Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling a Package Router

Illustrating the Synchronous Parallelism of ASMs

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

See Ch.2.3 of Modeling Companion[1]

[1] Figures are © 2018 Springer-Verlag Germany, reused with permission.

Synchronous parallelism of mono-agent ASMs

Synchronous parallelism of ASM executions supported by:

- **par** to express bounded parallelism
- **forall** to express potentially unbounded parallelism

In the PackageRouter example we

- introduce these two constructs and
- illustrate how application-domain-driven decisions steer the formulation of the ground model

Package Router requirements (1)

PlantReq. A package router sorts packages according to their destination into *destination bins*. Packages arriving in the *entry station* carry code indicating their *destination* which they can reach sliding down a path formed by two-position *switches* equipped with entry and exit *sensors* and connected by *pipes*

FunctionalReq. The controller reads the destination code and steers for each package its path to the destination bin by appropriately positioning the switches on this path.

EntryStationReq. The entry station elaborates one entering package per round. First it reads the package code, then it lets the package slide down while blocking the entry of other packages until the entered package has left the entry station.

Package Router requirements (2)

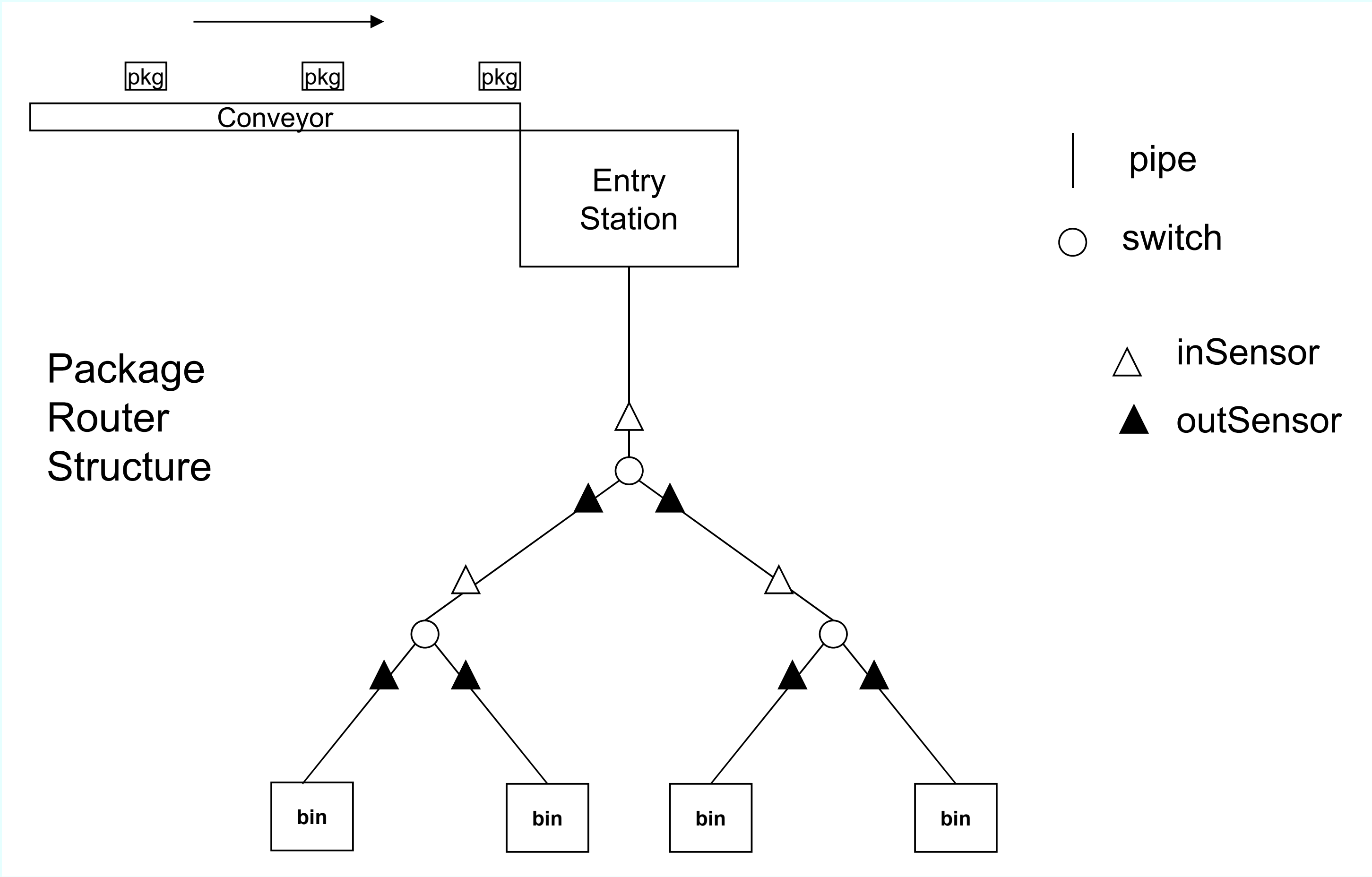
SensorReq. The sensors of switches are guaranteed to detect each package separately.

SwitchReq. A switch must be free when its position is flipped, i.e. there must be no package between the two sensors.

MisroutingReq. When a package arrives at the entry sensor of a switch that has to be flipped to correctly route this package before the preceding package has passed the exit sensor of the switch, then the switch is not flipped and the arriving package is misrouted. Since this can happen repeatedly, a misrouted package may be routed to any bin and an appropriate report should be issued.

PkgSlidingReq. Packages may slide down at different speeds so that more than one package may be in a pipe or switch. No package can overtake another, neither in a pipe nor in a switch.

Package Router Background structure



Package Router Signature: Static Part

- Static sets $Bin, Pipe, Switch$ with static *tree structure*
  - root $EntryStation$ with static successor functions:
  - $succ(EntryStation) \in Pipe$,
  - $succ(p) \in Switch \cup Bin$ for every $p \in Pipe$,
  - $succ_{left}(sw), succ_{right}(sw) \in Pipe$ for every $sw \in Switch$.
  - We use the inverse predecessor function $predecessor(b) = a$ for $succ(a) = b$.
- $bin : Destination \rightarrow Bin$ associating a destination with a bin.
- $dirToDest : Switch \times Bin \rightarrow \{left, right, none\}$
  - indicating the position where to direct sw to enter a path from sw to bin (if there is some)

Package Router Signature: Dynamic Part (1)

- Dynamic functions belonging to switches:
  - monitored $inSensor$, $outSensor_{left}$, $outSensor_{right}$ with values in $\{high, low\}$, written also with parameters $inSensor(sw)$ or $inSensor_{sw}$, etc.
  - a controlled variable *pos* with values in $\{right, left\}$ indicating the current position of the switch, also written $pos(sw)$ or pos_{sw}
- Monitored predicates of the entry station:
  - $PkgArrival$ signalling to the reader component the presence of a package that can be read
  - $PkgLeftEntry$ signalling that the package has left the entry station

These predicates represent sensor events; the link of a real-world phenomenon to its sensor predicate is typically expressed by a *Sensor Assumption* like the *PkgArrival/ExitAssumption* below.

Package Router Signature: Dynamic Part (2)

- controlled var $pkgId$ updated by reader of entry station to values in $PkgId$, internally representing an entered package.
- function $dest(id)$ controlled by the entry station to record the value it reads from the package code associated with id, provided by an external reader function $pkgDest$ which extracts from the currently read package code $currPkgCode$ the destination of the encoded package. $currPkgCode$ is a monitored variable.

The *PkgSlidingReq* together with *SwitchReq* and *MisroutingReq* indicate that pipes and switches may contain a sequence of packages, represented in the model by a FIFO $queue$ structure:

- $queue_{pipe}$ representing the packages in the $pipe \in Pipe$,
- for each $sw \in Switch$ a $queue_{sw}$ to contain $pkgid$s which
  - entered into switch at its entry point $inSensor$,
  - did not yet exit the switch at an exit point $outSensor_{left/right}$

PackageRouter **behavior**

In the requirements a monoprocessor solution is asked for. The three component types are defined separately below.

- Pipes are inactive components (without own behavior), representing only queues which record the current position of packages.

PackageRouter =

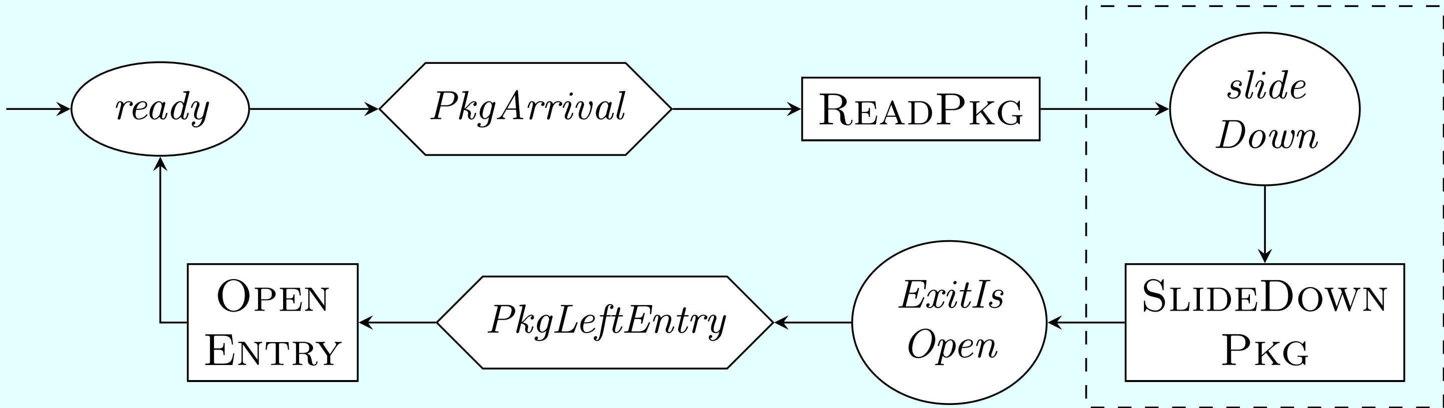
  EntryEngine

  forall $sw \in Switch$ Switch(sw)

  forall $bin \in Bin$ EnterPkgInto(bin)

NB. Parameter sw used here to denote an instance of Switch; each instance comes with its own dynamic state functions. This is a form of machine call, similarly for EnterPkgInto(bin).

Sequential EntryEngine (see **EntryStationReq**)



READPKG =

let $id =$ **new** $(PkgId)$

$pkgId := id$

$dest(id) := pkgDest(currPkgCode)$ -- extract *dest* from barcode

SLIDEDOWNPKG =

ENQUEUE$(pkgId, queue(succ(EntryStation)))$

OPENEXIT

NB. The dotted component will be refined below

EntryEngine components and assumptions

Submachines OpenExit and OpenEntry are left abstract, assuming appropriate initialization conditions and:

- *SingleEntryAssumption*. OpenExit, once the destination of the currently examined package has been decoded, opens the entry station exit to let this package slide down by gravity while blocking the entry of the next package. OpenEntry reopens the entry station to read a next package, once the just examined package has left the entry station entering the successor pipe.
- *PkgArrival/ExitAssumption*. When a package arrives at the reader component of the entry station, the predicate $PkgArrival$ becomes true. It switches back to false when OpenExit opens the entry station to let the package slide down. When a package has left the entry station, the predicate $PkgLeftEntry$ becomes true and switches back to false when OpenEntry reopens the entry station to let the next package enter the reader component.

SWITCH component

FunctionalReq and *SwitchReq* imply that switch control performs two actions:

- a SWITCHENTRY to update the switch position, if needed to correctly route an incoming package
- a SWITCHEXIT when a package slides down to the successor pipe

Since these two actions are independent of each other

- one operating at the head and one at the tail of the switch queue

there is no need to sequentialize them so that we put them in parallel.

$\textsc{Switch}(sw) =$

$\quad \textsc{SwitchEntry}(sw)$

$\quad \textsc{SwitchExit}(sw)$

SwitchEntry

A switch has a sequential character:

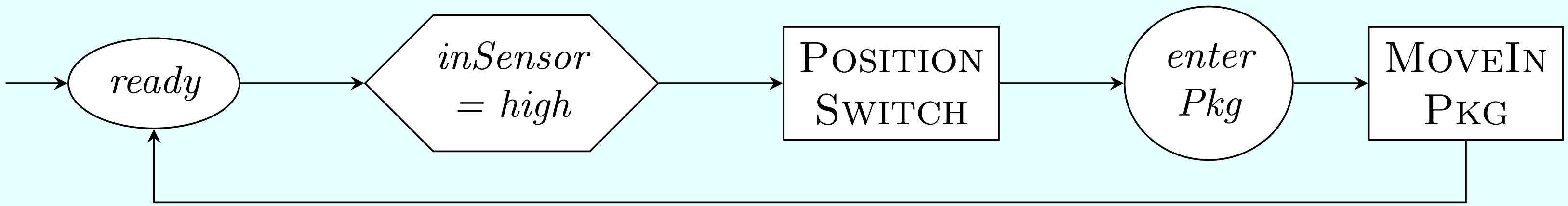
- upon the arrival of a $p$ackage (detected by $inSensor_{sw}$) and *before* letting it enter (by MoveInPkg)

it must first try to update its position (if needed)

- to the direction $dirToDest(sw, destBin)$

to correctly route the $p$ackage from this switch to its destination bin $destBin = bin(dest(p))$

- unless the package is already misrouted, case in which $dirToDest(sw, destBin) = none$



$\textsc{PositionSwitch}$ component

$\textsc{PositionSwitch} =$

let $pipe = predecessor(\mathbf{self})$ -- the pipe before the switch

let $pkg = head(queue(pipe))$ -- the package arriving from $pipe$

if $NeededToSwitch(pos, pkg)$ **and** $Free(\mathbf{self})$ **then** $\textsc{Flip}(pos)$

$NeededToSwitch(pos, p)$ iff $pos \neq dirToDest(\mathbf{self}, bin(dest(p)))$

-- current switch position would lead to misrouting

$Free(sw)$ iff $queue(sw) = [\,]$ -- there is no pkg in the switch

$\textsc{Flip}(pos) = (pos := pos')$

$pos' =$ the opposite value of pos

Package moving components

$\textsc{MoveInPkg} =$

let $pipe = predecessor(\textbf{self})$

let $pkg = head(queue(pipe))$ **in**

$\textsc{Dequeue}(queue(pipe))$ -- move package out of $pipe$

$\textsc{Enqueue}(pkg, queue(\textbf{self}))$ -- move package into switch

$\textsc{SwitchExit}$ moves a package from the switch into its successor pipe:

$\textsc{SwitchExit} =$

if $outSensor_{left} = high$ **or** $outSensor_{right} = high$ **then**

let $pkg = head(queue(\textbf{self}))$ -- $\textsc{MoveOutPkg}$

$\textsc{Enqueue}(pkg, queue(succ(\textbf{self})))$

$\textsc{Dequeue}(queue(\textbf{self}))$

$\textsc{EnterPkgInto}(bin)$

NB. *MisroutingReq* completed by stipulating that misrouting is reported upon pkg arrival in a bin

$\textsc{EnterPkgInto}(bin) =$

let $pipe = predecessor(bin)$

let $pkg = head(pipe)$

$\textsc{Dequeue}(queue(pipe))$

$\textsc{Insert}(pkg, bin)$

$\textsc{ReportMisrouting}(pkg, bin)$

where

$\textsc{ReportMisrouting}(p, b) =$

if $bin(dest(p)) \neq b$ **then**

$\textsc{Display}($'p has been misrouted to b instead of $dest(p)'$$)$

Throughput concern: a refinement

Problem: from the way the PackageRouterRequirements are formulated they would still be satisfied even if every package is misrouted!

Throughput concern: 'most packages will be routed correctly'

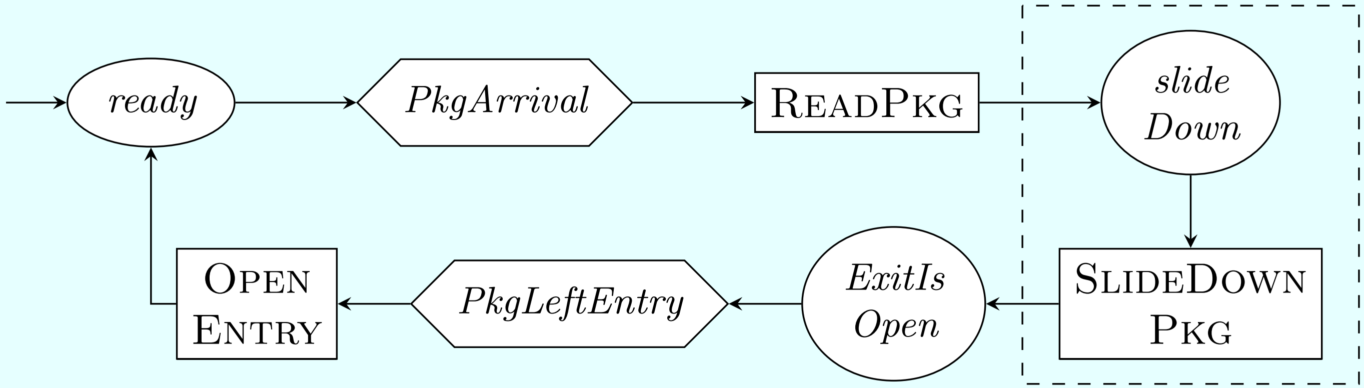
- is an issue of how often it happens that in a switch packages meet which need opposite switch positions to be routed correctly
- by *PkgSlidingReq* it is an issue of how fast packages slide down

Software alone cannot solve the problem: domain knowledge needed.

Additional EntryStationDelayRequirement

The entry station checks whether the destination of the currently entering package is the same as for the previously entered one. If this is not the case, then *sliding down* the currently entering package must be *delayed* such that the switches the package has to pass can be flipped in time where needed for a correct routing.

Component structure permits to insert the new requirement into the two affected components ReadPkg and SlideDownPkg

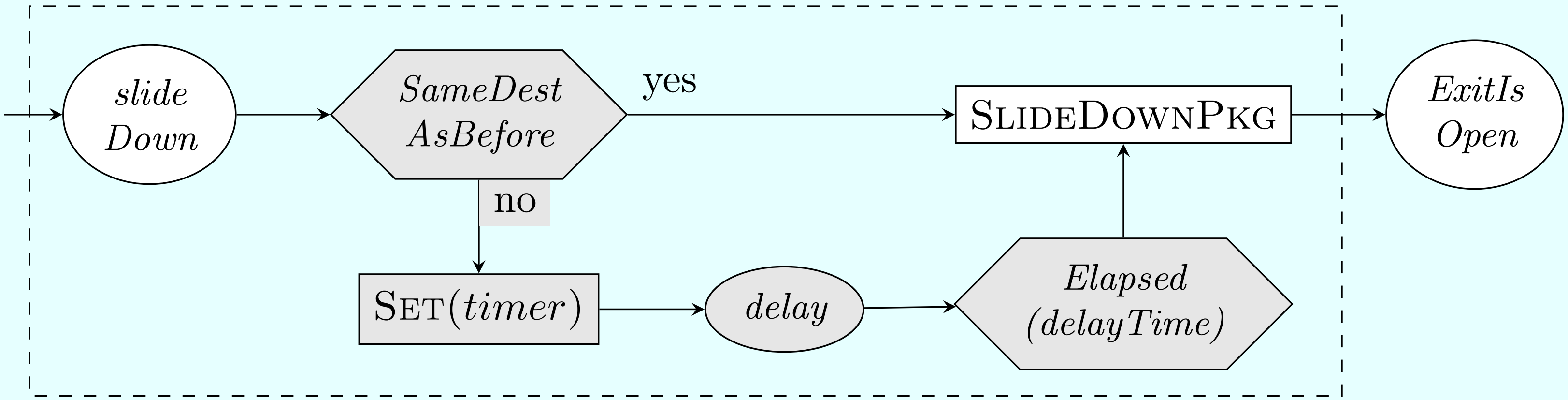


DelayedSlideDown **component of** EntryEngine

DelayRefinedReadPkg =

previousPkgDest := $dest(pkgId)$ -- record the previous destination

ReadPkg



$SameDestAsBefore$ iff $dest(pkgId) = previousPkgDest$

$Elapsed(delayTime)$ iff $now - timer \geq delayTime$

References (1)

The case study requirements document:

- G. Hommel: Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage. Kernforschungszentrum Karlsruhe, TR, August 1980.

Some publications which investigate the case study:

- R. M. Balzer and N. M. Goldman and D. S. Wile: Operational specification as the basis for rapid prototyping. ACM SIGSOFT Software Engineering Notes 7 (5), 3-16 (1982)
- W. Wartout and R. Balzer: On the inevitable intertwining of specification and implementation. Comm. ACM 25(7) 438-440 (1982)
- P. E. London and M. S. Feather: Implementing specification freedoms. In: C. Rich and C. E. Waters (eds): Readings in AI and Software Engineering, Morgan Kaufmann 1986, 285-305

References (2)

- D. Jackson and M. Jackson: Problem decomposition for reuse. Software Engineering J. 11 (1), 19-30, 1996
- M. S. Feather and S. Fickas and A. Finkelstein and A. van Lamsweerde: Requirements and specification exemplars. Automated Software Engineering 4 (4) 419-438, 1997
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018 `http://modelingbook.informatik.uni-ulm.de`

Copyright Notice