAY\_START}$

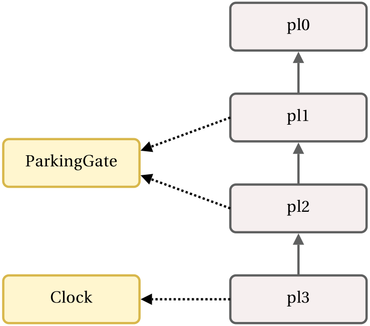
Model events

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

$$\text{ObserveTime} \quad = \quad \text{BEGIN} \quad \textbf{currentTime} \quad \text{END}$$

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

Development structure

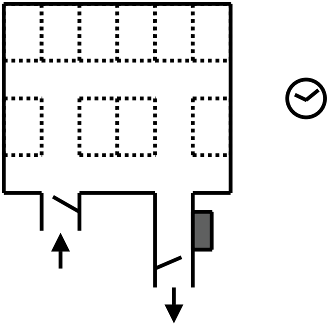


five interactive proofs (30 total)

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:

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

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):

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

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:

$$\boxed{\textit{exit\_current} = \textit{empty} \lor \textit{confPay} = \mathrm{TRUE} \implies \textit{pm\_balance} = 0}$$

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

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

Model events

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

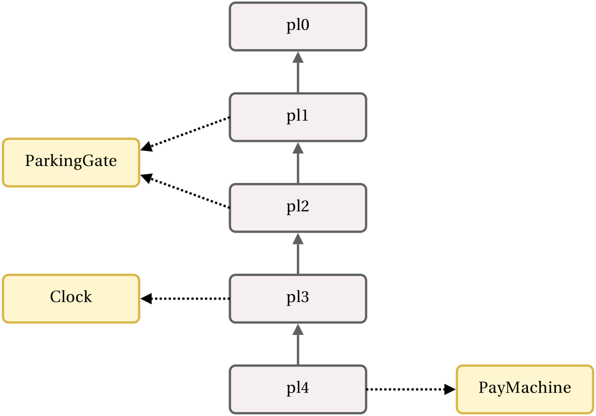
$$\begin{array}{lcl} \mathsf{CtrlPay} & = & \textsc{when} \\ & & \quad \mathit{confPay} = \mathrm{TRUE} \\ & & \textsc{then} \\ & & \quad \mathit{pm\_void} := \mathbf{pm\_Configure}(\mathit{cartime}(\mathit{outcar})) \\ & & \quad \mathit{confPay} := \mathrm{FALSE} \\ & & \textsc{end} \end{array}$$

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):

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

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

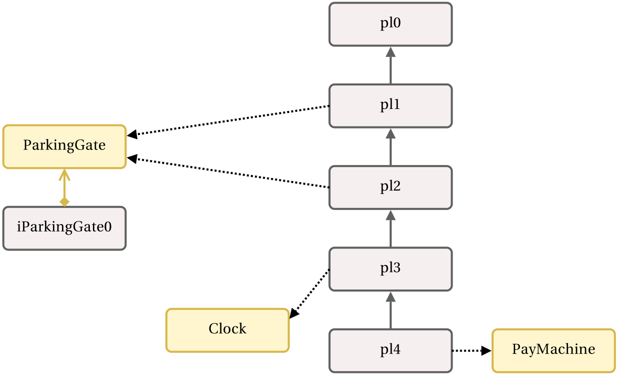
button = FINAL GROUP Button
WHEN
$current = empty$
THEN
$current :\in CAR \setminus \{empty\}$
$Button\_carid := current$
END

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.

$$
\begin{array}{lll}
\mathsf{gate\_succ} & = & \textsc{final group}\ \mathsf{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{current} \neq \mathit{empty} \\
 & & \textsc{then} \\
 & & \quad \mathit{OpenGate\_moved} := \mathit{TRUE} \\
 & & \quad \mathit{mcars} := \mathit{mcars} + 1 \\
 & & \quad \mathit{current} := \mathit{empty} \\
 & & \textsc{end} \\
\mathsf{gate\_nocar} & = & \textsc{final group}\ \mathsf{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{current} \neq \mathit{empty} \\
 & & \textsc{then} \\
 & & \quad \mathit{OpenGate\_moved} := \mathit{FALSE} \\
 & & \textsc{end}
\end{array}
$$

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.

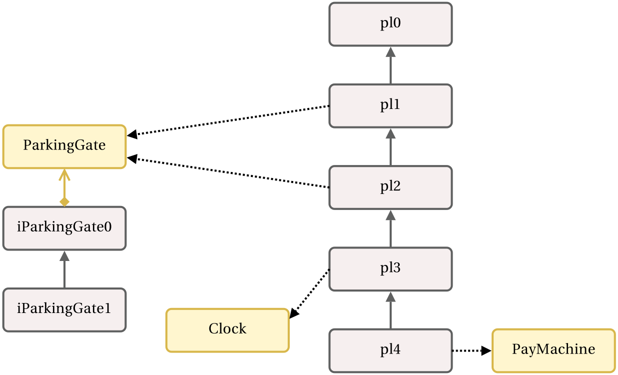
$$
\begin{array}{lcl}
\mathsf{open\_gate} & = & \textsc{group}\ \mathsf{OpenGate} \\
 & & \textsc{when} \\
 & & \quad gate = CLOSED \wedge stage = 0 \\
 & & \textsc{then} \\
 & & \quad gate := OPEN \\
 & & \quad stage := 1 \\
 & & \textsc{end} \\
\mathsf{close\_gate} & = & \textsc{group}\ \mathsf{OpenGate} \\
 & & \textsc{when} \\
 & & \quad stage = 2 \\
 & & \textsc{then} \\
 & & \quad gate := CLOSED \\
 & & \quad stage := 3 \\
 & & \textsc{end}
\end{array}
$$

Implementing **ParkingGate**: first refinement

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

$$
\begin{array}{lcl}
\text{readSensor} & = & \textsc{group}\ \text{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{stage} = 1 \\
 & & \textsc{then} \\
 & & \quad \mathit{sensor} :\in \mathit{BOOL} \\
 & & \quad \mathit{stage} :\in 1, 2 \\
 & & \textsc{end}
\end{array}
$$

Development structure



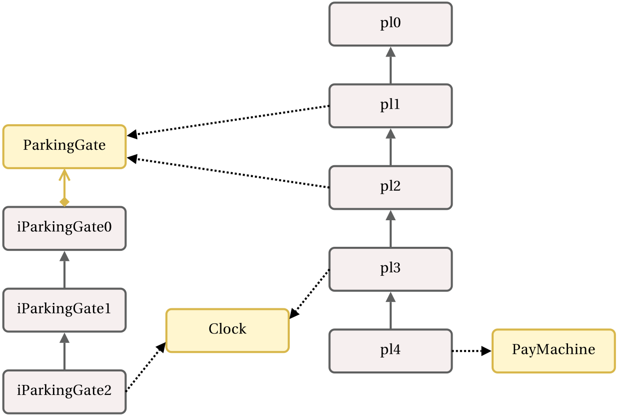
all proof obligations are discharged automatically (26 total)

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.

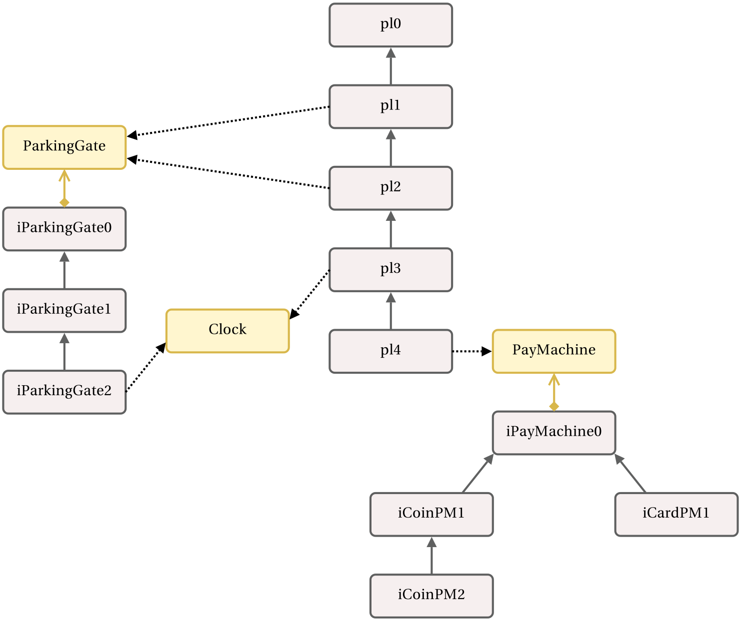
$$
\begin{array}{lcl}
\text{readSensor} & = & \textsc{group}\ \text{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{stage} = 1 \\
 & & \quad \mathit{prev} < \mathit{delay} \\
 & & \textsc{then} \\
 & & \quad \mathit{sensor}, \mathit{stage} :\mid (\mathit{sensor}' = \mathrm{TRUE} \land \mathit{stage}' = 2) \lor \\
 & & \qquad\qquad (\mathit{sensor}' = \mathrm{FALSE} \land \mathit{stage}' = 1) \\
 & & \quad \mathit{time} := \textbf{currentTime} \\
 & & \textsc{end}
\end{array}
$$

Development structure



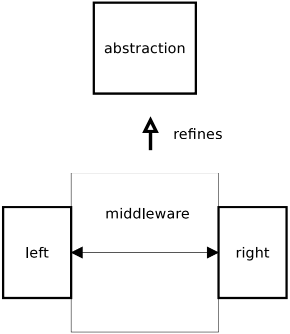
two interactive proofs (8 total)

The overall development structure



Decomposition Patterns

The purpose of the patterns is to facilitate a specific form of model decomposition where the core functionality of an abstraction is distributed among two or more separate modules connected by a relatively primitive middleware model. Typically, the modules correspond to the software or hardware being developed while the middleware model should match an existing coordination infrastucture.



Component template

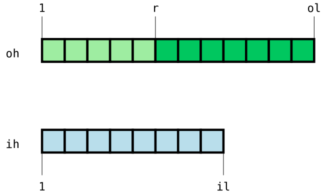
A generic component description is given by an interface. The interface variables are seen by the middleware component and can be used to formulate a gluing invariant.

The integration is achieved by the means of two operations: one for informing the component about a new incoming messsage and another confirming the processing of an outgoing message.

The production of new messages is modelled as an independent thread of control witin a component. At the interface level the thread manifests itself by adding new messaging into output message queue.

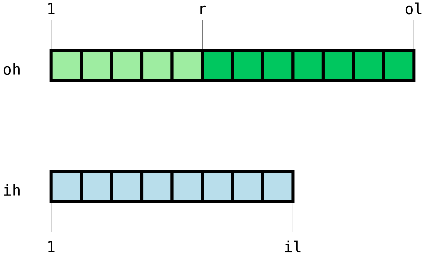
Component template variables

Interface defines message sequences recording all the incoming and outgoing messages during the component lifetime. The output message history also plays the role of output message queue.



In the above and further, light green are messages already processed by the middleware and are either delivered or in transmission. Dark green slots correspond to fresh messages not yet processed by the middleware. Blue slots are messages originated at another component and delivered by the middleware.

Component template variables



$ih \in 1 \ldots il \rightarrow MSG$	input history sequence
$oh \in 1 \ldots ol \rightarrow MSG$	output history sequence
$ol \geq r \wedge ol r \leq QUEUE\_LENGTH$	output queue capacity

Component template operations

The *receive* operation reacts on an incoming. At the interface level the observed effect is a new message in the input history.

$$
\begin{array}{lll}
\mathsf{receive} & = & \textsc{any}\ m\ \textsc{pre} \\
 & & \quad m \in MSG \\
 & & \textsc{post} \\
 & & \quad ih' = ih \cup \{il + 1 \mapsto m\} \\
 & & \quad il' = il + 1 \\
 & & \textsc{end}
\end{array}
$$

An actual model would define further pre- and post-conditions.

Operation *deliver* marks a message in the output queue as processed. It also frees one slot in the output message queue.

$$
\mathsf{m} \leftarrow \mathsf{deliver} \quad = \quad \begin{array}{l} \text{PRE} \\ \quad ol > r \\ \text{POST} \\ \quad m' = oh(r+1) \\ \quad r' = r + 1 \\ \text{END} \end{array}
$$

When the operation is called it returns a message to be delivered.

Component template process

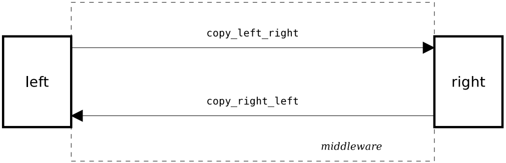
A component has a thread of control that allows it accomplish some tasks independently of middleware and other components. One observed effect of the thread execution is the generation of new messages in the output queue.

$$\mathsf{proc} \;=\; \text{GUARANTEE} \quad ol' \geq ol \wedge ol'r \leq QUEUE\_LENGTH \quad oh = 1 \ldots ol \lhd oh' \quad \text{END}$$

The process guarantee states that the process may change the output queue by adding new messages to the queue tail untill the capacity limit is reached.

Synchronous Coordination

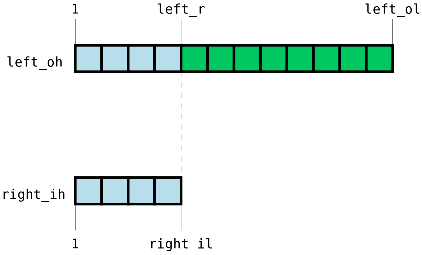
The templates defines a middleware transmitting a message between two components in one atomic step. The middleware has no memory (hence no model variables of its own) and acts as a wiring logic relating two modules.



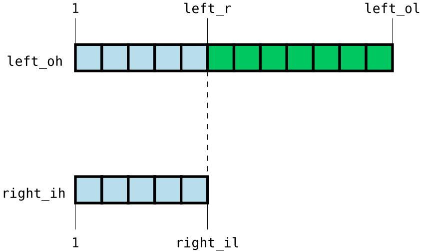
It is convinient to represent bi-directional channels as a pair of uni-directional ones.

When two components are linked by a synchronous template the following properties are maintained:

$$left\_ih = 1 \ldots right\_r \lhd right\_oh$$
$$left\_il = right\_r$$

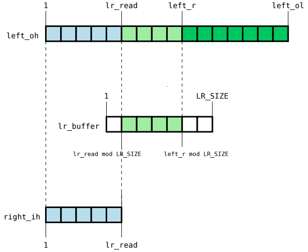


Event *copy_left_right* transfers one message from component *left* to component *right* in an atomic step.



Asynchronous Coordination

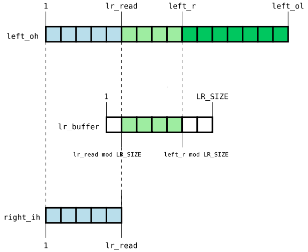
With the asynchronous communication a middleware is equipped with some memory used to temporarily save messages travelling across two components. This permits one to relax an assumption that a message is received the moment it is sent.



For each channel there are three message classes: messages sent and received, messages in transmission and messages yet to be transmitted.

Asynchronous Coordination

A buffer stores the messages currently in transmission. Its content is a projection of the messages generated by a sender.



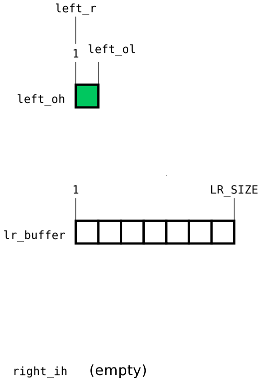
$$\forall i \cdot i \in lr\_read \ldots left\_r - 1 \implies$$
$$lr\_buffer(i \ mod \ LR\_BUFFER\_SIZE + 1) = left\_oh(i + 1)$$

Buffer is a FIFO realised as a circular buffer (should have used an infinite sequence to simplify the proofs!).

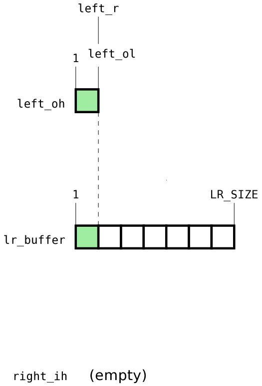
lr_buffer is a buffer is attached to the *left* to *right* channel. Its write position is $lr\_write \ mod \ LR\_BUFFER\_SIZE + 1$ and read position is $lr\_read \ mod \ LR\_BUFFER\_SIZE + 1$. To maintain a lossless buffer we require that $lr\_read \leq lr\_write$.

The number of messages received on a channel equals the number of messages read from the channel buffer: $right\_il = lr\_read$ and $left\_il = rl\_read$

Initially, a message appears in the output queue (and also the output history of a component). Such message is generated by an internal process of a component.

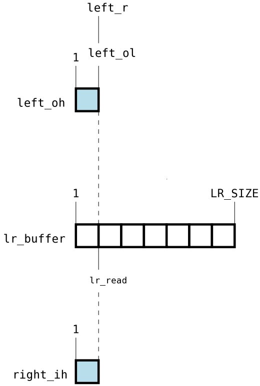


The middleware copies the message in a channel buffer

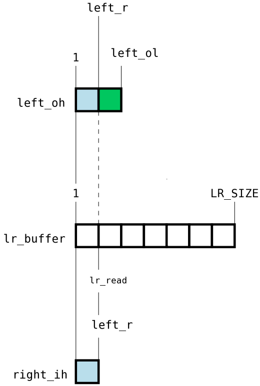


Asynchronous Coordination

The message is delivered to the receiver

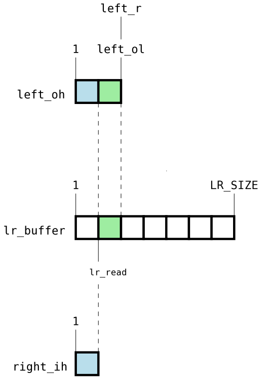


The sender generates another message



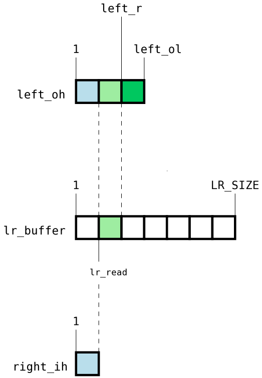
Asynchronous Coordination

The message is coped by the middleware into the buffer



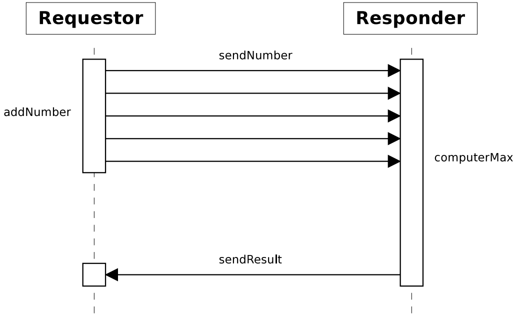
Asynchronous Coordination

The sender generates another message



Example: MaxOf5

As an example we consider the specification of a simple protocol following the typical request/response pattern.



The requester sends five distinct numbers to the responder which, as reply, sends a message with the maximum of the numbers.

Example: MaxOf5

The complexity in the example is not in the nature or the properties of the protocol but rather in proving the refinement step leading from an abstract protocol specification to a decomposed (distributed/concurrent) model.

The development in this example is built as around the synchronous communication template.

Abstract Model

- *set* - set of generated numbers (requests)
- *result* - the computed result variable

$$
\begin{array}{l}
\mathit{set} \subseteq \mathbb{N} \\
\mathit{finite}(\mathit{set}) \\
\mathit{card}(\mathit{set}) \leq 5 \\
\mathit{result} \in \mathbb{Z}
\end{array}
$$

Abstract Model

generates a new number until there are total 5 numbers in the set

$$
\begin{array}{lcl}
\text{addNumber} & = & \text{ANY } n \text{ WHERE} \\
 & & \quad n \in \mathbb{N} \\
 & & \quad n \notin set \\
 & & \quad card(set) < 5 \\
 & & \text{THEN} \\
 & & \quad set := set \cup \{n\} \\
 & & \text{END}
\end{array}
$$

Abstract Model

computes the maximum of the generated number set and saves the result

$$
\text{computeMax} \;=\; \begin{array}{l} \textsc{when} \\ \quad \mathit{card}(\mathit{set}) = 5 \\ \quad \mathit{result} = 1 \\ \textsc{then} \\ \quad \mathit{result} := \mathit{max}(\mathit{set}) \\ \textsc{end} \end{array}
$$

Concrete Model: Import Invariants

own variables: **none**

set is replaced by the contents of the output history of the generator component:

$$\boxed{set = ran(g\_oh)}$$

result is replaced by the first entry of the processor component output history:

$$\boxed{\begin{array}{l} p\_ol > 0 \implies result = p\_oh(1) \\ p\_ol = 0 \implies result = 1 \end{array}}$$

Concrete Model: Import Invariants

Connecting the communication histories of the components

$$p\_ih = 1 \ldots g\_r \lhd g\_oh$$
$$p\_il = g\_r$$
$$g\_ih = 1 \ldots p\_r \lhd p\_oh$$
$$g\_il = p\_r$$

Protocol property: when the responder has received five messages the requestor has nothing more left to send

$$card(ran(p\_ih)) = 5 \implies g\_r = g\_ol$$

Concrete Model (Middleware)

an unbuffered channel connecting requestor and responder (one way only) this event takes a message from the output history and places it in the input history of another component

$$\text{copy\_gp} \;=\; \textsc{when} \quad g\_ol > g\_r \quad \textsc{then} \quad \textbf{p\_receive}(\textbf{g\_deliver}) \quad \textsc{end}$$

Concrete Model (Middleware)

an unbuffered channel connecting responder and requestor (one way only)

$$\text{copy\_pg} = \text{WHEN} \quad p\_ol > p\_r \quad \text{THEN} \quad \mathbf{g\_receive(p\_deliver)} \quad \text{END}$$

Module process definitions

an unbuffered channel connecting responder and requestor (one way only)

$$\text{generate} \;=\; \begin{array}{l} \text{WHEN} \\ \quad n \in \mathbb{N} \setminus ran(oh) \\ \quad card(oh) < 5 \\ \text{THEN} \\ \quad oh(ol+1) := n \\ \quad ol := ol + 1 \\ \text{END} \end{array}$$

$$\text{compute} \;=\; \begin{array}{l} \text{WHEN} \\ \quad card(ran(ih)) = 5 \\ \quad ol = 0 \\ \text{THEN} \\ \quad oh(ol+1) := max(ran(ih)) \\ \quad ol := 1 \\ \text{END} \end{array}$$

Summary and Future Work

- working on several other patterns: circular buffer, lossy buffer
- maxof5 examples for all the patterns
- longer-term: distributed AOCS

Experience

- AOCS mode consistency: NCL/Abo (1400 POs)
- AOCS messaging model: (900 POs)
- SAP protocol model: (350 POs)

Tool evolution (until 1 Oct):

- 4 releases
- 22 bug reports (all fixed)
- 12 feature requests (10 implemented)
- $>$200 installations (45 for the latest version)

Experience

Initial modelling attempts by industrial partners were unsuccessful

- not using an implementaton machine
- introducing modules too early
- too complex pre- and post-conditions
- misunderstanding of the purpose of operations

Half day training sessions were sufficient for further independent work

Some common mistakes

Modularisation should not be used to compose models

- composition is a (degenerate) form of decomposition refinement where no abstraction is given for the new functionality introduced by a module
- results in a fragmented model
- not a part of the refinement methodology
- it is a poor specification technique: inflates a model without generating any interesting (related to the modelled problem) proof obligations

Some common mistakes

It is mandatory to supply an import invariant

- import invariant is a form of a gluing invariant connecting the exported variables of a module with the variables of the importing machine
- only abstract and preserved variables of the parent machine may be used in import invariant
- the good technique is to try to replace some variables of parent machine with the exported variables
- this allows to refine actions of the abstract parent machine with operation calls

Semi-automatic decomposition

A tool that mechanises a specific decomposition approach using in the modelling of communication protocols.

- define number of modules and their names
- define variable distribution
- define operations