Problem Decomposition for Reuse

Daniel Jackson
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Michael Jackson
MAJ Consulting Ltd
101 Hamilton Terrace
London NW8 9QX

January 2, 1995

Abstract

An approach to software development problems is presented, and illustrated by an example. The approach is based on the ideas of *problem frames* and structuring specifications by *views*. It is claimed that decompositions obtained by this approach result in a more effective separation of concerns, and that the resulting components are more likely to be reusable than those obtained by more conventional approaches. The characteristics of desirable integration mechanisms are discussed, together with some other considerations arising out of the approach presented.

1 Introduction

Problem decomposition serves two purposes. By decomposing a large problem into smaller subproblems we hope to master its complexity: the smaller subproblems should be simpler than the large problem. By the same decomposition we hope to factor out subproblems that are already solved, and to re-use their existing solutions.

Hierarchical decomposition can rarely achieve these goals. It is not difficult to sketch a plausible hierarchy of procedures, but very difficult indeed to be sure that each successive level of decomposition is making the task easier and not harder: a smaller problem is not necessarily a simpler problem. The constraints imposed by the difficulty of finding a workable hierarchical decomposition leave too little freedom for recognising and exploiting opportunities for re-use.

This is not surprising. Hierarchical structure is extremely specialised. Few problem domains, and even fewer problems, exhibit hierarchical structure. Instead, both problems and problem domains usually exhibit parallel structure. When a problem with an essentially parallel structure is forced into a hierarchical decomposition, the resulting components are likely to be unsatisfactory in both the problem and the solution domain.

The problem domain, for example, may be decomposed into domain entities, viewed as *agents* [8, 2]. The component for each agent will then inevitably entangle different aspects of its behaviour that are unlikely to reappear in exactly that combination in any other problem. In the solution domain the same difficulty is found in a different context. Each hierarchically derived module must serve several computational purposes simultaneously: the particular combination of purposes is unlikely to be useful elsewhere. Essentially, this is why libraries of reusable modules have proved so disappointing, except in applications such as numerical computation and window interface implementation, where the problem domains and problems are already highly standardised.

It is parallel structure, and the benefits of a decomposition that respects its nature, that characterises the present approach and other approaches based on the notion of viewpoints [10, 9]. The approach presented in this paper combines two ingredients, both of which aim to exploit parallel structuring of problems and of problem domains. They are: the idea of problem frames [6, 7]; and the idea of structuring Z specifications as views [4].

A problem frame is a template characterising a class of simple problems—that is, problems for which a reliable solution method is known. It also characterises a class of problem whose solutions are likely to be reusable. In general, different problem frames invite the use of different specification languages—sometimes more than one for a frame. However, it is not our purpose here to discuss multiparadigm specification [12], which is largely orthogonal to the immediate concerns of this paper (although not orthogonal to problem decomposition generally). We have therefore chosen to use Z for all the problem frames: partly because this avoids some still unsolved difficulties of integrating specifications written in different languages, and partly because the syntactic mechanisms of Z offer a number of features that are convenient for conjoining parallel specifications.

The approach is illustrated by an example problem: the control of a package routing machine [11, 1]. We show the decomposition of the problem into three subproblems, or views, that fit into three problem frames; the specification of each view; and the connection of the subproblem views into a single specification. In a section at the end of the paper we discuss some general aspects of our approach. We also discuss some difficulties of view integration as they appear in Z and other languages, and outline some desirable properties of an integration mechanism.

For convenience, the complete specification is gathered together at the end of the paper, in an Appendix.

2 Problem Frames

A problem frame [6, 7] is a structure of *principal parts* and a *solution task*. Each *principal part* is one of the following:

- The machine to be constructed—that is, to be described by the software being developed.
- A part of the environment or domain of the problem.
- A required relationship among parts of the environment or domain.

The *solution task* is always to construct the machine so that it brings about or maintains the required relationships.

Particular problem frames are specialised to particular problem classes. A particular problem frame can be thought of as a template to be fitted over a problem: the principal parts of the frame are identified with parts and aspects of the problem's system, environment, and requirements. When a frame is fitted to a problem in this way, it is said to be *applied* to the problem. It then guides the choice and use of an appropriate method for solving the problem. The purpose of identifying and studying particular problem frames is to build up a repertoire of problem classes, each having at least one well-understood and reliable solution method.

Problem frames must be simple if they are to be useful. But realistic problems are rarely simple. The complexity of a realistic problem can be regarded as a parallel composition of two or more problem frames. Problem decomposition is then the recognition of the appropriate problem frames and the identification of their principal parts. A problem frame applied in this way has something in common with an instantiated Viewpoint [9], which incorporates a representation scheme, a development process, and an area of concern in the problem domain.

In different problem frames the principal parts have different characteristics and are differently connected. Many problem frames are needed, to accommodate the many parts and aspects of realistic problems. Here we outline only three particular problem frames, chosen because they are the frames we will use in our example problem.

2.1 Control Frame

The *Control* problem frame might be used for a problem such as the control of a simple level crossing in a railway. The frame has three principal parts:

- The *Controller*, which is the machine to be built.
- The *Controlled Domain*, which is a dynamic part of the problem environment. The *Controlled Domain* is connected directly to the *Controller*: for example, sensing the arrival of a train, and switching on the motor that raises and lowers the barrier are events both in the machine and in the domain. The *Controlled Domain* is partly autonomous—the arrival of the train requires no external stimulus. It is also partly reactive—when the motor is switched on the barrier will rise or fall. It is

the reactive property of the *Controlled Domain* that makes it amenable to control by the *Controller*.

- The *Required Behaviour* is a constraint on the events and states of the *Controlled Domain*, which the *Controller* is required to maintain: for example, that the barrier should be down whenever a train is in the vicinity of the crossing.

To solve a problem fitting the *Control* frame it is necessary to consider the autonomous events of the *Controlled Domain* and determine the appropriate response of the *Controller* to bring about or maintain the *Required Behaviour*. It may be necessary to implement a model of the *Controlled Domain* inside the *System*, to allow appropriate responses to be determined at execution time.

2.2 Static Information Frame

The *Static Information* problem frame might be used for a simple Bible concordance system. It has five parts:

- The *System*, which is the machine to be built.

- The *Subject Domain*, which is a static part of the world, not connected to the *System*. Because it is static, it has a fixed state and no events: the text of the Bible is not subject to change.

- The *Association Events*, which form an unstructured stream of events in which associations are defined between the phenomena of the *Subject Domain* and the phenomena of the *System*. These events are not events in the *Subject Domain* of the Bible text. They are events in a set-up procedure by which the *System* acquires an internal model of the Bible text.

- The *Information Requests*, which are an unstructured set of questions and associated responses about the *Subject Domain*. For example, a question might be "Where do the names *Cain* and *Abel* occur in the same verse?". In the implementation of the *System* a question and its response might be a function invocation and result; or the reading of an input message and writing of an output reply; or simply the accessing of an internal data structure of the *System* by another application program.

- The *Information Rules*, which are required relationships between the *Subject Domain* and the *Information Requests*. For example, the meaning of 'in the same verse' is to be interpreted in a certain way in terms of the text of the Bible.

Solving a *Static Information* problem typically requires that the *System* should have an internal model of the *Subject Domain*. The model may be implemented by an internal data structure, by a database, and in many other ways. The *Association Events* initialise this model; subsequently the *System* uses the model but does not update it.

2.3 Dynamic Information Frame

The *Dynamic Information* problem frame might be used for monitoring the traffic using a segment of road. The frame has four principal parts:

- The *System*, which is the machine to be built.

- The *Subject Domain*, which is a dynamic part of the world, directly connected to the *System*: sensors laid in the roadway ensure that the passage of a wheel over a certain line across the road is an event both in the *System* and in the *Subject Domain*. The *Subject Domain* is entirely autonomous, its dynamic behaviour being unaffected by the *System*. The behaviour of interest in the *Subject Domain* would be the passage of time, and the passage of vehicles over the road segment being monitored.

- The *Information Requests*, which are again an unstructured set of questions and associated responses about the *Subject Domain*, and have a similarly wide range of possible implementations. A request might concern the rate of increase of traffic in the road segment in a particular time period.

- The *Information Rules*, which are required relationships between the behaviour of the *Subject Domain* and the *Information Requests*. For example, certain patterns of wheel pulses are to be interpreted as the passage of a three-axle vehicle.

A problem fitting the *Dynamic Information* frame requires the *System* to collect and maintain information from the autonomous events in the *Subject Domain*. This information is needed for responding to the *Information Requests*. It may take the form of an elaborate model of the changing state of the *Subject Domain*, or may be little more than a record of selected past events.

In the following section we describe our example problem and decompose it into three subproblems that fit the frames given above.

3 An Example Problem

Our example is adapted from a well-known problem discussed by Swartout and Balzer [11, 1]. It concerns the control of a package router.

3.1 Problem Statement

A package router consists of a binary tree of pipes through which packages slide by gravity to destination bins, passing through two-position switches that can be set to direct them to the right bins. A reading station detects the bar-coded destination of each package on entry to the machine. The system must control the router by setting the switch positions appropriately.

Each pipe has sensors at its top and bottom, which close when the leading edge of a package arrives and open when its trailing edge leaves. The pipes are bent in the vicinity of the sensors, so that the sensors are guaranteed to detect each passing package, no matter how closely the packages follow one another. Package size is restricted so that no overtaking is possible, either in the pipes or in the switches.

A switch may be reset only when no package is present between its incoming and outgoing pipes. Because packages slide at unpredictable rates, a package may follow another too closely for correct setting of a switch. A wrongly routed package may be routed to any bin: a message is displayed showing the intended and actual destinations.

3.2 Problem Frames for the Package Router

At first sight the whole problem appears to fit into the Control problem frame. The router equipment, with the packages, forms the *Controlled Domain*; the machine we are to build is the *Controller*. As required by the problem frame, the machine is directly connected to the router: it can detect the current settings of the switches and any change of state in a sensor; it can detect the reading of a bar-coded destination and the value read; and it can flip a switch or display a message by executing appropriate procedures. The *Required Behaviour* is that each package should either find its way to its proper destination or have its misrouting appropriately notified by a message.

However, the problem is a little more complex than this. The connections provide no information about the configuration of the pipes, sensors, switches and bins. The machine can detect a change in a sensor state, but it has no way of knowing where that sensor is in the tree of router pipes. It can flip a switch, but it has no way of knowing where that switch is in relation to the sensors and bins. This information is essential for controlling the router, and must come from some kind of set-up procedure, which creates an internal model of the topology of the router. In addition to the *Control* frame we therefore need a *Static Information* frame.

There is a further complexity. The destination of each package is read just once, from its bar-coded label, by the reading station. When the package is detected by each sensor in its subsequent journey through the router, there is nothing to associate the package directly with its destination: the label is not read again at each sensor. It is therefore necessary to deduce the package destination from what is known about the topology of the router and the way the packages flow through it. This problem fits into the *Dynamic Information* problem frame.

The problem of controlling the router may therefore be decomposed into three subproblem views. One subproblem view fits the *Control* frame: we will call this the Switch Control view. One fits the *Dynamic Information* frame: we will call this the Package Tracking view. The third fits the *Static Information* frame: we will call this the Router Topology view.

3.3 Adequacy of a Problem Frame

None of the three problem frames we have described is alone enough to solve the whole problem. The *Static Information* frame can not possibly deal with the dynamic aspects of the problem. The *Dynamic Information* frame lacks an effective set-up procedure. It also assumes that its *Subject Domain* is entirely autonomous and unaffected by the *System*, and therefore lacks provision for describing a *Required Behaviour* of the *Subject Domain* and any actions by which the *System* might bring it about.

Of course, the limitations of a problem frame can be ignored, and the problem forced, in the fashion of Procrustes, into an unsuitable mould. This is often done. For example, JSD, which is essentially a method for *Dynamic Information* problems, has been pressed into service to solve problems that would fit far better into the *Control* frame [5]. We are accustomed to this kind of fudging, and expect to devote some effort to finding work-arounds. But this cavalier unconcern for the nature and structure of a problem is not a recipe for successful development.

4 Router Topology View

The Router Topology view fits the *Static Information* problem frame. The principal parts are these:

- The *System* is, of course, the machine we are developing.
- The *Subject Domain* is the static configuration of the router equipment. The current settings of the switches and the current locations of the packages are not a part of the *Subject Domain* in this view.
- The *Association Events* are the events of a set-up procedure in which the *System* receives the information it needs about the router: which sensors are attached to which pipes; which pipes enter and leave each switch; and so on. This set-up procedure would be performed when the physical components of the router are initially configured, and whenever the configuration is changed.
- The *Information Requests* are the interactions between this subproblem view and the other subproblem views, in which it provides information

about the router's static configuration: for example, the position of a particular sensor in relation to its pipe, and whether a particular bin can be reached from a particular switch.

- The *Information Rules* are the relationships between the *Subject Domain* and the *Information Requests* that assure correctness of the answers. For example, the router is a directed tree in which the leaves are reachable from the root but the root is not reachable from the leaves.

For reasons of brevity, we will omit any account of the *Association Events* and the set-up procedure. The set-up procedure produces a model of the *Subject Domain* inside the *System*. That is, it produces an arrangement of machine phenomena—data structures and values of variables— that is isomorphic to the *Subject Domain*. The *System* then uses this model to answer the questions posed in the *Information Requests*.

We will assume that the router equipment has been assembled; that the set-up procedure has been performed; and that the model has been constructed. We restrict ourselves to describing the *Subject Domain* and the *Information Requests* and *Information Rules*.

The *RouterDomain* schema below is a description both of the model inside the *System* and of the *Subject Domain*—the router—itself. The individual parts of the router are its pipes, switches, bins, and sensors. These parts are related in certain ways. Each sensor is located on one pipe, at either its upwards or its downwards end. Each switch has a pipe entering it, and a left and a right pipe leaving it. There is a *top* pipe, into which packages flow on leaving the reader. It enters a switch. Each pipe either *enters* a switch or terminates at (*fills*) a bin. The whole configuration of pipes forms a tree whose root is the *top* pipe and whose leaves are pipes terminating at bins.

(Notice that we have chosen to treat the *reader* as if it were a sensor, although it is not attached to any pipe; this distortion of reality is discussed later in the paper.)

$$[SWITCH, PIPE, BIN, SENSOR]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RouterDomain \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
top : PIPE \\
reader : SENSOR \\
sensUp, sensDn : PIPE \rightarrowtail SENSOR \\
enters : PIPE \rightarrowtail\!\!\!\!\!+ \; SWITCH \\
pipeL, pipeR : SWITCH \rightarrowtail PIPE \\
fills : PIPE \rightarrowtail\!\!\!\!\!+ \; BIN \\
\hline
reader \notin \operatorname{ran}(sensUp \cup sensDn) \\
top \in \operatorname{dom} enters \\
\operatorname{dom} enters \cap \operatorname{dom} fills = \varnothing \\
\operatorname{ran} sensUp \cap \operatorname{ran} sensDn = \varnothing \\
\operatorname{ran} pipeL \cap \operatorname{ran} pipeR = \varnothing \\
\operatorname{ran} pipeL \cup \operatorname{ran} pipeR = (\operatorname{dom} enters \setminus \{top\}) \cup \operatorname{dom} fills \\
\hline
\end{array}
$$

The *Information Requests* will come from two sources:

1. The Switch Control view must control the flow of packages by setting the switches. It therefore needs to determine:

   - for each sensor, whether the sensor *guards* the entrance to a switch (so that a package that has just passed that sensor is entering the switch) and, if so, which one; and
   - for each switch and each destination bin, whether the *way* to the bin lies through the left or right pipe of the switch.

2. The Package Tracking view needs enough information to keep track of each package in its journey through the router. Since the Router Topology view is concerned only with the static configuration of the router, it can provide no information about current switch settings. The information it provides is the form of a *precedes* relation: given a sensor S (other than the reader), it identifies the sensor (possibly the reader) that precedes S in the tree.

We specify the *Information Requests* and the *Information Rules* together in two schemas. The first establishes some useful definitions; the second specifies the *guards*, *way* and *precedes* relations.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{RouterStaticInformation}} \\
\textit{RouterDomain} \\
\textit{flow} : \textit{PIPE} \leftrightarrow \textit{PIPE} \\
\textit{reachL}, \textit{reachR} : \textit{SWITCH} \leftrightarrow \textit{BIN} \\
\hline
\textit{flow} = \textit{enters} \mathbin{⨾} (\textit{pipeL} \cup \textit{pipeR}) \\
\textit{reachL} = \textit{pipeL} \mathbin{⨾} \textit{flow}^* \mathbin{⨾} \textit{fills} \\
\textit{reachR} = \textit{pipeR} \mathbin{⨾} \textit{flow}^* \mathbin{⨾} \textit{fills} \\
\hline
\end{array}
$$

$DEST == BIN$
$DIR ::= L \mid R$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{RouterRequestsAndRules}} \\
\textit{RouterStaticInformation} \\
\textit{guards} : \textit{SENSOR} \rightarrowtail\!\!\!\!\rightarrow \textit{SWITCH} \\
\textit{way} : (\textit{SWITCH} \times \textit{DEST}) \leftrightarrow \textit{DIR} \\
\textit{precedes} : \textit{SENSOR} \nrightarrow \textit{SENSOR} \\
\hline
\textit{guards} = \textit{sensDn}^{\sim} \mathbin{⨾} \textit{enters} \\
\textit{way} = (\textit{reachL} \times \{L\}) \cup (\textit{reachR} \times \{R\}) \\
\textit{precedes} = \{\textit{reader} \mapsto \textit{sensUp}(\textit{top})\} \cup \\
\ \{p : \textit{PIPE} \bullet \textit{sensUp}(p) \mapsto \textit{sensDn}(p)\} \cup \\
\ \{(p, p') \in \textit{flow} \bullet \textit{sensDn}(p) \mapsto \textit{sensUp}(p')\} \\
\hline
\end{array}
$$

5 Package Tracking View

The Package Tracking view fits the *Dynamic Information* problem frame. The principal parts are these:

- The *System* is, of course, the machine we are developing.
- The *Subject Domain* is a dynamic view of the router and the packages flowing through it. The relevant events in the router—essentially, the reading of a package destination or the detection of a package by a sensor—are directly connected to the *System*: that is, each event is both

an event in the router and an event in the machine that is tracking the packages. The effect of each relevant event in the machine is to update an internal model of the package locations; this model is used to satisfy the *Information Requests*.

- The *Information Requests* provide information about the dynamic behaviour of the router and packages.

- The *Information Rules* are the relationships between the *Subject Domain* and the *Information Requests* that assure correctness of the information given. For example, because packages do not overtake one another in pipes or switches, the packages in one pipe or one switch form a FIFO queue.

The information to be provided is essentially an interpretation of each event in the journey of each package. When a package passes a sensor, the Switch Control view may need to determine:

1. What is the destination of the package? and

2. If the package has reached a switch, is the switch empty?

The internal model maintained by the Package Tracking view consists of queues of packages (represented only by their destinations). One queue is associated with each sensor, excluding those sensors that lead directly into bins. Once a package has passed into a bin it will be of no further interest.

This model is sufficient to answer the questions above. It does not require information about the settings of switches in the router, since it relies only on determining which queue a package is leaving when it arrives at the next sensor. It does, of course, depend on information about the router configuration, obtained from the Router Topology view. In the Package Tracking view, this information appears in the schema below as declarations of the variables *precedes*, *bin* and *reader*. The values of these variables are unconstrained in this view, and will be determined when the view is connected to the Router Topology view.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, PackageTrackingDomain \,\rule{5cm}{0.4pt}} \\
queue : SENSOR \rightarrowtail\!\!\!\!\!\!+ \ \mathrm{seq}\, DEST \\
precedes : SENSOR \leftrightarrow SENSOR \\
bin : SENSOR \leftrightarrow BIN \\
reader : SENSOR \\
\hline
\end{array}
$$

The *queue* model must be initialised. A *Static Information* model is initialised by the *Association Events* in its set-up procedure. For a *Dynamic Information* model there may in some cases be an appropriate initial event in the *Subject Domain*, but this is unusual. More often, as here, it is necessary to define, and have the *System* execute, a special initialisation event. The model is initialised by setting all its queues to the empty sequence:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Init\_PackageTrackingDomain \,\rule{5cm}{0.4pt}} \\
PackageTrackingDomain' \\
\hline
queue' = (\mathrm{dom}\, bin) \lhd (SENSOR \times \langle\rangle) \\
\hline
\end{array}
$$

The *queue* model is maintained by updating on each of the following events:

- *Transfer*, in which a package passes from the upper to the lower sensor of the same pipe, or from the entry pipe of a switch to one of its exit pipes;
- *ReadDest*, in which the package passes the *reader* and its destination is read; and
- *Deposit*, in which a package arrives at a bin.

In the schemas describing these events we need not constrain the relations *precedes*, *bin* and *reader* to remain unchanged by the events. They will be constrained appropriately when the views are connected together.

A *Transfer* event is one in which a package passes a sensor that is not the *reader* and does not lead to a bin. A *Transfer* event causes the model to be updated by moving the transferred package from the head of the queue it is leaving to the end of the queue it is joining. The value of the destination d in a *Transfer* event is determined from the *queue* model: the destination is always that of the package at the head of the queue associated with the preceding sensor, because that is the package being transferred.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transfer}} \\
\hline
\Delta \mathit{PackageTrackingDomain} \\
se : \mathit{SENSOR} \\
d : \mathit{DEST} \\
\hline
se \neq reader \land se \notin \operatorname{dom} bin \\
d = \mathsf{head}\ queue(precedes(se)) \\
queue' = queue \oplus \\
\quad \{precedes(se) \mapsto \mathsf{tail}\ queue(precedes(se)), \\
\quad\ se \mapsto queue(se) \frown \langle d \rangle\} \\
\hline
\end{array}
$$

A *ReadDest* event is one in which a package passes the sensor that is the *reader*. A *ReadDest* event causes the model to be updated by adding the package just read (represented by its destination) to the end of the queue associated with the *reader*. The value of the destination d in a *ReadDest* event is determined from the state shared between the router and the *Controller*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadDest}} \\
\hline
\Delta \mathit{PackageTrackingDomain} \\
se : \mathit{SENSOR} \\
d : \mathit{DEST} \\
\hline
se = reader \\
queue' = queue \oplus \{reader \mapsto queue(reader) \frown \langle d \rangle\} \\
\hline
\end{array}
$$

A *Deposit* event is one in which a package passes a sensor that leads to a bin, into which the package is then deposited. A *Deposit* event causes the model to be updated by removing the deposited package from the queue it has just left. The value of the destination d in a *Deposit* event is determined from the *queue* model. The value of the bin b reached is calculated from the *bin* relation.

$$
\begin{array}{|l}
\hline
\textit{Deposit} \\
\Delta \textit{PackageTrackingDomain} \\
se : \textit{SENSOR} \\
d : \textit{DEST} \\
b : \textit{BIN} \\
\hline
se \in \operatorname{dom} bin \wedge b = bin(se) \\
d = \mathsf{head}(queue(precedes(se))) \\
queue' = queue \oplus \{precedes(se) \mapsto \mathsf{tail}\ queue(precedes(se))\} \\
\hline
\end{array}
$$

A package is deposited in the right bin if the sensor it has just passed is at the entry to its destination bin. Otherwise it is deposited in a wrong bin, and a message must be displayed.

$$
\begin{array}{|l}
\hline
\textit{RightBin} \\
\textit{Deposit} \\
\hline
b = d \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{WrongBinMessage} \\
\textit{Deposit} \\
actual!, desired! : \textit{BIN} \\
\hline
b \neq d \\
desired! = d \\
actual! = b \\
\hline
\end{array}
$$

6 Switch Control View

The Switch Control view fits the *Control* frame. The principal parts are these:

- The *Controller* is, of course, the machine we are developing.
- The *Controlled Domain* is the router switches and the packages flowing through them. The autonomous aspect of the *Controlled Domain* is the package flow: packages pass sensors under the force of gravity, and the

Controller can neither stimulate nor inhibit these events. The reactive aspect of the *Controlled Domain* is that the *Controller* can set the router switches, and so determine the path taken by each package.

- The *Required Behaviour* is that each package should be directed along a path to the destination given in its bar-coded label, subject to the overriding rule that no switch may be reset while it contains any package.

The *Controller* is connected to the router by two classes of shared event and by a shared state. The shared state is the current *setting* of each switch. At any time, each switch is set either in the L or in the R direction, and this state is shared with the *Controller*, perhaps by a DMA connection. The shared state is simply:

SwitchSettings

$$setting : SWITCH \longrightarrow DIR$$

As in the two other views, the schema describing the model is a description both of the state of the *Controlled Domain* and of a part of the internal model to be maintained by the *Controller*.

The shared events are *ArriveAtSwitch* and *FlipSwitch*:

- *ArriveAtSwitch* events are autonomous events in the *Controlled Domain*, in which a package arrives at a switch.

- *FlipSwitch* events are initiated by the *Controller*. In a *FlipSwitch* event the direction of a switch is reversed (from L to R or from R to L).

For each *ArriveSwitch* event that occurs, the *Controller* must determine whether a *FlipSwitch* event should occur in response. To do so, it may need to determine:

- the current setting of the switch;

- the destination of the arriving package;

- the appropriate switch setting for the destination; and

- whether the switch is currently empty.

The current setting of the switch is directly available from the shared state *SwitchSettings*. In the schema below, the appropriate switch setting for a destination is declared as an unconstrained variable *choice*. In this frame we need not constrain *choice* in any way. We may leave open all the possibilities, namely that it is:

- a total function: there is exactly one path to a given bin from a given switch;
- a partial function: there is one path, or no path, to a given bin from a given switch; or, most generally,
- a relation: there is one path, no path, or more than one path to a given bin from a given switch.

By treating *choice* as a relation we leave all the possibilities open in this view. (When we come to connect this view with the Router Topology view we will see that *choice* is a partial function; but if paths through the router could merge, then *choice* would have to be an unrestricted relation.)

Another unconstrained variable in the schema is *empty*: this is the set of currently empty switches. The values of *choice* and *empty* will be determined when the views are connected together. Together with *SwitchSettings* they constitute the model of the *Controlled Domain* available to the *Controller*:

SwitchesModel
$$
\begin{array}{l}
SwitchSettings \\
choice : (SWITCH \times DEST) \leftrightarrow DIR \\
empty : \mathbb{P}\, SWITCH
\end{array}
$$

An *ArriveAtSwitch* event involves the switch where the arrival has occurred, and the destination of the arriving package. We distinguish two kinds of *ArriveAtSwitch* event, according to whether the *Controller* responds by flipping the switch. (Although the arrival and the response are distinct events in the real world, it is appropriate and convenient in Z to treat them as a single event. This distortion of reality is discussed later in the paper.)

The *Required Behaviour* demands that the switch direction should be changed only if the switch is empty, and only if the changed direction would offer a path where the current direction would not. There is no point in changing the direction of the switch for a package for which *choice* specifies no appropriate direction: the package is already irretrievably misrouted.

$$
\begin{array}{l}
\mathit{flipped} : \mathit{DIR} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{DIR} \\
\hline
\mathit{flipped}(R) = L \\
\mathit{flipped}(L) = R
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{ArriveAndFlip} \\
\Delta \mathit{SwitchModel} \\
sw : \mathit{SWITCH} \\
d : \mathit{DEST} \\
\hline
sw \in \mathit{empty} \\
\mathit{choice}(\!| sw, d |\!) = \{\mathit{flipped}(\mathit{setting}(sw))\} \\
\mathit{setting}' = \mathit{setting} \oplus \{sw \mapsto \mathit{flipped}(\mathit{setting}(sw))\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{ArriveNoFlip} \\
\Delta \mathit{SwitchModel} \\
sw : \mathit{SWITCH} \\
d : \mathit{DEST} \\
\hline
\neg \ \mathrm{pre}\ \mathit{ArriveAndFlip} \\
\mathit{setting}' = \mathit{setting} \\
\hline
\end{array}
$$

7 Connections

The fundamental connection technique for the components of a parallel decomposition is logical conjunction [12]. For a Z specification this means combining schemas by including one within another, or by the conjunction operation of the Z schema calculus. It is then possible to impose additional invariants to constrain the components of the combined schemas.

7.1 State Connections

The schema below, prefaced by the necessary declarations, includes the model parts of the three views of the Package Router Control problem, and specifies the invariants that must hold among them. These are:

- *precedes* in the *PackageTrackingDomain* has the same value as *precedes* in the *RouterRequestsAndRules*. This need not be specified explicitly, because the two relations have the same name. In Z schema inclusion this implies that they denote the same thing.

- *reader* in the *PackageTrackingDomain* has the same value—that is, it is the same SENSOR—as *reader* in the *RouterRequestsAndRules*. Again, this need not be specified explicitly, because the two variables have the same name.

- *bin* in the *PackageTrackingDomain* has the value determined by *sensDn* and *fills* in the *RouterRequestsAndRules*: a sensor leads to a bin if it is the lower sensor of the pipe that fills the bin.

- *choice* in the *SwitchesModel* has the same value as *way* in the *RouterRequestsAndRules*.

- *empty* in the *SwitchesModel* has the value determined by the *queue* model in the *PackageTrackingDomain*. A switch is empty if the *queue* associated with the sensor leading into it is empty. The association between switches and sensors is determined by the relation *guards* in the Router Topology view.

$[SWITCH, PIPE, BIN, SENSOR]$
$DEST == BIN$
$DIR ::= L \mid R$

PackageRouterControl

$$
\begin{array}{|l}
RouterRequestsAndRules \\
PackageTrackingDomain \\
SwitchesModel \\
\hline
bin = sensDn^{\sim} \mathbin{;} fills \\
choice = way \\
empty = guards \mathrel{(\!|} \operatorname{dom}(queue \rhd \langle\rangle) \mathrel{|\!)}
\end{array}
$$

7.2 Event Connections

The *Init_PackageTrackingDomain* event is to be executed by the machine before any other event occurs in the router operation. It is already fully specified in the Package Tracking view, and need only be promoted to an operation on the PackageRouterControl state.

The events involving the flow of packages through the router are all subclasses of a more general event class, in which a package passes a sensor. The physical connection between the router and the machine indicates which sensor has been passed, but other arguments of these events must be determined from the relevant model states.

The whole set of events may be classified like this:

- The *Init_PackageTrackingDomain* event.
- *PassSensor* events, each of which is:
  - a *ReadDest* event; or
  - a *Deposit* event that is also a *RightBin* event; or
  - a *Deposit* event that is also a *WrongBinMessage* event; or
  - a *Transfer* that is also an *ArriveAndFlip* event; or
  - a *Transfer* that is also an *ArriveNoFlip* event.

ReadDest events are fully determined in the Package Tracking view. As mentioned in Section 4 above, the value of the destination d in a *ReadDest* event is determined by the state—perhaps a machine register—shared by the reader and the machine.

Similarly, *WrongBinMessage* and *RightBin* events are also fully determined in the Package Tracking view, where their specifications include the specification of the more general *Deposit* event class: each *WrongBinMessage* and each *RightBin* event is also a *Deposit* event. The values of *actual!* and *desired!* in *WrongBinMessage* events are determined from the sensor se of the *Deposit* event and the *queue* model of the *PackageTrackingDomain*.

However, *ArriveAndFlip* and *ArriveNoFlip* events are not fully determined by the Switch Control view in which they are specified. Each of these events is also a *Transfer* event, specified in the Package Tracking view. The values of their switch sw and destination d components must be determined from the sensor *se* of the *Transfer* event, using information from the Router Topology view. So we need schemas for the two combinations:

$$\begin{array}{|l} flipped : DIR \rightarrowtail\!\!\!\twoheadrightarrow DIR \\ \hline flipped(R) = L \\ flipped(L) = R \end{array}$$

$$\begin{array}{|l} \textit{TransferAndFlip} \\ \hline \Xi RouterRequestsAndRules \\ Transfer \\ ArriveAndFlip \\ \hline (se \mapsto sw) \in guards \\ d = \mathsf{head}\ queue(precedes(se)) \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{TransferNoFlip} \\ \hline \Xi RouterRequestsAndRules \\ Transfer \\ ArriveNoFlip \\ \hline (se \mapsto sw) \in guards \\ d = \mathsf{head}\ queue(precedes(se)) \\ \hline \end{array}$$

Now we can define the events as operations on the *PackageRouterControl* state, and write the necessary event classification:

$$\begin{array}{l} Init\_PackageRouterControl \mathrel{\widehat=} \\ \quad \Delta PackageRouterControl \wedge \Xi RouterRequestsAndRules \wedge \\ \quad Init\_PackageTrackingDomain \end{array}$$

$$\begin{array}{l} PassSensor \mathrel{\widehat=} \\ \quad \Delta PackageRouterControl \wedge \Xi RouterRequestsAndRules \wedge \\ \quad (ReadDest \vee WrongBinMessage \vee RightBin \vee \\ \quad\ \ TransferAndFlip \vee TransferNoFlip) \end{array}$$

8 Discussion

8.1 Problem Frames

The original motivation of the idea of problem frames was chiefly methodological [6]. A problem frame bounds a class of problem for which an effective and systematic method is known. The method can be readily applied to a problem of the class because the method is expressed in terms of the *principal parts* of the problem. The JSD method, for example [5], treats *Dynamic Information* problems. In JSD the first step is to describe the *Subject Domain*—which in JSD is called the *Real World*— as a collection of concurrent sequential processes each with a regular structure. The same process descriptions are then used to construct a model of the *Subject Domain* within the *System*. In a later step, further processes are added for handling the *Information Requests* in accordance with the *Information Rules*: in JSD these parts are called the *Function* of the *System*.

This association of problem frames with methods goes some way to filling the gap between teaching problems and realistic problems. A teaching problem is an exercise in the application of a highly circumscribed method. For example, students of abstract data type specification may first discuss a specification of a stack; then they are asked to write their own specifications of a queue, or of a bag, or of a double-ended queue. Similarly, students of finite automaton theory may solve problems in which deterministic machines are derived from non-deterministic machines, or from regular grammars. These problems are not trivial, but they fall clearly into the classes that can be solved by the techniques being taught.

But such teaching problems, and the associated techniques, are not quite large enough, and not quite rich enough, to suggest clearly identifiable subproblems in a realistically rich and complex problem. It is not easy to see what abstract data types, and what finite automata, are needed in an accounting system or in a telephone switching system. Problem frames offer an approach to analysing more complex problems like these, because they offer structures—of *principal parts*—that can be matched against the problem domain or environment, and used to identify aspects—rather than parts—of the problem.

8.2 Reusability of Views

In applying a frame to a subproblem view it is helpful to imagine that the other subproblems are already solved. Their solutions provide the context of the subproblem in hand. For example, in applying the *Dynamic Information* frame to the Package Tracking subproblem, we imagined that the *System*—the machine we were building—had direct access to the *precedes* and *bin* relations, and to the identity of the *reader* sensor. In applying the *Control* frame to the Switch Control subproblem, we imagined that the *choice* and *empty* relations provided the machine with directly accessible information about the available routes and the traffic in the switches; and also that each *ArriveAtSwitch* event directly indicated the identity of the switch and the destination of the arriving package.

The resulting decomposition produced subproblems with a significant potential for reuse. In each frame no more is assumed about the problem context than is necessary to the *solution task* of the frame. Examples of this indeterminacy have already been pointed out, but perhaps have been somewhat masked by the use of the same names in different views. For example, the Package Tracking view might have been expressed in terms of *Travellers* passing *Checkpoints*. A *Checkpoint* may serve as a *Startpoint* or *Finishpoint*, but not both at once. In the complete problem, the *Checkpoints*, of course, are the sensors; a *Startpoint* is a reader; and a *Finishpoint* is a sensor leading to a bin. But the Package Tracking view assumes only that: when a *Traveller* passes a *Checkpoint* it is known whether the *Traveller* is starting or finishing—that is, whether the *Checkpoint* is currently a *Startpoint* or a *Finishpoint*; when it passes a *Startpoint* its desired *Finishpoint* is known and fixed; when it passes any intermediate *Checkpoint* its previous *Checkpoint is known*; and that there is no overtaking between *Checkpoints*.

The views developed in our treatment of the problem were not devised with reuse in mind. Rather, the approach merely avoided building into each view assumptions and features that were irrelevant to the view—and therefore most likely to inhibit subsequent reuse. The Switch Control view, for example, deals only with the events involving the switches. It assumes only that: each package arriving at a switch has a destination that may or may not be reachable from that switch; that there is a constraint *empty* that may preclude flipping a switch; and that if flipping is desirable and not precluded the machine can perform it. Such a subproblem seems likely to appear in many switching problems, perhaps involving railway signalling, control of motor traffic, or even the transmission of message packets.

8.3 Z Specifications

Z seems especially well suited to specifications in this style. Some of the benefits accrue from features shared by many specification languages. The ability to write implicit invariants—shared with VDM and Larch, for instance—is crucial to describing the relationships between the states of the various views without indicating how, in the implementation, the relationships are to be maintained.

Other benefits are unique to Z. The underlying treatment of all functions, sequences, bags, mappings, and so on, as relations contributes greatly to the terseness of the invariants connecting the views, since it obviates the need for frequent type coercions. Schema structuring is a great help too. Since state invariants and operations are just logical assertions, they may be conjoined flexibly and simply. Other specification languages do not offer such freedom of structuring. Moreover, because the pre-conditions of two operations are always composed (in the style advocated here) in the same way as the post-conditions, by conjunction or disjunction, it is useful that Z combines the pre- and post-condition of an operation into a single assertion.

View specification does, however, expose some deficiencies of Z. Most significantly, schema inclusion does not preserve the origin of the state components. Having formed *PackageRouterControl*, for example, we cannot ask whether *choice* belongs to *RouterRequestsAndRules* or to *PackageTrackingDomain*. Schema inclusion is purely syntactic, and the components originating in different schemas are thrown together in a single unstructured namespace. Intuitively this seems wrong, since it leaves only synactic and not semantic evidence of the view structuring in the final specification. It also has serious pragmatic implications. Names chosen for components belonging to different views may accidentally clash when the views are composed, and the invariants relating components of different views are hard to read. We could of course adopt a convention, such as prefixing every name by the name of the view it belongs to. It would much better, however, for the language itself to provide some structuring of the namespace, so that within a view components may have short names, and in the composition they are labelled with the names of their views.

The type system of Z is not ideal for view specification. Ideally we would like to treat a name like *SENSOR* sometimes as a set of values—taken from an underlying type *INDIVIDUAL*—and sometimes as a type. We treated the reader awkwardly as a sensor in the Router Topology view, where it would have been more natural to declare a fresh type *READER*, disjoint from *SENSOR*. Strong typing of the Router Topology schemas would then have ensured no confusion of the reader with the sensors. The Package Tracking view, on the other hand, associates queues with checkpoints. In its schemas, we therefore would like to use a type *CHECKPOINT*.

In the composition of the views, we would then assert that the checkpoints correspond to exactly the sensors *and* the reader. But unfortunately the assertion

$$CHECKPOINT = SENSOR \cup READER$$

is a type error. Strong typing, it seems, is a local property of a view, and should not extend beyond it. Z forces an early decision of whether a set should be treated as a type, seriously compromising the independence of views. There are almost no sets that can be confidently declared to be disjoint without knowing which views may be added subsequently. Even *SWITCH* and *PIPE* and *SENSOR*, for example, might become subsets of *PART* in a Maintenance view that keeps track of the condition of the router's components.

Finally, there is the controversial question of the role of convention in Z. A schema is just a logical assertion, and its standard interpretation as an operation is not part of the language definition. Our specification assumes a number of further conventions that we have not articulated in detail. A precondition is a guard and not a disclaimer; in bad states, it precludes execution of the operation rather than allowing it while insisting that its effect is unspecified. We use conjunction and disjunction of operation schemas to classify events. Each operation schema name in the definition of *PassSensor* denotes a set of events; any particular event may belong to more than one event class.

8.4 Integration Mechanisms

In software development, which is concerned to create machines that will interact with the world, the foundation of meaning must be *observable phe-*

nomena—states and events—in the machine and in the parts of the world that constitute the application domain. The basis of integration—integration of one view with another, and integration of the machine into the world—is *shared phenomena*. A shared event or state occurs both in the machine and in the application domain; or it is described both in one specification view and in another.

Integration by shared phenomena demands a sound treatment of two aspects of sharing. First, a shared phenomenon is viewed differently by different sharers. We have already mentioned the example of the *reader*, which should properly be viewed as a unique individual of type *READER* in the Router Topology view, and as one of many *CHECKPOINT*s in the Package Tracking view. Different views and classifications of shared phenomena are simply a microcosm of the different views we adopt of larger aspects of the problem.

Second, for both shared events and shared states it is important to specify the locus of its control. The *Init_PackageTrackingDomain* event is controlled by the machine; but the *ReadDest* and *Transfer* events are controlled by the application domain. The *ArriveAndFlip* events should really be treated as event pairs: in the first event of each pair, controlled by the domain, a package arrives at a switch; in the second event, controlled by the machine, the machine flips the position of the switch. The two events of the pair are combined in our specification, because Z lacks a convenient mechanism for expressing their relationship. The schema expressions:

$$Arrive \mathbin{;} Flip$$

and

$$Arrive \gg Flip$$

do not describe pairs of operations. Rather, they describe single operations by identifying the post-state of one schema with the pre-state of the other and hiding the linking state. Any notion that there are two operations, one following the other in time, would depend on interpretation by a special—and heterodox—convention, supported by informal commentary.

We believe that the interpretation of the specification in terms of observable phenomena should not be relegated to informal, unstructured commentary, but should be governed, for a given style of specification, by well-understood and precisely articulated rules. This remains to be done. As a first step, a translation into a simpler formal model closer to the observed phenomena—such as a labelled transition system—might help. Such a feature, combined with good support for multiple views of shared phenomena, would go far to encourage specification in the style we advocate.

References

[1] Robert M Balzer, Neil M Goldman and David S Wile; Operational Specification as the Basis for Prototyping; ACM Sigsoft SE Notes Volume 7 Number 5 pages 3-16, December 1982; reprinted in New Paradigms for Software Development; ed W W Agresti; IEEE Tutorial Text, IEEE Computer Society Press, 1986.

[2] Martin S Feather, Stephen Fickas, and B Robert Helm; Composite System Design: the Good News and the Bad News; in Proceedings of the 6th RADC Conference on Knowledge-Based Software Engineering; IEEE Computer Society Press, 1992.

[3] David Garlan and Mary Shaw; An Introduction to Software Architecture; in Advances in Software Engineering and Knowledge Engineering Volume 1, V Ambriola and G Tortora eds; World Scientific Publishing Co, New Jersey, 1993.

[4] Daniel Jackson; Structuring Z Specifications with Views; Technical Report CMU-CS-94-126, School of Computer Science, Carnegie Mellon University, March 1994.

[5] Michael Jackson; System Development; Prentice-Hall International, 1983.

[6] Michael Jackson; Software Development Method; in A Classical Mind: Essays in Honour of C A R Hoare; A W Roscoe ed; pages 211-230; Prentice-Hall International, 1994.

[7] Michael Jackson; Problems, Methods and Specialisation; SE Journal Volume 9 Number 6 pages 249-255, November 1994; edited and abridged in IEEE Software Volume 11 Number 6 pages 57-62, November 1994.

[8] W Lewis Johnson; Deriving Specifications from Requirements; in Proceedings of the 10th International Conference on Software Engineering; IEEE Computer Society Press, 1988.

[9] Bashar Nuseibeh, Jeff Kramer and Anthony Finkelstein; Expressing the Relationships Between Multiple Views in Requirements Specification; Proceedings of 15th International Conference on Software Engineering, pages 187-196; IEEE Computer Society Press, 1993.

[10] Gerald Kotonya and Ian Sommerville; Viewpoints for Requirements Definition; Software Engineering Journal Volume 7 Number 6, pages 375-387, November 1992.

[11] William Swartout and Robert Balzer; On the Inevitable Intertwining of Specification and Implementation; Comm ACM Volume 25 Number 7 pages 438-440, July 1982.

[12] Pamela Zave and Michael Jackson; Conjunction as Composition; ACM Transactions on Software Engineering Methodology, Volume 2 Number 4 pages 379-411; October 1993.

Appendix

The formal parts of the specification are brought together here. They are given in the order of their original presentation, except that the following declarations are given first, here:

$$[SWITCH, PIPE, BIN, SENSOR]$$

$$DEST == BIN$$
$$DIR ::= L \mid R$$

$$\begin{array}{|l}
flipped : DIR \rightarrowtail\!\!\!\!\twoheadrightarrow DIR \\
\hline
flipped(R) = L \\
flipped(L) = R
\end{array}$$

Router Topology View

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{RouterDomain}} \\
top : PIPE \\
reader : SENSOR \\
sensUp, sensDn : PIPE \rightarrowtail SENSOR \\
enters : PIPE \twoheadrightarrow\!\!\!\!\!\!+\;\; SWITCH \\
pipeL, pipeR : SWITCH \rightarrowtail PIPE \\
fills : PIPE \twoheadrightarrow\!\!\!\!\!\!+\;\; BIN \\
\hline
reader \notin \operatorname{ran}(sensUp \cup sensDn) \\
top \in \operatorname{dom} enters \\
\operatorname{dom} enters \cap \operatorname{dom} fills = \varnothing \\
\operatorname{ran} sensUp \cap \operatorname{ran} sensDn = \varnothing \\
\operatorname{ran} pipeL \cap \operatorname{ran} pipeR = \varnothing \\
\operatorname{ran} pipeL \cup \operatorname{ran} pipeR = (\operatorname{dom} enters \setminus \{top\}) \cup \operatorname{dom} fills \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{RouterStaticInformation}} \\
RouterDomain \\
flow : PIPE \leftrightarrow PIPE \\
reachL, reachR : SWITCH \leftrightarrow BIN \\
\hline
flow = enters \fatsemi (pipeL \cup pipeR) \\
reachL = pipeL \fatsemi flow^{*} \fatsemi fills \\
reachR = pipeR \fatsemi flow^{*} \fatsemi fills \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{RouterRequestsAndRules}} \\
RouterStaticInformation \\
guards : SENSOR \rightarrowtail\!\!\!\!\!\!+\;\; SWITCH \\
way : (SWITCH \times DEST) \leftrightarrow DIR \\
precedes : SENSOR \rightarrow\!\!\!\!\!\!+\;\; SENSOR \\
\hline
guards = sensDn^{\sim} \fatsemi enters \\
way = (reachL \times \{L\}) \cup (reachR \times \{R\}) \\
precedes = \{reader \mapsto sensUp(top)\} \cup \\
\quad \{p : PIPE \bullet sensUp(p) \mapsto sensDn(p)\} \cup \\
\quad \{(p, p') \in flow \bullet sensDn(p) \mapsto sensUp(p')\} \\
\hline
\end{array}
$$

Package Tracking View

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PackageTrackingDomain}} \\
\hline
queue : SENSOR \pfun \operatorname{seq} DEST \\
precedes : SENSOR \pfun SENSOR \\
bin : SENSOR \pfun BIN \\
reader : SENSOR \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init\_PackageTrackingDomain}} \\
\hline
PackageTrackingDomain' \\
\hline
queue' = (\operatorname{dom} bin) \lhd (SENSOR \times \langle\rangle) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transfer}} \\
\hline
\Delta PackageTrackingDomain \\
se : SENSOR \\
d : DEST \\
\hline
se \neq reader \land se \notin \operatorname{dom} bin \\
d = \mathsf{head}\, queue(precedes(se)) \\
queue' = queue \oplus \\
\quad \{precedes(se) \mapsto \mathsf{tail}\, queue(precedes(se)), \\
\quad\; se \mapsto queue(se) \frown \langle d \rangle\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadDest}} \\
\hline
\Delta PackageTrackingDomain \\
se : SENSOR \\
d : DEST \\
\hline
se = reader \\
queue' = queue \oplus \{reader \mapsto queue(reader) \frown \langle d \rangle\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Deposit} \\
\Delta \textit{PackageTrackingDomain} \\
se : \textit{SENSOR} \\
d : \textit{DEST} \\
b : \textit{BIN} \\
\hline
se \in \operatorname{dom} bin \wedge b = bin(se) \\
d = \mathsf{head}(queue(precedes(se))) \\
queue' = queue \oplus \{precedes(se) \mapsto \mathsf{tail}\, queue(precedes(se))\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RightBin} \\
\textit{Deposit} \\
\hline
b = d \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{WrongBinMessage} \\
\textit{Deposit} \\
actual!, desired! : \textit{BIN} \\
\hline
b \neq d \\
desired! = d \\
actual! = b \\
\hline
\end{array}
$$

Switch Control View

$$
\begin{array}{|l}
\hline
\textit{SwitchSettings} \\
setting : \textit{SWITCH} \longrightarrow \textit{DIR} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SwitchesModel} \\
\textit{SwitchSettings} \\
choice : (\textit{SWITCH} \times \textit{DEST}) \leftrightarrow \textit{DIR} \\
empty : \mathbb{P}\, \textit{SWITCH} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } ArriveAndFlip \\
\quad \Delta SwitchModel \\
\quad sw : SWITCH \\
\quad d : DEST \\
\hline
\quad sw \in empty \\
\quad choice(\!| sw, d |\!) = \{flipped(setting(sw))\} \\
\quad setting' = setting \oplus \{sw \mapsto flipped(setting(sw))\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } ArriveNoFlip \\
\quad \Delta SwitchModel \\
\quad sw : SWITCH \\
\quad d : DEST \\
\hline
\quad \neg \mathrm{pre}\ ArriveAndFlip \\
\quad setting' = setting
\end{array}
$$

Connections

$$
\begin{array}{l}
\textbf{schema } PackageRouterControl \\
\quad RouterRequestsAndRules \\
\quad PackageTrackingDomain \\
\quad SwitchesModel \\
\hline
\quad bin = sensDn^{\sim} \mathbin{;} fills \\
\quad choice = way \\
\quad empty = guards\ (\!|\ \mathrm{dom}(queue \rhd \langle\rangle) |\!)
\end{array}
$$

$$
\begin{array}{l}
\textbf{schema } TransferAndFlip \\
\quad \Xi RouterRequestsAndRules \\
\quad Transfer \\
\quad ArriveAndFlip \\
\hline
\quad (se \mapsto sw) \in guards \\
\quad d = \mathsf{head}\ queue(precedes(se))
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TransferNoFlip}} \\
\hline
\Xi RouterRequestsAndRules \\
Transfer \\
ArriveNoFlip \\
\hline
(se \mapsto sw) \in guards \\
d = \mathsf{head}\ queue(precedes(se)) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
Init\_PackageRouterControl \mathrel{\widehat=} \\
\quad \Delta PackageRouterControl \land \Xi RouterRequestsAndRules \land \\
\quad Init\_PackageTrackingDomain
\end{array}
$$

$$
\begin{array}{l}
PassSensor \mathrel{\widehat=} \\
\quad \Delta PackageRouterControl \land \Xi RouterRequestsAndRules \land \\
\quad (ReadDest \lor WrongBinMessage \lor RightBin \lor \\
\quad\ \ TransferAndFlip \lor TransferNoFlip)
\end{array}
$$