Egon Börger (Pisa) & Alexander Raschke (Ulm)

A Sluice Gate Control Model

ASM model reuse via ASM refinements

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

See Ch. 2.2 of Modeling Companion

Sluice Gate Reqs (M. Jackson: Problem Frames p.49)

PlantReq A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to control the sluice gate.

FunctionalReq. The requirement is that the gate should be held in the fully open position for ten minutes in every three hours and otherwise kept in the fully closed position.

MotorReq. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses.

SensorReq. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut

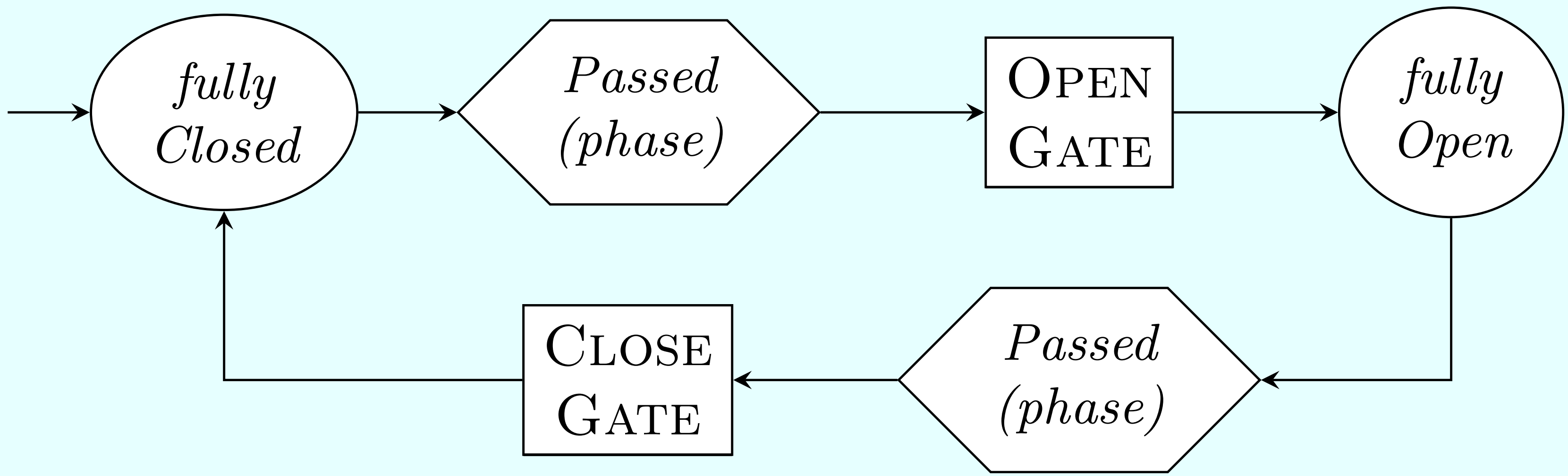
PulseReq. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensors.

Sluice Gate elements: the signature

- a controlled variable (0-ary function) *phase* indicating the current phase with possible values $fullyOpen, fullyClosed$,
- a controlled variable *gatePos* $\in \{fullyOpen, fullyClosed\}$ indicating the current (no intermediate!) gate position,
- as for $1\textsc{WayTrafLightSpec}$:
  - a derived time signal $Passed(phase) = Elapsed(period(phase))$
  - a monitored timer function $Elapsed$ coming with the *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.
- static functions $interval = 3\,h$ and $period(fullyOpen/Closed)$ (called $open/closedPeriod$) with value 10 resp. $170\,min$ satisfying

$$interval = period(fullyClosed) + period(fullyOpen)$$

Functional Behavioral Ground Model SLUICEGATESPEC



Abstract from motor, screws, sensors, connection bw computer and gate:

$\text{OpenGate} = (gatePos := fullyOpen)$

$\text{CloseGate} = (gatePos := fullyClosed)$[1]

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

Initialization and Correctness

InitReq. The sluice gate initially is in the fully closed position.

Correspondingly we define the initial state S_0 in the ASM model as follows, with the external time count $Elapsed$ assumed to be started in the initial state:

$phase = fullyClosed$ **and** $gatePos = fullyClosed$

Legal runs are started in the initial state and satisy the above *TimerAssumption*.

Correctness Property: each legal run of SluiceGateSpec satisfies the *FunctionalReq* with $openPeriod = 10min$, $closedPeriod = 170min$.

Easily justified due to the atomicity of actions Open, Close.

Vertical refinement (driven by domain knowledge)

MotorReq and *SensorReq* express domain knowledge on how the gate is moved by a motor with the help of screws and sensors.

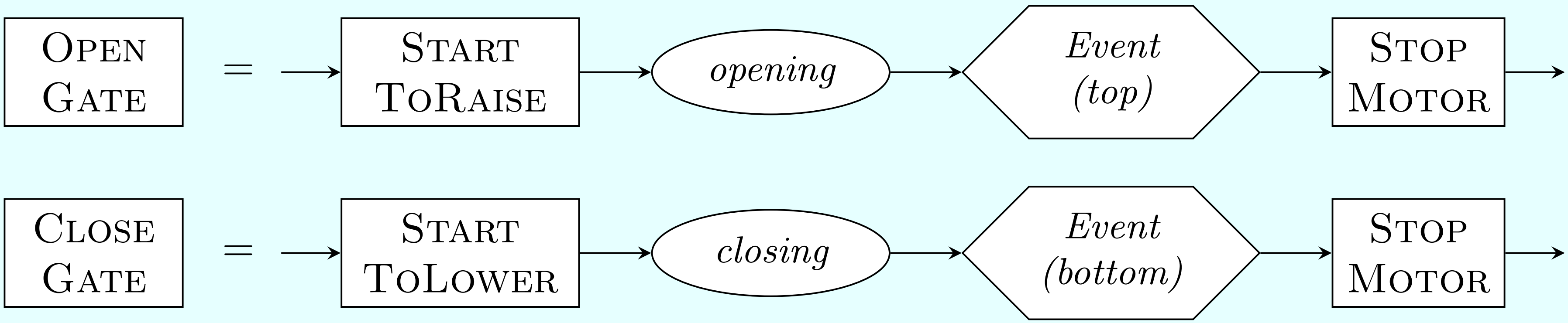
New signature elements represent this knowledge, still abstracting from how the computer is connected to the motor:

- controlled variables indicating
  - current *motorStatus* $\in \{on, off\}$
  - current *moveDir* $\in \{clockwise, anticlockwise\}$
- monitored (parameterized) variables *Event(top)*,*Event(bottom)* assumed to signal that the gate in the real world did reach its top/bottom position (see *GateMotorAssumption* below).

Wlog assume that moving up/down is realized by turning the screw clockwise/anticlockwise.

InitReq: add $motorStatus = off$, $moveDir = clockwise$

OpenGate, CloseGate for MotorDrivenSluiceGate



StartToRaise =

$moveDir := clockwise$ **par** $motorStatus := on$

StartToLower =

$moveDir := anticlockwise$ **par** $motorStatus := on$

StopMotor = $(motorStatus := off)$[2]

But what about the FunctionalReq for MotorDrivenSluiceGate?

$interval = closedPeriod + openPeriod?$

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

Completing FunctionalReq for MotorDrivenSluiceGate

$$interval = closedPeriod + openPeriod + openingTime + closingTime$$

Keeping $interval = 3h$ we must decide upon the duration of the opening/closing phases and where to put them, for example:

$0 < closedPeriod < 170\ min$ **and** $0 < openPeriod < 10\ min$

$$closedPeriod = interval - (openPeriod + openingTime + closingTime)$$

Legal MotorDrivenSluiceGate runs must satisfy the following:

GateMotorAssumption. If after any StartToRaise/Lower step the motor remains on in the corresponding direction, at the latest after $opening/closingTime$ the $Event(top/bottom)$ happens, namely when the gate has reached its final position $gatePos = fullyOpen/Closed$.

Refinement correctness for MotorDrivenSluiceGate

Correctness Property. MotorDrivenSluiceGate correctly refines SluiceGateSpec and each of its legal runs satisfies the *MotorReq*uirement and the *SensorReq*uirement.

- MotorDrivenSluiceGate in each 3-h-interval moves from $fullyClosed$ to $fullyOpen$ and back to $fullyClosed$. It
  - stays $fullyClosed$ for $closedPeriod$ and then after at most $openingTime$ enters phase $fullyOpen$
  - stays $fullyOpen$ for $openPeriod$ and then after at most $ClosingTime$ enters phase $fullyClosed$

Refinement Type

Refinement type is (1,2), meaning that:

- every segment consisting of one single step OPEN resp. CLOSE in SLUICEGATESPEC is refined by
- a segment of two corresponding MOTORDRIVENSLUICEGATE steps:
  - a step STARTTORAISE resp. STARTTOLOWER together with entering the intermediate ctl_state $opening$ resp. $closing$, followed by
  - one STOPMOTOR step together with entering the main ctl_state $fullyOpen$ resp. $fullyClosed$

Vertical refinement $\textsc{SluiceGatePulseControl}$

PulseReq. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensors.

Domain knowledge provides info for a refinement which separates sw control from physical motor reaction.

Output location $pulseLine(e)$ represents where to output the pulse e:

$\textsc{StartToRaise}/\textsc{Lower} =$

$\quad \textsc{Emit}(Pulse(clockwise/anticlockwise))$

$\quad \textsc{Emit}(Pulse(motorOn))$

$\textsc{StopMotor} = \ \textsc{Emit}(Pulse(motorOff))$

where

$\qquad \textsc{Emit}(Pulse(e)) = (pulseLine(e) := high)$ -- output to pulse line

Environment interacting with SluiceGatePulseCtl

For the sake of correctness analysis, we define an env ASM to describe the physical equipment actions when pulses e appear on the $pulseLine(e)$ (location monitored by MotorResponse):

MotorResponse $=$

if $Event(e)$ **then**

if $e = clockwise/anticlockwise$ **then**

$moveDir := clockwise/anticlockwise$

if $e = motorOn/motorOff$ **then** $motorStatus := on/off$

Consume(e)

where

$Event(e)$ **iff** $pulseLine(e) = high$

Consume$(e) = (pulseLine(e) := low)$

Assumptions Relating Software and Environment

A mechanism is needed to relate software and physical components:

- *PulseOutput Assumption*: each $\textsc{Emit}(Pulse(e))$ in the computer $\textsc{SluiceGatePulseCtl}$ yields $Event(e)$ to immediately happen in the environment $\textsc{MotorResponse}$
  - NB. Treating $pulseLine(e)$ as a shared location—output location for $\textsc{SluiceGatePulseCtl}$ and monitored location for $\textsc{MotorResponse}$—implies interpreting 'immediate $\textsc{MotorResponse}$' as letting perform its step before $\textsc{SluiceGatePulseCtl}$ emits a new pulse.
- Similarly, the *GateMotorAssumption* on $Event(Top/Bottom)$ is assumed to be satisfied in the presence of the status line transmission of sensor values.

Proving Correctness of the Pulse Refinement

This refinement is of type (1,2):

- one abstract step $\textsc{StartToRaise}/\textsc{Lower}$ of $\textsc{MotorDrivenSluiceGate}$ corresponds to
- two refined steps
  - $\textsc{Emit}(Pulse(clockwise/anticlockwise))$
    $\textsc{Emit}(Pulse(motorOff/On))$
    of $\textsc{SluiceGatePulseCtl}$ followed by
  - a $\textsc{MotorResponse}$ step updating $moveDir$ and $motorStatus$
- analogously for $\textsc{StopMotor}$ steps

They have equivalent effect wrt setting $moveDir$, $motorStatus$ and ctl_state. Therefore the correctness property holds—adapted to consider the possible time delay (if any) between computer and env steps.

Adding requirements by data refinement

MotorDecelerationReq: for some $motorDecelarationTime$, gate may still move after the motor has been turned off when moving up/down.

Solution by a pure data refinement:

- add $motorDecelarationTime$ twice to $interval$ (once per stopping moving up/down)

  $$interval = closedPeriod + openPeriod \\ + openingTime + closingTime + 2 \times motorDecelarationTime$$

- maintain as $fullyOpen/Closed$ position the one reached in $opening/closingTime$ with respect to which the difference of the $gatePos$ition that is reached by deceleration can be neglected
- reformulate Correctness Property adding $motorDecelarationTime$ to $closed/openPeriod$

NB. Alternative: operation refinement based upon additional sensors which report the complete gate movement stop: domain experts decide!

Model reuse for SluiceGateOperator

Requirements: *MotorReq*, *SensorReq* as in *SluiceGateReq*.

Changes for the other requirements:

PlantReq. ... to raise and lower the sluice gate in response to the commands of an operator.

FunctionalReq. ... the operator can position the gate as desired by issuing Raise, Lower and Stop commands: the machine should respond to a Raise by putting the gate and motor into a Rising state, and so on ... The Rising and Falling states are mutually exclusive.

PulseReq. ... and a status line for each class of operator command.

Multiple command problem

Idea: replace time-triggered raise/lower transitions by operator-command-triggered ones

- replace time events $Passed(fullyClosed/Open)$ by command events $Event(Raise/Lower)$,
- rename phases $opening/closing$ to $rising/falling$,

Replacing checks of $Elapsed(time)$ (governed by timer rules or assumptions) by guards checking whether a command has been issued leads to a general question:

- How to discipline command issuing by the operator?
- How to prevent the machine from executing some in a given context undesired but issued command?

Solutions: declaratively (constraining legal runs by excluding certain command sequences) or operationally.

How to Discipline Operator Commands

In a math model one can always constrain runs to exclude certain cmd sequences. In real-life, one can issue operating instructions, but one cannot rely upon such instructions being followed always perfectly.

- Event classification ('Reasons for disobedience' op.cit. p.112)
  - *not sensible* cmd: 'makes no sense in the context of preceding cmds'
  - *not viable* cmd: 'inappropriate or impermissible in the current state'
  - *not overrunnable* cmd: the response to the cmd has to be finished before other cmds come in (Ch.9.2 op.cit. on 'overrun concern' due to mismatch of speeds bw triggering an action and its exec)
    - to be resolved by *inhibition*, *ignoring* or *buffering* events that occur when the machine is not ready to participate in them

Example for cmd sequence constraints

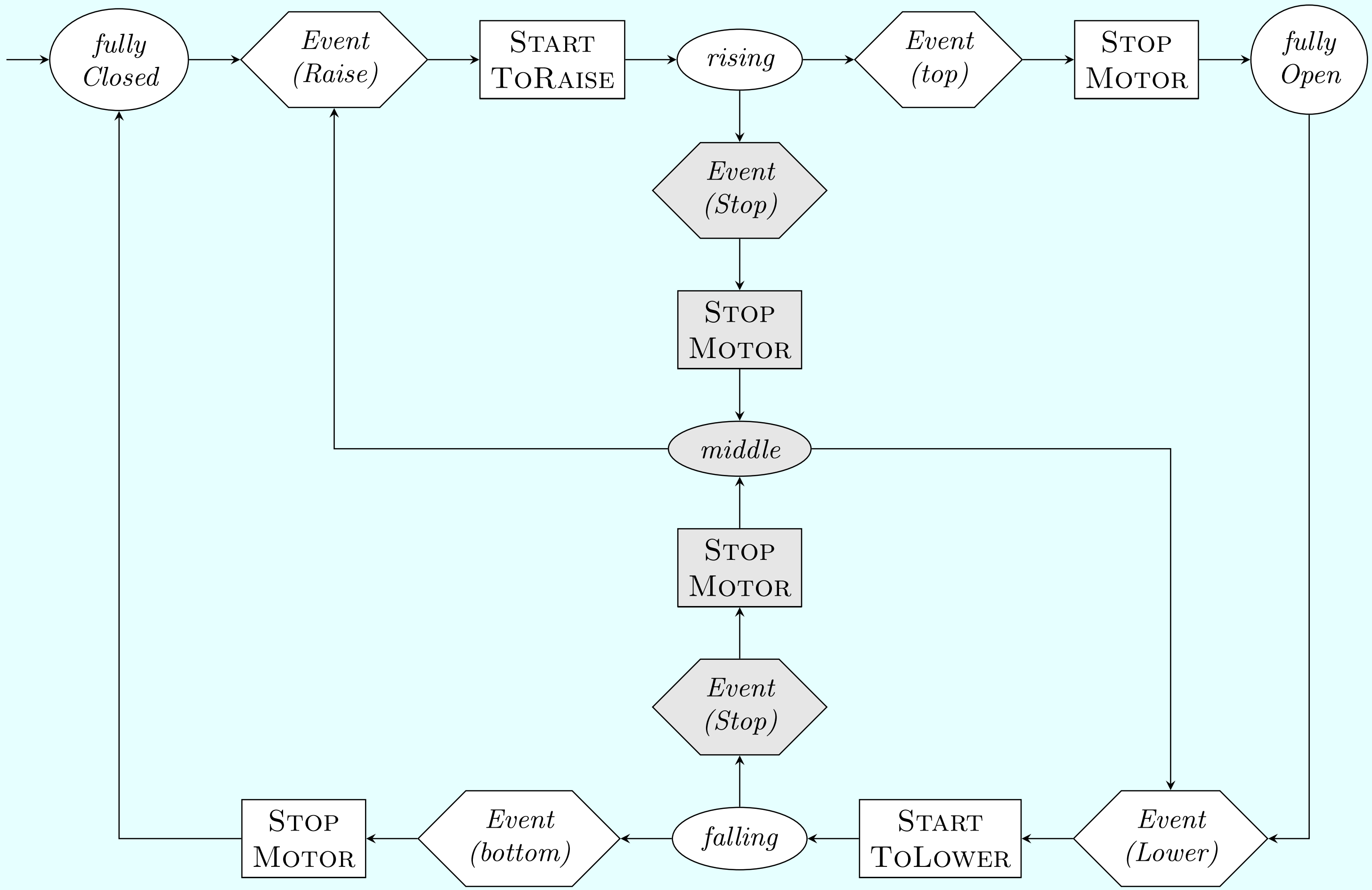
*CommandSequenceReq*uirement:

- at each moment at most one command is issued: a reasonable assumption if there is only one operator
- only successive command pairs (*Raise,Stop*) or (*Lower,Stop*) or vice versa (*Stop,Raise*) or (*Stop,Lower*) make sense

Then one can reuse the $\textsc{SluiceGateSpec}$ rules as follows:

- replacing time guard $Passed(phase)$ by $Event(cmd)$
- adding $\textsc{Consume}(cmd)$ to $\textsc{Perform}(cmd)$ where
  $\textsc{Perform}(Raise/Lower) = \textsc{StartToRaise}/\textsc{Lower}$
  $\textsc{Perform}(Stop) = \textsc{StopMotor}$

SluiceGateOperator for CommandSequenceReq



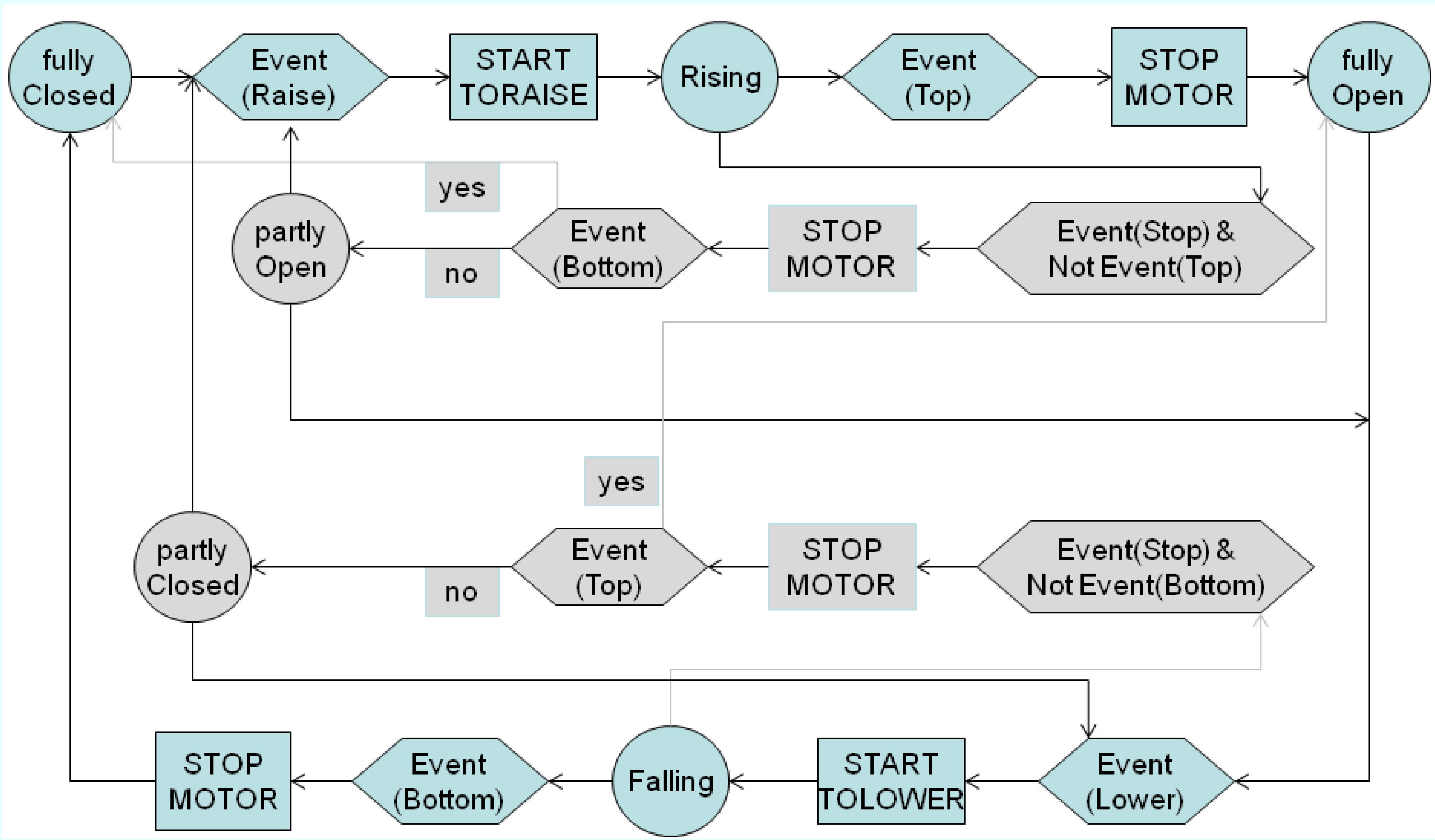
NB. This machine in each *phase* reacts only to certain events. No warning is reported for unforseen events.[3]

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

InertialEffectRequirement

If $\text{Event}(\text{Raise}/\text{Lower})$ is immediately followed by $Event(Stop)$, then the gate travel may not yet have started

- then the correct control state to navigate to is $fullyClosed/Open$



CommandSequenceRequirement implications

SluiceGateOperator should 'reject' as insensible the second command in any of the following command pairs, should one of them be issued:

- Raise, Lower: possibly not viable for physical reasons,
- Raise, Raise: not viable, including 'no Raise in top position',
- Lower, Raise: possibly not viable for physical reasons,
- Lower, Lower: not viable, includes 'no Lower in bottom position',
- Stop, Stop: Stop when Stopped not reasonable.

For robustness concerns add the case of a Lower/Raise command issued when the gate is in its bottom/top position.

We interpret rejection as a) 'not executing' the to-be-rejected command and b) 'notifying' the appearance of the insensible command sequence.

ROBUSTSLUICEGATEOPERATOR

$= \text{SLUICEGATEOPERATOR}$ **par** $\text{DETECTINSENSIBLECMD}$

$\text{DETECTINSENSIBLECMD} =$

if $phase \in \{fullyClosed, fullyOpen, middle\}$ **then**

$\text{REJECT}(Stop, phase)$ -- no $StopStop$

if $phase = fullyClosed$ **then** $\text{REJECT}(Lower, phase)$

-- no $Lower$ at bottom

if $phase = fullyOpen$ **then** $\text{REJECT}(Raise, phase)$

-- no $Raise$ at top

if $phase \in \{rising, falling\}$ **then**

$\text{REJECT}(Lower, phase)$ -- No RaiseLower, no LowerLower

$\text{REJECT}(Raise, phase)$ -- No RaiseRaise, no LowerRaise

$\text{REJECT}(cmd, phase) =$ **if** $Event(cmd)$ **then**

$\text{REPORT}(cmd, phase)$ $\text{CONSUME}(cmd)$

REPORT of erroneous commands case-wise definable

The same way one can separate normal from exceptional behavior.

$\textsc{Report}(cmd, phase) =$

  if $cmd = Stop$ **and**

    $phase \in \{fully/partlyClosed, fully/partlyOpen\}$

      then $\textsc{Notify}(StopStop)$

  if $cmd = Lower$ **then**

    if $phase = fullyClosed$ **then** $\textsc{Notify}(LowerAtBottom)$

    if $phase = rising$ **then** $\textsc{Notify}(RaiseLower)$

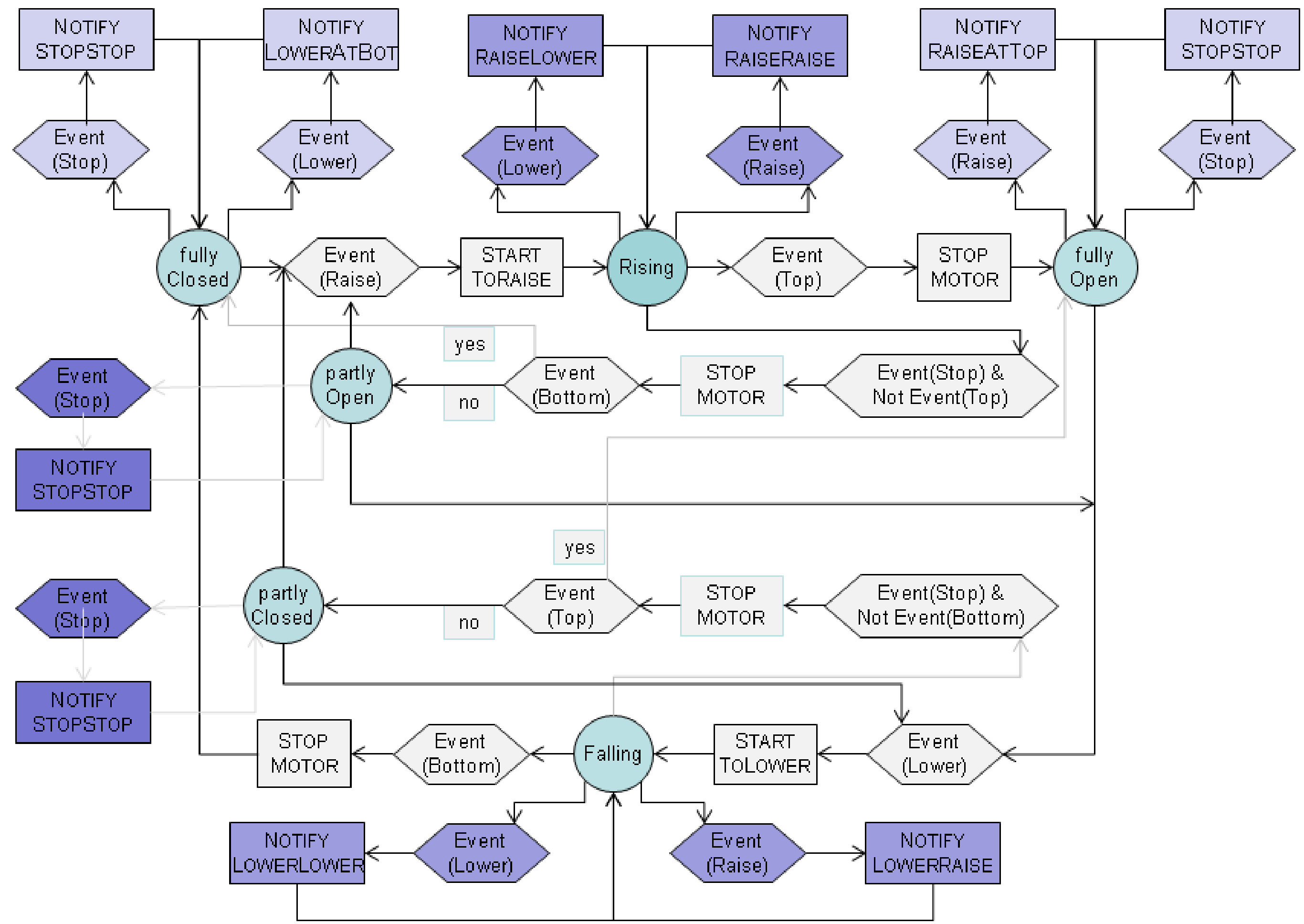
    if $phase = falling$ **then** $\textsc{Notify}(LowerLower)$

  if $cmd = Raise$ **then**

    if $phase = fullyOpen$ **then** $\textsc{Notify}(RaiseAtTop)$

    if $phase = rising$ **then** $\textsc{Notify}(RaiseRaise)$

    if $phase = falling$ **then** $\textsc{Notify}(LowerRaise)$


NOTIFY STOPSTOP
NOTIFY LOWERATBOT
NOTIFY RAISELOWER
NOTIFY RAISERAISE
NOTIFY RAISEATTOP
NOTIFY STOPSTOP
Event (Stop)
Event (Lower)
Event (Lower)
Event (Raise)
Event (Raise)
Event (Stop)
fully Closed
Event (Raise)
START TORAISE
Rising
Event (Top)
STOP MOTOR
fully Open
yes
partly Open
no
Event (Bottom)
STOP MOTOR
Event(Stop) & Not Event(Top)
Event (Stop)
NOTIFY STOPSTOP
yes
Event (Stop)
partly Closed
no
Event (Top)
STOP MOTOR
Event(Stop) & Not Event(Bottom)
NOTIFY STOPSTOP
STOP MOTOR
Event (Bottom)
Falling
START TOLOWER
Event (Lower)
NOTIFY LOWERLOWER
Event (Lower)
Event (Raise)
NOTIFY LOWERRAISE

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
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de

Copyright Notice