Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling Traffic Light Control

From Requirements to an ASM Ground Model

Università di Pisa, Dipartimento di Informatica boerger@di.unipi.it
Universität Ulm, Abteilung Informatik alexander.raschke@uni-ulm.de

See Ch. 2.1 of Modeling Companion

One-Way Traffic Light Requirements (M.Jackson, p.49)

PlantReq ... to enforce one-way traffic...the traffic is controlled by a pair of simple portable traffic light units...one unit at each end of the one-way section...connect(ed)...to a small computer that controls the sequence of lights.

UnitReq. Each unit has a Stop light and a Go light.

PulseReq. The computer controls the lights by emitting rPulses and gPulses, to which the units respond by turning the light on and off.

RegimeReq. The regime for the lights repeats a fixed cycle of four phases. First, for 50 seconds, both units show Stop; then, for 120 seconds, one unit shows Stop and the other Go; then for 50 seconds both show Stop again; then for 120 seconds the unit that previously showed Go shows Stop, and the other shows Go. Then the cycle is repeated.

Model elements (signature)

By *UnitReq*uirement: two units each with

- a $StopLight(i)$ and a $GoLight(i)$ for $i = 1, 2$

By *PulseReq*irement there are two attribute values on or off:

- $StopLight(i), GoLight(i) \in \{on, off\}$

RegimeReq: the two lights of a unit seem to take opposite values (but see below for a variation)

$Stop(i)$ iff

$StopLight(i) = on$ **and** $GoLight(i) = off$ // show Stop i

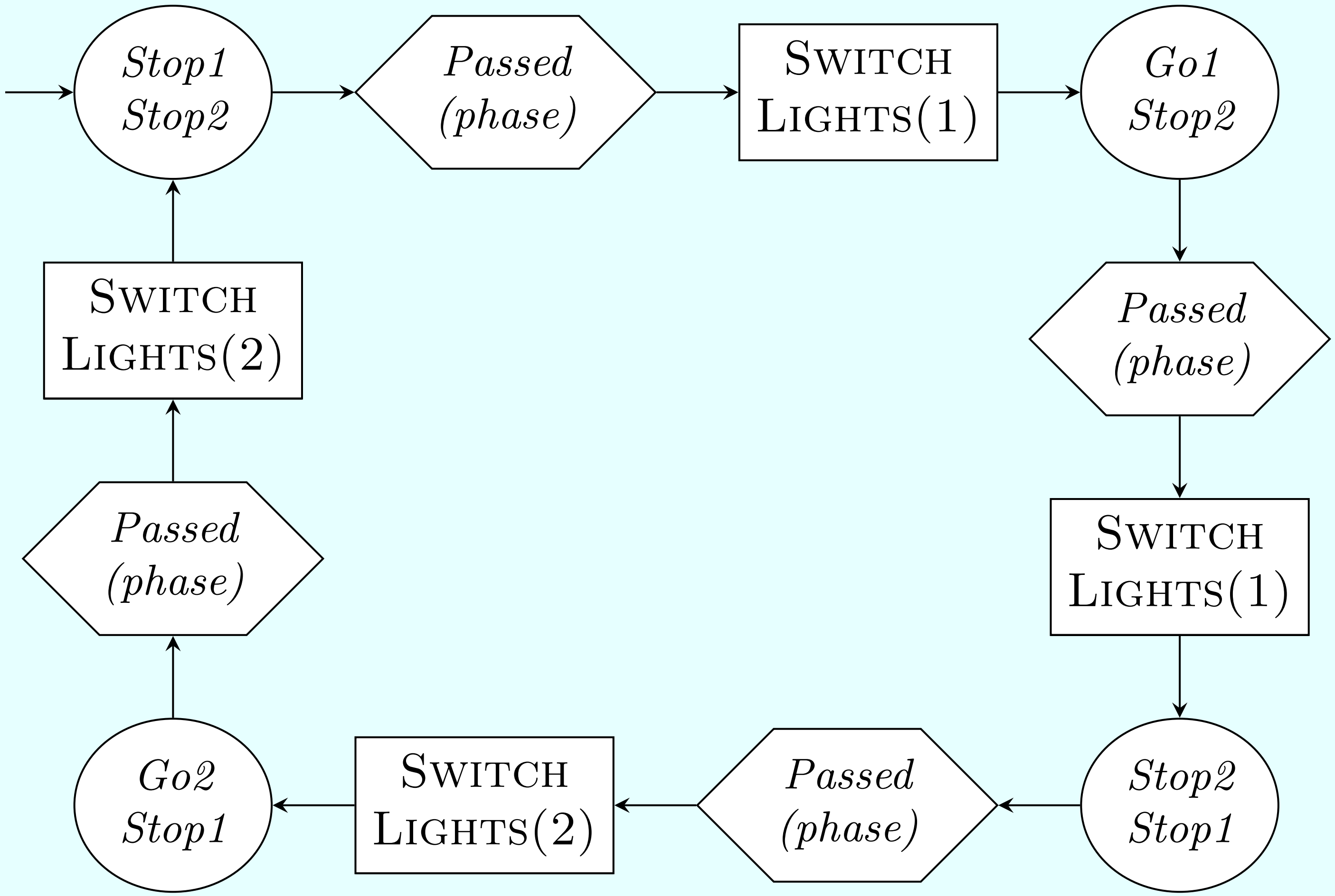
$Go(i)$ iff

$StopLight(i) = off$ **and** $GoLight(i) = on$ // show Go i

Combinations of these *'derived' predicates* $Stop(i), Go(i)$ define the intended interpretation of the phase values (see below).

One-Way Traffic Light Ground Model 1WayTrafLightSpec

RegimeReq cyclic 4-phases behavior model:[1]



[1] Figure © 2010 Springer Berlin-Heidelberg, reused with permission.

Flowchart representation

- phase values represented by *control states* (just names):

  $Stop1 \quad Go1 \quad Stop2 \quad Go2$
  $Stop2 \quad Stop2 \quad Stop1 \quad Stop1$

  pictorially represented by circles labeled with the control state name
- *state changes* pictorially represented by *arrows* leading
  - from a control state $phase = source$
  - to a rhomb labeled by a transition guard $condition$, from there
  - to a rectangle labeled by an $action$ and from there
  - to a control state $target$

Behavioral meaning:

- if the system is in the control state $source$ and the $condition$ is true in the current system state, then the $action$ is performed changing the system state and the system enters the next control state $target$

Functions/Locations

- $Passed(phase) = Elapsed(period(phase))$: a derived function
- $period$: a static (possibly configurable) function
  - here with values as stated in *RegimeReq*: 50 or 120 seconds depending on the $phase$ argument
- $Elapsed$: a monitored (say Boolean-valued) timeout function
  - assumed to provide an external clock signal each time $period(ctlstate)$ has $Elapsed$ since $phase$ has been updated to $ctlstate$

TimerAssumption. If in a run $phase$ is updated by a rule to a $ctlstate$, then after $period(ctlstate)$ the timeout signal $Elapsed(period(ctlstate))$ is set by an external timer (to true). It is reset (to false) when the rule it triggers is executed.

Functional model of RegimeReq action

Abstracting from the computer and its connection to the light units, at the functional level of abstraction:

- $\textsc{SwitchLights}(i)$ for $i = 1, 2$ means *updating* the unit i light $StopLight(i), GoLight(i)$ values—either on or off—from their current value to their opposite value required for the next phase

$\textsc{SwitchLights}(i) =$

$\quad \textsc{Switch}(StopLight(i))$

$\quad \textsc{Switch}(GoLight(i))$

where

$\quad \textsc{Switch}(l) = (l := l')$ // $'$ denotes the opposite value

Textual form of 1WayTrafLightSpec

$\textsc{1WayStopGoLightSpec} =$

if $phase \in \{Stop1Stop2, Go1Stop2\}$ **and** $Passed(phase)$ **then**

$\textsc{SwitchLights}(1)$ -- from $Stop(1)$ to $Go(1)$ or viceversa

if $phase = Stop1Stop2$ **then** $phase := Go1Stop2$

else $phase := Stop2Stop1$

if $phase \in \{Stop2Stop1, Go2Stop1\}$ **and** $Passed(phase)$ **then**

$\textsc{SwitchLights}(2)$ -- from $Stop(2)$ to $Go(2)$ or viceversa

if $phase = Stop2Stop1$ **then** $phase := Go2Stop1$

else $phase := Stop1Stop2$

where $\textsc{SwitchLights}(i) =$

$\textsc{Switch}(StopLight(i))$ $\textsc{Switch}(GoLight(i))$

$\textsc{Switch}(l) = (l := l')$ -- $'$ denotes the opposite value

$Passed(phase) = Elapsed(period(phase))$

Recap of $1\textsc{WayTrafLightSpec}$ elements

- 5 controlled fcts (0-ary):
  - to express *RegimeReq*

$$phase \in \{\overset{Stop1}{Stop2}, \overset{Go1}{Stop2}, \overset{Stop2}{Stop1}, \overset{Go2}{Stop1}\}$$

  - by *UnitReq* $StopLight(i), GoLight(i) \in \{on, off\}$ (with $i = 1, 2$)
- 1 derived fct (predicate): $Passed(phase) = Elapsed(period(phase))$
  - defined by *RegimeReq* in terms of
    - a static fct $period$ and
    - a monitored fct (predicate) $Elapsed$
- 4 rules updating controlled fcts triggered by dynamic guards
  - defined by flowchart above
  - using submachines $\textsc{SwitchLights}(i)$ with $i = 1, 2$

NB. Here there is no shared and no output function

Initialization and Correctness of $1\textsc{WayTrafLightSpec}$

*InitReq*uirement (added). The light regime initiates with both units showing Stop.

Correspondingly initial states in the ASM model are defined by:

$$\left(phase = \begin{array}{c} Stop1 \\ Stop2 \end{array} \right) \quad \textbf{and } Stop(1) \textbf{ and } Stop(2)$$

By model inspection one can verify:

Correctness Property: Each legal run of $1\textsc{WayTrafLightSpec}$ satisfies the *RegimeReq*.

A run of $1\textsc{WayTrafLightSpec}$ is legal if it is started in the initial state and satisfies the *TimerAssumption*.

Refinement for Management of Design Decisions

Needed: generalization of classical refinement method (Wirth/Dijkstra)

- to cope with the "explosion of 'derived requirements' (the requirements for a particular design solution) caused by the complexity of the solution process" and encountered "when moving from requirements to design" (Glass 2003, Fact 26)
- to check and document by correctness proofs the design decisions taken in linking through various levels of abstraction the system architect's view (at the abstraction level of a blueprint) to the programmer's view (at the level of detail of compilable code)
  - *split checking complex detailed properties* into a series of simpler checks of more abstract properties and their correct refinement
  - provide systematic *rigorous system development documentation*, including behavioral information and needed internal interfaces by state-based abstractions

Refinement guided by domain knowledge

Here: transform 1-agent ASM 1WayTrafLightSpec into a 2-agent ASM *separating*

- *computer actions* (pulse emission) performed by a control software ASM 1WayTrafLightCtl
- resulting light *equipment actions* performed by an environment ASM LightUnitResponse

This means to split

- one (at the abstract level 'atomic') SwitchLights step of the system model 1WayTrafLightSpec into
- two (at the more detailed level of abstraction again 'atomic') steps:

i.e. a computer action Emit($pulse$) and a corresponding environment action which Switches the lights.

Sw/Env ASMs $1\textsc{WayTrafLightCtl}/\ \textsc{LightUnitResponse}$

For $1\textsc{WayTrafLightCtl}$ a submachine refinement suffices:

$\textsc{SwitchLights}(i) =$

$\textsc{Emit}(rPulse(i))$ -- trigger $\textsc{Switch}(StopLight(i))$

$\textsc{Emit}(gPulse(i))$ -- trigger $\textsc{Switch}(GoLight(i))$

where $\textsc{Emit}(p) = (p := high)$

For the env *PulseReq* steers the env re-action to pulses:

$\textsc{LightUnitResponse} =$

forall $i \in \{1, 2\}$ $\textsc{LightUnitResponse}(i)$

where $\textsc{LightUnitResponse}(i) =$ $\textsc{ReactTo}(rPulse(i))$
$\textsc{ReactTo}(gPulse(i))$

Environment LightUnitResponse components

$\textsc{ReactTo}(rPulse(i)) =$ **if** $Event(rPulse(i))$ **then**

$\textsc{Switch}(StopLight(i))$

$\textsc{Consume}(rPulse(i))$

$\textsc{ReactTo}(gPulse(i)) =$ **if** $Event(gPulse(i))$ **then**

$\textsc{Switch}(GoLight(i))$

$\textsc{Consume}(gPulse(i))$

where

$Event(p)$ iff $p = high$

$\textsc{Consume}(p) = (p := low)$

NB. $rPulse(i)$ and $gPulse(i)$ shared by sw $1\textsc{WayTrafLightCtl}$ and env $\textsc{LightUnitResponse}$. Inconsistent updates are excluded by the LightUnitResponseAssumption below.

Refining initialization and assumptions

- *InitPulseReq*. Initially no pulses have been emitted.

Corresponding stipulation for $1\textsc{WayTrafLightCtl}$ and $\textsc{Pulses}$:

forall $i \in \{1,2\}$ $rPulse(i) = gPulse(i) = low$

- *LightUnitResponseAssumption*. Every $\textsc{SwitchLights}(i)$ step of the software component $1\textsc{WayTrafLightCtl}$ triggers in the environment immediately the corresponding event so that the execution of the $\textsc{LightUnitResponse}(i)$ rule happens immediately
  - this means that the light unit response time is negligible with respect to the beginning of the $Passed$ count when the software control $1\textsc{WayTrafLightCtl}$ enters its next phase

The *LightUnitResponseAssumption* guarantees the update consistency for the shared locations $rPulse(i)$ and $gPulse(i)$.

Analysis of refinement correctness: elements to relate

Each sw step $\textsc{SwitchLights}(i)$ is linked sequentially to an env step $\textsc{LightUnitResponse}(i)$ by *LightUnitResponseAssumption*.

In the refined 2-agent ASM model M^* a pair of sequentially linked (*successive*) steps is called a *segment of interest* in a run of M^*.

States S^* of the refined model M* and S of the abstract model M are called equivalent if

- they have the same $phase$ value and
- satisfy the same light combination, i.e. satisfy the same conjunction $showLight(i)$ **and** $showLight(j)$ with $i \neq j$
  - where $showLight(k)$ is one of $Stop(k)$ or $Go(k)$

Refinement Correctness Property

A run of the refined model M^* is *legal* if it is started in its initial state and satisfies the *TimerAssumption* and *LightUnitResponseAssumption*.

Proposition. For each legal run R^* of the refined model M^* there is a legal run R of the abstract model M such that for every natural number n the state S^* which is reached in the run R^* at the end of the n-th segment

$$[\textsc{SwitchLights}(i), \textsc{LightUnitResponse}(i)]$$

of interest is equivalent to the state S reached in the run R after the n-th step $\textsc{SwitchLights}(i)$ (a one-element run segment) of $1\textsc{WayTrafLightSpec}$.

Proof by induction on runs.

NB. One $M = 1\textsc{WayTrafLightSpec}$ step refined to two M^* steps. In general: ASM refinement type (m, n) for any natural numbers m, n.

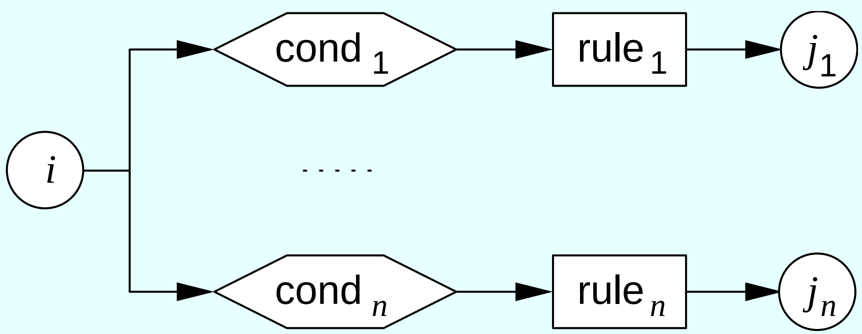
When to Use Axioms and When Local Actions

- reqs engineer together with domain expert prepare ground model and domain description for sw designer
  - grd mod Stop/Go ctl-states reflect *req phenomena* in terms of which requested 4-phase light regime is expressed directly. This supports customer/designer mediation & linking ground to sw model
- sw designer refines ground model to sw spec for programmer
  - GPulse/RPulse reflect *sw-interface phenomena*
- proving refinement correctness provides needed link bw the models: preservation of correctness and completeness of the ground model
  - having instead of logical axioms abstract local actions, which directly transform each traffic light phase into its successor phase, simplifies
    - understanding of how the pulses emitted by the sw spec for the control program are linked to their effect in the environment
    - correctness proof for this link

The underlying general concept: Control State ASMs

Control State ASM = ASM all of whose rules have the form[2]

if $ctl\_state = i$ **and** $cond$ **then** $\begin{array}{l} rule \\ ctl\_state := j \end{array}$



control-states $i, j, \ldots$ represent an overall system status (mode, phase), which allows the designer to

- *structure* the set of *states* into subsets, *visualizing* this structure
- *refine* control-state transitions by control-state submachines (modules)
  – sequentializing (overall parallel) control where needed

[2] Figure © 2003 Springer Berlin-Heidelberg, reused with permission.

General form of ASM rules: parallel execution

Finitely many rules of the following form (three equivalent notations):

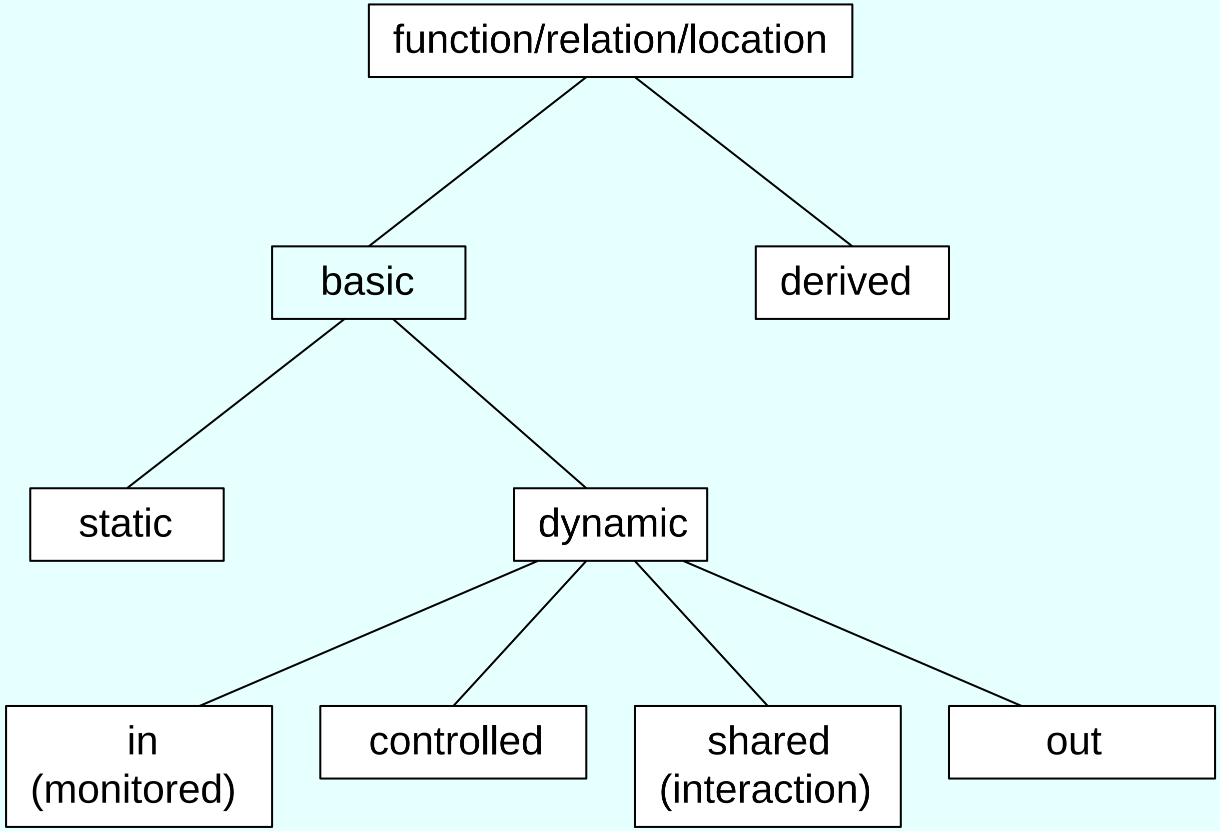
if $cond$ **then** M_1	**if** $cond$ **then** M_1	**par** $(M_1, \ldots, M_n)$
$\vdots$	$\vdots$	
M_n	**if** $cond$ **then** M_n	

NB. In one ('atomic') step all of the rules whose guard evaluates to true are executed in parallel.

- This is to avoid conceptually unnecessary sequentialization.

NB. Be careful to avoid inconsistent updates.

Classification of locations/functions



supporting the separation of concerns: information hiding, data abstraction, modularization and stepwise refinement[3]

[3] Figure © 2003 Springer Berlin-Heidelberg, reused with permission.

Characteristics of the ASM refinement concept

- ASM refinement *refines objects and data* (the states)
  - in 1WayTrafLightCtl the lights are replaced by pulses

  and operations on them
  - freedom to adapt abstraction level to design needs
- Refinement Correctness Property stipulates the equivalence only for *corresponding states of interest*
  - an intermediate segment state in the refined model (here a step reached by the first step in a segment) usually has nothing it would correspond to in the abstract model

  hiding details which are specific to refined abstraction level and cannot (or need not) be related to anything in the abstract model
- Equivalence can be any precise (not necessarily functional) relation between parts of abstract/refined model
  - reduces complexity of a precise intuition-guided formulation and difficulty of refinement verification.

Why multiple models and domain descriptions are needed

- in realistic problems, the gap between
  - *requirements penomena* that belong to the ground model
  - *sw-interface penomena* that belong to the to-be-developed-program

  has to be bridged by
  - their clear distinction as belonging to different models, at different levels of abstraction

    'a description of what you want to achieve—the *optative* properties described in the requirement and specification' (p.110)
  - an explicit statement of their refinement relation, which may be guided by props that appear in the application domain description

    'description of the domain properties that you're relying on—the *indicative* properties described in the domain description' (p.110)
- NB. Ground model abstractions and their refinement in the sw models enhance modifiability and reusability of the models (*modeling-for-change*)

Integrate new reqs by horizontal ASM refinement

Additional requirements:

OrangeLightReq. Use simultaneous Stop and Go lights to indicate 'Stop, but be prepared to Go'.

OrangeLightRegimeReq. The simultaneous Stop and Go lights period is 10 seconds and is inserted into the cycle between the Stop period and the Go period of the corresponding light.

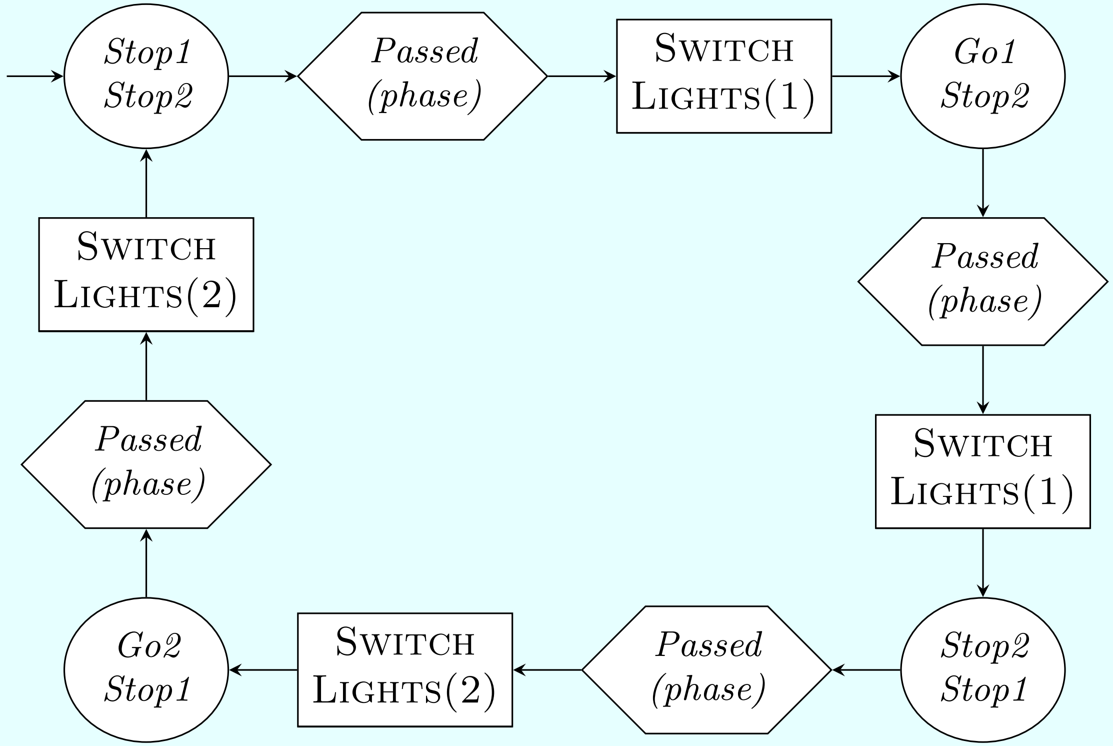
- **Additional phases** $prepareToGo1$ and $prepareToGo2$:

  $PrepareToGo(i)$ iff

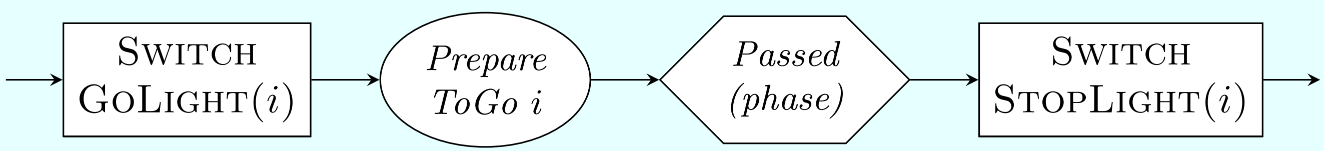
  $StopLight(i) = on$ **and** $GoLight(i) = on$ **and**

  $Stop(j)$ for $j \neq i$
- Safety: $Stop(j)$ for $j \neq i$ (2 units never simultaneously 'show Go')
- $period$ function extended to take as value 10 seconds for new phases (data refinement with extended *TimerAssumption*)

Refined SwitchLights(i) in 1Way3ColorTrafLightSpec



Upper and lower occurrences of the SwitchLights component are replaced by refined control state ASM of refinement type (1,2):[4]



[4] Figures © 2010 Springer Berlin-Heidelberg, reused with permission.

Same refinement applicable at refined component level

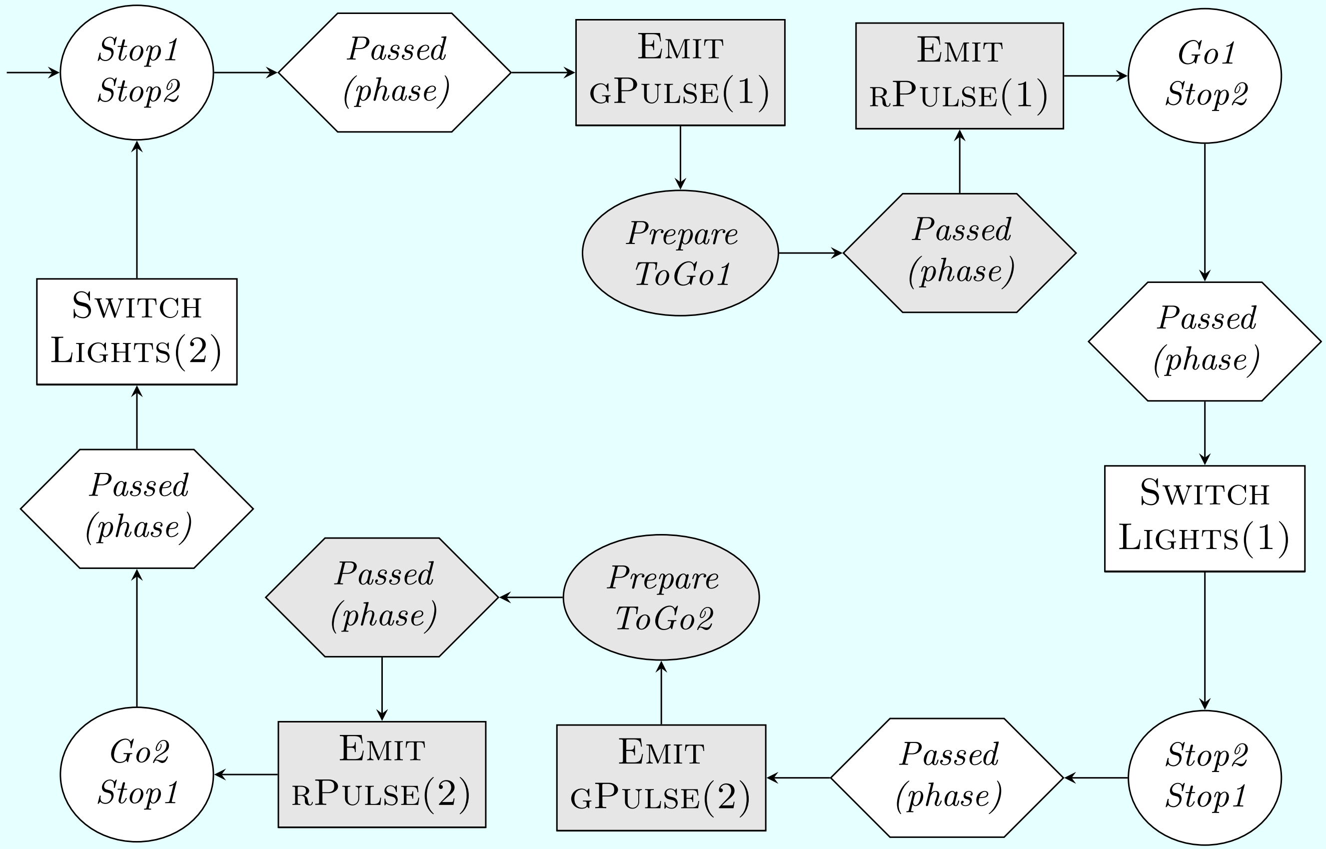
Similarly refine $1\textsc{WayTrafLightCtl}$ by refining its two $\textsc{SwitchLights}$ components in question, using:

- $\textsc{Emit}(gPulse(i))$ instead of $\textsc{Switch}(GoLight(i))$
- $\textsc{Emit}(rPulse(i))$ instead of $\textsc{Switch}(StopLight(i))$

The $\textsc{LightUnitResponse}$ environment ASM remains unchanged.

Rephrase the refined Timer Assumption, taking into account *period* for the new phase, and formulate and prove the Refinement Correctness Property.

1WAY3COLORTRAFLIGHTCTL by added Green&Red-lights



NB. Such *horizontal ASM refinements* add new features to the given models but do not change the level of abstraction. *Vertical ASM refinements* add details at refined levels of abstraction.[5]

[5] Figure © 2018 Springer-Verlag Germany, reused with permission.

Model reuse: Two-Way Traffic Light Control

Requirements (Abrial 2010):

PlantReq. We intend to install a traffic light at the crossing between a main road and a small road ... in such a way that the traffic on the main road is somehow given a certain advantage over that on the small road.

MainRoadPriority. When the light controlling the main road is green, it only turns ... red ... when some cars are present on the small road (the presence of such cars is detected by appropriate sensors) ... provided that road has already kept the priority for at least a certain (long) fixed delay.

SmallRoadAllowance. ... the small road, when given priority, keeps it as long as there are cars willing to cross the main road ... provided a (long) delay (the same delay as for the main road) has not passed. When the delay is over, the priority systematically returns back to the main road.

Reuse idea

Re-interpret the concept of *1-way control*

- providing the 'permission to pass on the road in direction i' for two opposite directions $i = 1, 2$

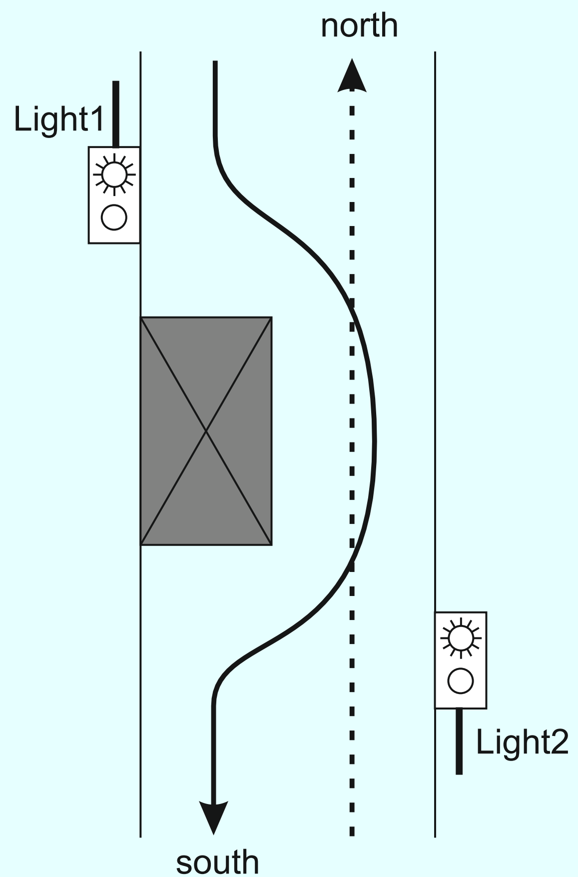
as a 2-way control

- providing the 'permission to pass on road i'

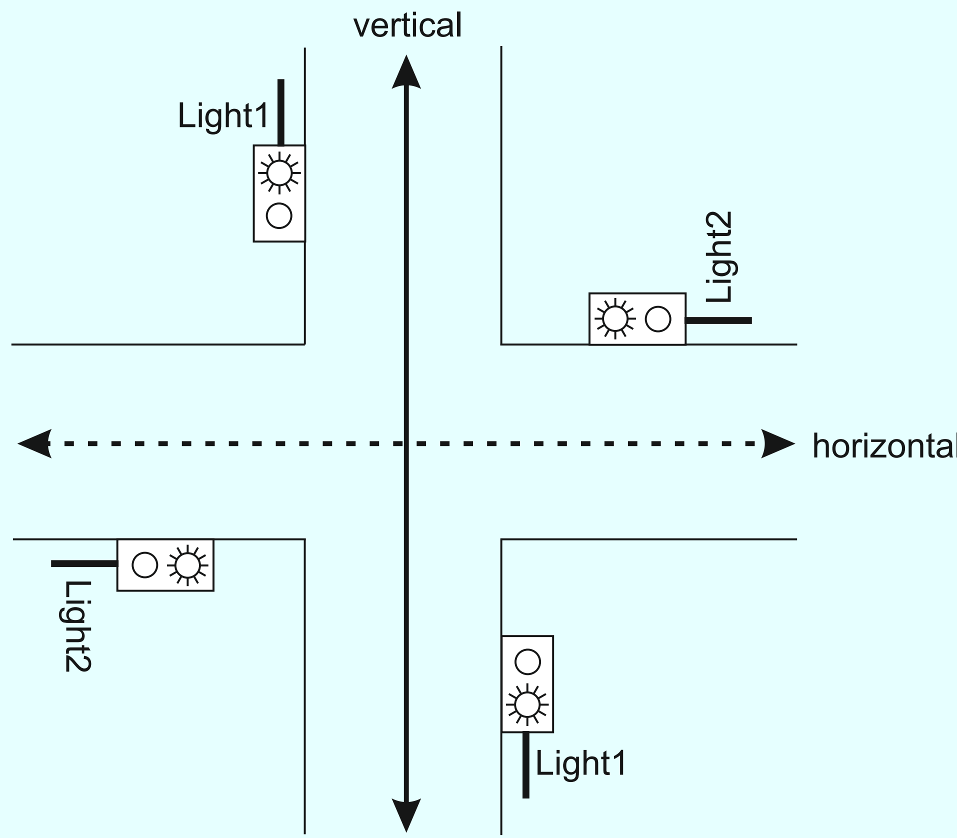
where 1 stands for a main and 2 for a secondary road which cross each other.

In other words: interpret the lights for the two *exclusive directions* as lights for the (two directions of the) *main road* respectively for the (two directions of the) *small road*.

Different interpretations of 'permission to go'



1-Way: Go either
north or south



2-Way: Go either
horizontal or vertical

6

[6] Figure © 2018 Springer-Verlag Germany, reused with permission.

Reuse $1\textsc{WayTrafLightSpec}$: a data refinement

- $Go(1)$ is re-interpreted as 'the main road is green'
- $Go(2)$ is re-interpreted as 'the small road is green'
- analogously for $Stop(1)$ and $Stop(2)$

Incorporate *MainRoadPriority* and *SmallRoadAllowance* by refining the monitored $Passed$ predicate

- adding for the two relevant $phase$s (but not for the other two) the car presence conditions to the time constraints

Sensor detecting the presence of cars on the small road is represented by a monitored Boolean-valued function $CarsOnSmallRoad$

Data refinement of $Passed$

if $phase = \overset{Go1}{Stop2}$ **then** // we are in main road Go phase

$Passed(phase)$ iff

$Elapsed(period(phase))$ **and** $CarsOnSmallRoad$

if $phase = \overset{Go2}{Stop1}$ **then** // we are in small road Go phase

$Passed(phase)$ iff

$Elapsed(period(phase))$ **or**

$NoCarsOnSmallRoad$

Delays mentioned in *MainRoadPriority* and *SmallRoadAllowance* represented by value of $period(phase)$ for the two phases in question.

2WayTrafLightSpec correctness: formulation and proof

Link bw sensor actions in env and the effect they produce in the model can be described by a run constraint (or by an env ASM):

- *SensorAssumption*. Whenever the sensor detects a car on the small road, the environment sets $CarsOnSmallRoad$ immediately to true; the value remains unchanged until the sensor detects that there is no car on the small road. This is the moment in which the environment resets $CarsOnSmallRoad$ to false.

Combining the already justified 1WayTrafLightSpec correctness property with model inspection for the new Two-Way Traffic Light features supports the correctness claim:

Correctness Property: each legal run of 2WayTrafLightSpec satisfies the above *2WayTrafLightReq*uirements.

Proof: induction on runs

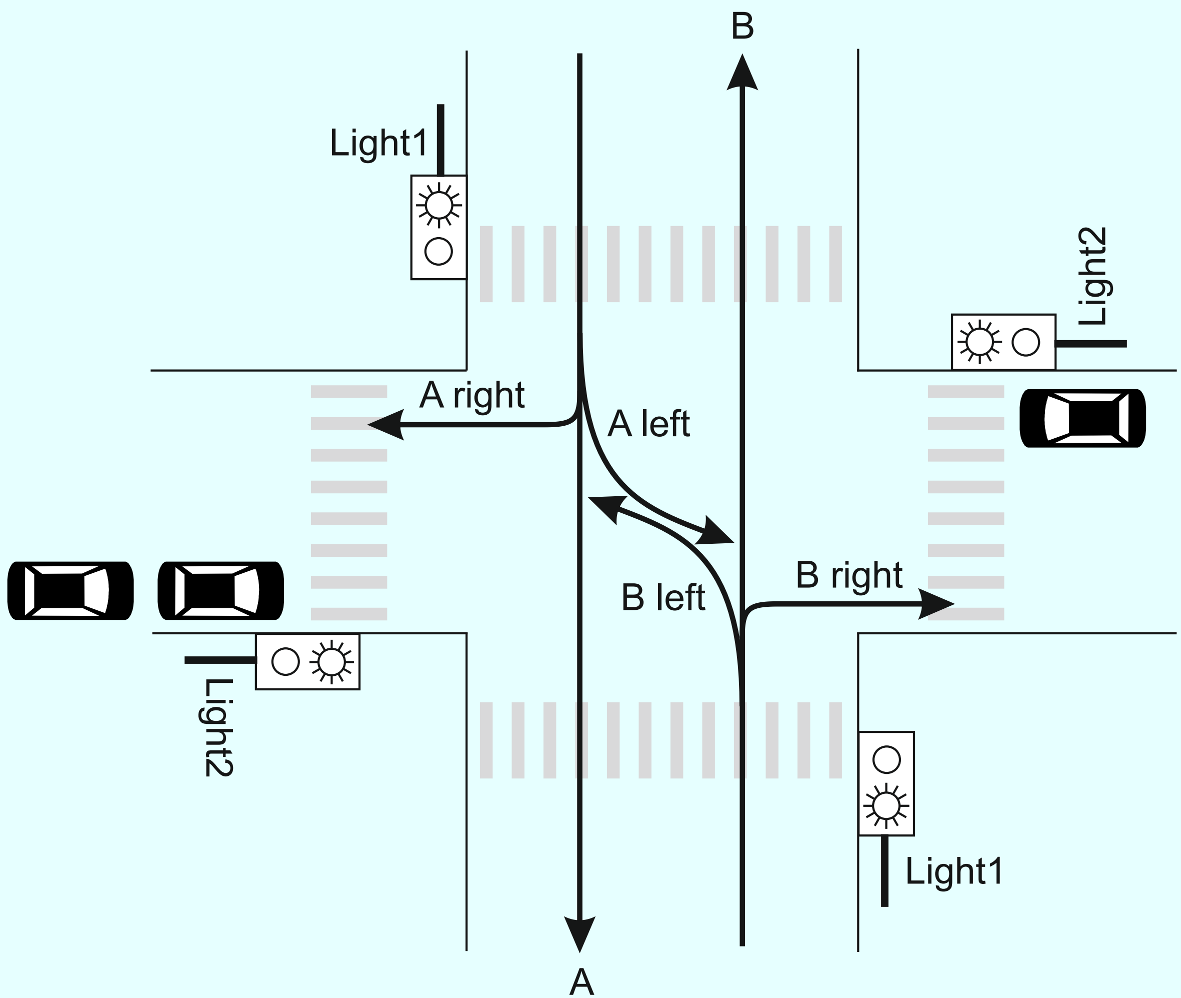
Separation of two different concerns

- reuse the one-way controller for two-way traffic lights
  - via reinterpretation of direction from ‘one of two opposite directions on the same road‘ in $\{dir(road), dir'(road)\}$ to a pair of the two opposite directions of each of the two crossing roads $\{(dir(main), dir'(main)), (dir(small), dir'(small))\}$

  linking one-way controller to two traffic lights (one for each direction) and two-way controller to two pairs of traffic lights, one pair for each road (components of each pair show the same light behaviour for the two opposite road directions).
- priority policy
  - modifiable by just redefining $Passed$ predicate, without changing the control program for the corresponding light sequence

This is a typical way ASM abstraction and refinement can be exploited to support the separation of concerns.

Model reuse for extension with traffic law regulation



[7]

[7] Figure © 2018 Springer International Publishing AG Switzerland, reused with permission.

Vertical timer refinement for 2WAYTRAFLIGHTSPEC

Design goal: replace *TimerAssumption* on external clock signal by an internal timing mechanism

Needed: a local $timer$ function which

- is SET to the current value of the monitored system clock now upon entering every to-be-timed $ctlstate$ and
- is used to check by the refined $Passed$ predicate whether the delay $time$ in question has $Elapsed$, i.e. whether $now - timer \geq time$

The system $clock$ is assumed to be monotonically increasing and measured in terms which are compatible with the terms used to formulate the length of $period$ for the $phase$s in question.

Internal timer module for 2WayTrafLightSpec

For Timed2WayTrafLightSpec it suffices to:

- add to each SwitchLights component a parallel timer component SetTimer
  - also written Set($timer$) to distinguish this $timer$ location from other timer locations

  SetTimer $= (timer := now)$

  Initially $timer = now$ -- initialization condition
- data redefine $Elapsed(phase)$ not as monitored but as derived fct
  - of the variable now, the controlled variable $timer$ and the static function $period$

  $Elapsed(phase)$ iff $now - timer \geq period(phase)$

For the rest just copy 1WayStopGoLightSpec from above (with unchanged SwitchLights(i) components):

1WayStopGoLightSpec $\rightsquigarrow$ Timed2WayTrafLightSpec

if $phase \in \{Stop1\,Stop2, Go1\,Stop2\}$ **and** $Passed(phase)$

then

SwitchLights(1)

SetTimer

if $phase = Stop1\,Stop2$ **then** $phase := Go1\,Stop2$

else $phase := Stop2\,Stop1$

if $phase \in \{Stop2\,Stop1, Go2\,Stop1\}$ **and** $Passed(phase)$

then

SwitchLights(2)

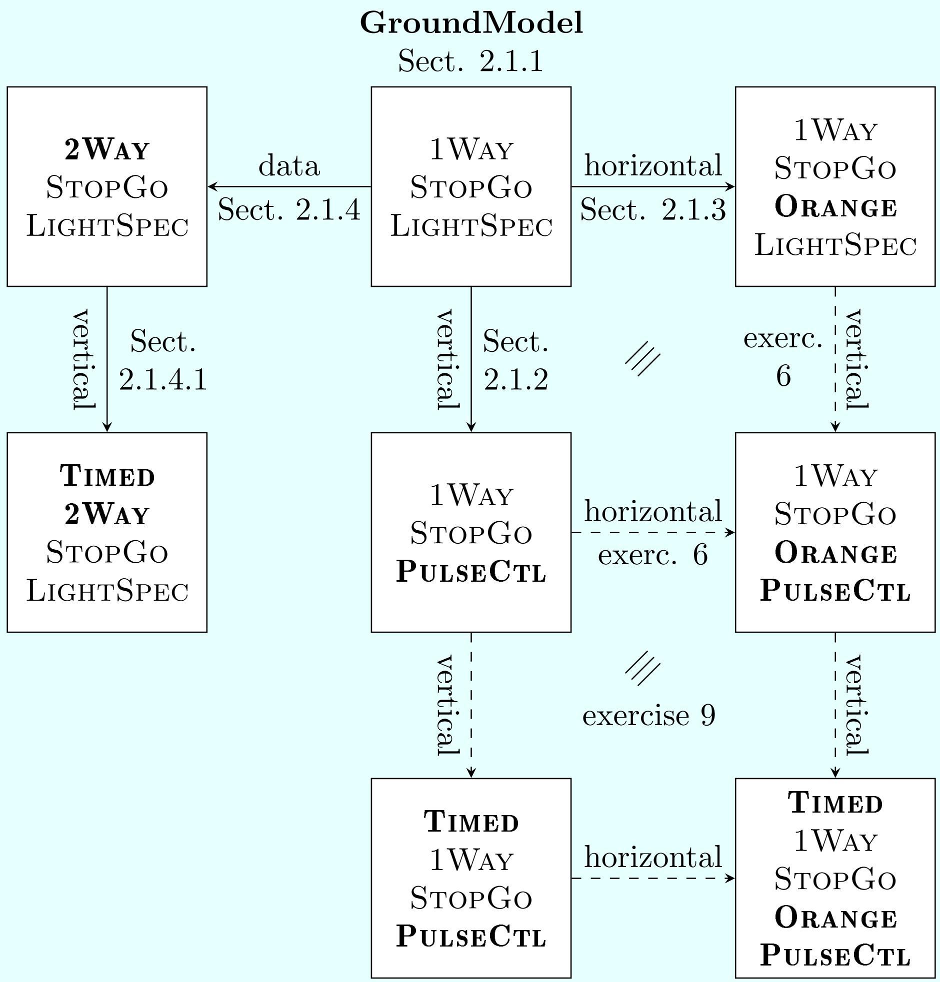
SetTimer

if $phase = Stop2\,Stop1$ **then** $phase := Go2\,Stop1$

else $phase := Stop1\,Stop2$

NB. The rule is literally copied from 1WayStopGoLightSpec!

Ground Model Refinements: Survey



[8]

[8] Figure © 2018 Springer International Publishing AG Switzerland, reused with permission.

How to combine requirements via refinement

Superposition problem: How are ... two requirements put together to form the combined requirement, and what are the effects elsewhere in the development? (M. Jackson op.cit. p.217)

An answer: use refinement and track its effect in the model hierarchy.

Illustration by OPERATED1WAYTRAFLIGHT:

An alternative new version of the one-way traffic lights provides for a traffic overseer who can override the default regime of Stop and Go lights. The machine is equipped with two buttons marked 'Hold' and 'NextPhase'. The overseer can extend the current phase of the light sequence by pressing the Hold button, or curtail it by pressing the NextPhase button. Pressing a button causes a pulse shared by the machine.

The rules for modifying the default behaviour may be something like this [we write NextPhase instead of Change in op.cit.]:

Operated One-Way Lights Requirements

- *FstHoldReq*: on a Hold cmd the current phase is extended from the point already reached by its default length
- *NextPhaseReq*: on a NextPhase cmd the current phase is terminated and the next phase is immediately begun
- *SndHoldReq*: if two Hold cmds are issued within one second, the current phase is extended until a NextPhase cmd is issued.
- *OtherHoldReq*: Other Hold cmds issued after the first Hold in a phase, and before the end of the phase or a NextPhase cmd, are ignored. (p.217-218)

Combining automated control with operator commands

Conflicts bw the new requirements and the original ones are typical.

Let TrafLightPgm be any of the above (ground or refined) traffic light models. We show how to combine it with an Operator command model to a new machine where conflicts are resolved:

OperatedTrafLight =

Operator -- with priority to 'override' automated behavior

if not $UnderOperatorCtl$ **then** TrafLightPgm

The interface *UnderOperatorCtl* must be defined in such a way that

- the automated control TrafLightPgm performs the light control also for commands issued by the operator, where not leading to an inconsistency
- conflicts between updates requested by operator commands and by the automated control TrafLightPgm are resolved

OPERATOR =

if *Event(cmd)* **then** -- events triggered by the operator
  if $FstHold(cmd)$ **then**
    EXTEND($phase$) -- to satisfy *FstHoldReq*
    RECORDFSTHOLD -- to recognize a $SndHold$ cmd
  if $SndHold(cmd)$ **then**
    AWAITCMD($NextPhase$) -- to satisfy *SndHoldReq*
    RECORDSNDHOLD -- to recognize ‘Other Hold cmds’
  if $NextPhase(cmd)$ **then**
    SWITCHTONEXT($phase$) -- to satisfy *NextPhaseReq*
    RESETOPCMDRECORD -- to return to main pgm
  CONSUME($Event(cmd)$)

Operator submachines

$FstHold(cmd)$ **iff** $cmd = Hold$ **and** $fstHold = -\infty$

$\textsc{RecordFst/SndHold} = (fst/sndHold := now)$

$Extend(phase) = (timer := now)$ -- restart current phase

-- 'current phase is extended ... by its default length'

$SndHold(cmd)$ **iff** $cmd = Hold$ **and** $sndHold = -\infty$ **and**

$0 < now - fstHold \leq 1\ sec$ -- NB. implies $fstHold \neq -\infty$

$\textsc{AwaitCmd}(NextPhase) = (WaitingForNextPhaseCmd := true)$

-- also derivable from $sndHold \neq -\infty$

$NextPhase(cmd)$ **iff** $cmd = NextPhase$

$\textsc{ResetOpCmdRecord} =$

$fstHold := -\infty$ $sndHold := -\infty$

$WaitingForNextPhaseCmd := false$

Interpretation of 'other Hold commands'

- *Other Hold cmds issued after the first Hold in a phase, and before the end of the phase or a NextPhase cmd, are ignored.*

This requirement seems to say that those 'other Hold cmds' satisfy:

$fstHold > -\infty$ -- cmd comes after a first Hold cmd

and

$(0 < now - fstHold > 1sec$ -- too late for a snd Hold cmd

or $sndHold > -\infty)$ -- or after the snd Hold cmd

NB. Upon entering a new $phase$, ResetOpCmdRecord will reset $fstHold$ and $sndHold$ to $-\infty$ and $WaitingForNextPhaseCmd$ to false

- either by Operator executing the $NextPhase(cmd)$ or by the refined TrafLightPgm)

Possible Conflicts bw OPERATOR and TRAFLIGHTPGM

- If $NextPhase(cmd)$ happens exactly when $Passed(phase)$, then the OPERATOR command requests the same update of $phase$, $timer$ (with corresponding light updates SWITCHTO...) as the ones TRAFLIGHTPGM is defined to perform.
- But what should happen if an $Event(Hold)$ happens when $Passed(phase)$?
  - $Event(Hold)$ requests to $\text{EXTEND}(phase)$
  - TRAFLIGHTPGM is defined to pass to $next(phase)$

There are various *options* the customer must decide upon.

Solving Conflicts bw OPERATOR and TRAFLIGHTPGM

- Hold cmd is ignored
  - e.g. by guarding the OPERATOR rule additionally with **not** $Passed(phase)$. TRAFLIGHTPGM enters the new phase.
- $phase$ update by TRAFLIGHTPGM is ignored
  - e.g. by adding to the guard $Passed(phase)$ the conjunct **not** $Event(Hold)$. Then the new phase is not entered and the current one is EXTENDed by the OPERATOR rule.
- Hold cmd affects the new phase
  - entered by $phase$ update rule of TRAFLIGHTPGM
  - OPERATOR is executed for $next(phase)$
    - e.g. by adding a special OPERATOR rule for this case

For the sake of illustration we decide to give priority to OPERATOR cmds and to restrict TRAFLIGHTPGM control to cases which are compatible with OPERATOR cmds.

$UnderOperatorCtl$ **interface definition**

The predicate should disallow TrafLightPgm to make a step

- when an Operator cmd is issued
  - i.e. in case $Event(cmd)$ holds for some cmd
- **or** when the system is $WaitingForNextPhaseCmd$
  - period in which the traffic lights are required not to change

Formally (remember ResetOpCmdRecord is executed upon entering a new $phase$) we define:

$UnderOperatorCtl$ **iff**

forsome $cmd \in \{Hold, NextPhase\}\ \ Event(cmd) = true$

or $WaitingForNextPhaseCmd = true$

It then suffices to refine TrafLightPgm by

adding ResetOpCmdRecord

to each rule which updates the $phase$.

Relating button pressing to machine events

Command Event Assumption. For each $cmd \in Cmd$, $Pressed(button(cmd))$ (in the real world) implies that $Event(cmd)$ immediately becomes true in $\textsc{Operator}$ model.

Apparently also the *at-most-one-cmd-per-time assumption* is tacitly made. Probably together with other assumptions, e.g. that default $length(phase) > 1\ sec$.

- in $\textsc{OperatedTrafLight}$, an instantaneous atomic (0-time) execution of rules is assumed so that if two $Hold$ cmds fire within 1 sec but in different phases, meantime $\textsc{ResetOpCmdRecord}$ happened during the phase change.

To complete the model, all such assumptions have to be listed, providing a complete basis for analysis.

- NB. Proof failure often indicates missing (forgotten) assumptions!

Recap: Modeling and refinement steps

- ground model 1WayTrafLightSpec for *functional behavior*
- vertical ASM refinement
  - to *separate computer and env* actions
    - 2-agent model: 1WayTrafLightCtl, LightUnitResponse
  - to *implement TimerAssumption*
    - replace monitored external time by internally computed time
- horizontal ASM refinement to *add new requirement* (for orange light)
  - 1Way3ColorTrafLightSpec/Ctl
- model reuse by *data refinement*: $2WayTrafLightSpec/Ctl$
- Combining requirements (superposition problem), *resolving conflicts* by appropriate ASM refinements

Exercise in function classification

$\textsc{TwoWayFsm}(nextMode, write, move) =$

$mode := nextMode(mode, input(head))$ -- update internal state

$out := write(mode, input(head))$ -- print output

$head := head + move(mode, input(head))$ -- move reading head

$\textsc{TuringMachine}(nextMode, write, move) =$

$mode := nextMode(mode, tape(head))$

$tape(head) := write(mode, tape(head))$ -- update $tape$ cell

$head := head + move(mode, tape(head))$

$\textsc{InteractiveTuringMachine}(nextMode, write, move) =$

$mode := nextMode(mode, tape(head), input)$

$tape(head) := write(mode, tape(head), input)$

$head := head + move(mode, tape(head), input)$

$out := output(mode, tape(head), input)$

References

- M. Jackson: Problem Frames. Addison-Wesley 2001
- C. A. Gunter, E. L. Gunter, M. Jackson, P. Zave: A Reference Model For Requirements and Specifications. IEEE Software, May/June 2000
- J.-R. Abrial: Modeling in Event-B: System and Software Engineering. Cambridge University Press 2010
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018 http://modelingbook.informatik.uni-ulm.de