ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Antony Bryant and Lesley Semmens (Eds)

Methods Integration

Proceedings of the Methods Integration Workshop, Leeds, 25-26 March 1996

Paper:

$JSD \cong \Delta CSP \oplus TLZ$ - A Case Study

Michael G. Hinchey

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

$JSD \triangleq \Delta CSP \oplus TLZ$ — A Case Study

Michael G. Hinchey

New Jersey Institute of Technology

Department of Computer and Information Science

University Heights, Newark, NJ 07102, USA

and

University of Limerick

Dept. of Computer Science and Information Systems

Limerick, Ireland

Abstract

We present a case study in the use of JSD, a popular structural design method, as a unifying mechanism for formal notations addressing different aspects of a system design. An asynchronous variant of CSP is used to address issues relating to time-ordering of events and communication between processes, while TLZ (a hybrid of Z and TLA) addresses state-based aspects of the system design and permits the expression of timing constraints and fairness conditions. The result is a hybrid real-time design method, appropriate for particular classes of real-time systems. The novelty is that a structural design method (with simplified semantics) serves to provide various views of the design, with the benefits and proof systems of the various formal methods being maintained.

1 Introduction

As computer systems have increased in size and complexity, both academia and industry alike have turned their attention to verifying the correctness of such systems.

Over the last thirty years or so, many techniques have evolved for the specification and verification of complex computer systems. The techniques that are currently enjoying most popularity — so-called ‘formal methods’ — are based on describing computer systems in terms of discrete mathematical structures, and subsequently proving properties of the described system.

The advent of concurrent and real-time systems has complicated matters somewhat, however. Concurrent systems are inherently more complex, offering the possibility of parallel and distributed computation, and the increased processing power that this obviously facilitates.

Standard methods of specifying and reasoning about computer systems are not sufficient for use with concurrent systems. They do not allow for side-effects, the occurrence of multiple events simultaneously, nor for the synchronization required between processes to ensure data integrity, etc. However, techniques based on formal methods have proven to be more successful than other (informal) structured techniques and various approaches based on temporal logic.

A realization that concurrent systems are ‘reactive’ systems has highlighted the fact that current design methods are insufficient. A design method is required that supports a reactive model of computation, with asynchronous communication and support for high-levels of concurrency (even if the eventual implementation may be in a uniprocessor environment).

For the purposes of this paper, and elsewhere [16], real-time systems are viewed as a subclass of concurrent systems — systems which are generally in continual interaction with their environments, and whose purpose is to react accordingly, and within given timing constraints to certain conditions in those environments. Such applications are typically large, complex, and difficult both to comprehend and design [10]. To overcome these problems, a large number of design methods have evolved, each claiming varying levels of appropriateness for use with real-time applications.

Unfortunately, many of these so-called ‘methods’ are not methods at all, but little more than visual programming languages or specification languages, many of which have no formal basis.

That is not to say that they are not useful; in fact some have been used successfully with *particular* real-time applications. But, their *general applicability* is questionable.

2 The ‘ideal’ design method

We can determine a set of seven essential criteria which the ‘ideal’ real-time design method should satisfy:

- (1) **It should be based on a reactive model of computation.** Real-time applications are reactive rather than proactive — they suffer events rather than generate them. Where they do actually generate events this is only in response to previous events.

Information systems and other static realities may be described in terms of entities having fixed properties and fixed relationships to each other (i.e., in terms of data flows and databases), but real-time applications represent a dynamic reality. They require an *event model* which permits the expression of timing constraints, and which can describe entities that have properties and relationships that vary over time and that suffer events that occur in a particular order.

- (2) **It should support high-levels of abstraction.** The method should enable us to describe the application in terms of highly abstract objects, and postpone design decisions until the later stages of the design.

If we begin to express the design in terms of domain-specific concepts such as semaphores, mailboxes and message-queues too early, then there is a danger that we will over-specify the application, and unnecessarily constrain the range of acceptable implementations.

- (3) **It should allow us to model concurrency.** Concurrency is inherent in the modelling of all real-time applications [10]. Even if the final implementation will not be distributed, it will probably involve some degree of multi-processing. More importantly, however, real-time applications will have to react promptly to conditions in the environment arising from various sources simultaneously. To cope with this, the design method must allow us to model the system using a large number of ‘processes’ or ‘objects’ executing concurrently.

Clearly, such concurrent execution might not be feasible in the current implementation environment (due to monetary or hardware constraints, etc.), so the method should provide some means of reducing the level of concurrency prior to implementation. It should be possible to reduce the number of processes in the design without compromising the constraints we have expressed, nor the correctness of the model; ideally this should be achieved through some form of correctness-preserving algebraic manipulation.

- (4) **It should be based on a bottom-up approach.** Top-down approaches to problem-solving involve splitting a problem into a number of subproblems and repeating the process until the sub-problems are such that they can be easily implemented in a programming language. This has proven to be very successful in practice, but requires that we fully understand the system at the outset.

Unfortunately, with real-time applications, this will not always be the case. More likely, as we progress with the design, we will uncover additional conditions and events that we previously hadn’t considered. For this reason, a bottom-up approach that allows us to add more detail and specialization to the model as we gain further insights into the operation of the system, is more beneficial.

- (5) **It should support structured diagramming techniques.** System procurers and end-users are unlikely to accept designs and specifications expressed in mathematical notations that are unfamiliar to them.

Graphical notations can be used to give high-level overviews of a system that are accessible to non-specialists. They can act as an aid to clear-thinking about the system, and help in enforcing good structure.

- (6) **It should have a formal semantics.** A formal semantics enables us to propose and formally verify properties of the system before implementation. For example, we can prove that if certain conditions arise, then our model will enable us to recover from these.

A formal semantics also allows us to prove the correctness of the implemented system with respect to its specification (of course, it is meaningless to speak in terms of absolute correctness). The semantics will ideally be expressed in terms of some sort of propriety or extended logic as standard logics such as predicate and temporal logics when used alone have proven to be too restrictive and too difficult to reason with.

- (7) **It should support the mechanical refinement of the design to executable code.** ‘Mechanical’ does not necessarily mean automatic, although it does indicate that the process could be automated. There are some arguments to suggest that refinement should not be automated as it restricts us to a particular implementation. But, refinement tools can certainly help in determining the proof obligations that must be discharged at each stage in the refinement process.

3 Real-time design methods

Most of the design methods in popular use will meet at least some of these requirements. But, no existing design method — neither structured nor formal — can meet all of these requirements.

Structured methods fail on many counts: most of them originated to deal with traditional ‘transformational’ computation, which is inherently different from the reactive or event model that we require.

Those methods specifically intended for use with real-time applications tend to introduce domain-specific concepts to model systems. While these may be useful in the later stages of design, when used too early in the development they severely restrict the levels of abstraction.

The main problem with structured methods, however, is their lack of a formal semantics. There is a move towards introducing more formalization in the way that components of diagrams are connected, but at best this results in semi-formal methods. On the other hand, most formal methods have no graphical representation, and their notations are often considered (rightly or wrongly) to be difficult to understand.

While the more popular formal methods do have a formal semantics and support the mechanical refinement to executable code, most fail to support the concept of concurrency. Exceptions are process algebras, which describe systems in terms of their algebras of communicating processes, executing concurrently. However, they fail to pay sufficient attention to modelling the state of the system. There is now a wide consensus that no single formal method is sufficient to describe all aspects of a system. We actually require a process algebra to describe concurrency and the time-ordering of events, and a state-based or model-based formal method to model the state of the system and changes in that state. But how do we integrate two such differing languages?

A potential solution lies in the integration of structured and formal methods.

Such an integration offers us the best of both worlds — a provably correct design method, with a graphical notation that can be readily understood by non-specialists. In effect two views of a system are presented, allowing us to concentrate on those aspects that are of interest to us.

4 JSD as a Real-Time Design Method

With the introduction of JSP (Jackson Structured Programming)[19], Michael Jackson greatly simplified the task of program design. The success of his approach lies in its intuitiveness and the simple graphical notation that facilitates the design of programs which obey the laws of structured programming — that use only the simple constructs of sequence, selection and iteration, and avoid the ‘harmful’ `goto`-statement or its equivalent.

Eight years later, with the evident need to address *system design* rather than merely program design, Jackson introduced JSD, or Jackson System Development (often erroneously, but still appropriately, referred to as Jackson Structured Design).

The method incorporates JSP itself, in that JSP descriptions of each program or *process* in the system are combined by placing such processes running in parallel and in communication over various types of buffered communication

channels. As such, the description of a system is at two levels — each process is described in terms of JSP, while the entire system is described as a network of such processes, executing independently. A process becomes suspended when it requires a particular input, and resumes again when that input is available to it.

As the method does not address process scheduling, clearly a certain degree of nondeterminism is inherent [18]. Although generally considered inappropriate in conventional (sequential) programming languages, nondeterminism can be advantageous in system design, and particularly in the development of concurrent systems. JSD also supports high levels of abstraction in that there are generally more processes in the network than available processors in the implementation environment. In the implementation phase of JSD, the number of processes in the network is reduced, often just to a single process (for execution in a single processor environment). This clearly represents some degree of multiprogramming.

Multiprogramming represents a class of concurrency whereby multiple processes ‘time-share’ a single processor. As the cost of processors has fallen over the past few years, concurrent programming has come more to mean multiprocessing — the execution of x processes on y processors, with $1 < y \leq x$.

More and more these processors are no longer part of the same computer terminal, nor in the same building, nor even the same country. The trend is certainly towards these so-called ‘distributed systems’. Such systems are frequently classed as ‘safety-critical’, meaning that their failure could result in the loss of human life, the mass destruction of property, or great financial loss.

One of the ‘critical’ criteria of such systems is that they react accordingly to certain conditions or certain events within given time-scales. Such systems are classed as ‘real-time’ systems.

Real-time systems are perhaps some of the most complex computer systems in existence. Certainly they are amongst the most difficult to design appropriately. A plethora of design methods have been proposed for them, but very few succeed in satisfying all of the essential criteria of an appropriate real-time design method, as described in the previous section.

JSD does however seem very appropriate — it supports high-levels of abstraction, ignoring the number of physical processors at the outset, inherently supports concurrency, and timing can be supported by the addition of specialized ‘clock’ processes. It is no surprise then that the method has been advocated for use in the design of concurrent and real-time systems [18, 26, 27], and has been used successfully in particular real-time applications.

Most structured methods employ some variant of the Data Flow Diagram (DFD), thereby emphasizing *data flow*. This, added to complex means by which processes must gain permission from centralized state-transition managers only serves to introduce unnecessary complexity into the design. Data-flow is considered to be a by-product of a JSD specification, however.

While Structured Analysis/Structured Design starts with a data-flow model of the system, it doesn’t model the possible sequence of events that may occur. Nor do the other popular design methods. JSD requires the derivation of an explicit model of reality, facilitating a better understanding of the proposed system.

This model of reality is expressed at high levels of abstraction in JSD, with the model strictly separated from the system function, with all implementation issues postponed until the implementation phase, where an executable system is derived by the application of a set of correctness-preserving transformations. Other methods tend to lead to design decisions too early in the development, constraining the range of acceptable implementations. Many methods do not support an explicit communication mechanism, limiting communication to shared variables and parameter passing. The shared memory model is certainly declining in popularity, as it greatly restricts possible implementations, emphasizing multiprogramming rather than multiprocessing and distributed processing. A CSP-like message-passing model [17], as supported by JSD, offers higher levels of abstraction and a wider range of possible implementations, whilst avoiding the need to address problems such as mutual exclusion.

Using the Theory of Asynchronous Processes [21], a variant of CSP that supports the modeling of asynchronous communication, for a formal investigation, Barnes [1] has encountered undesirable behaviour of a real-time system designed with JSD. It was found that the ‘Hi-Ride’ elevator [20] would refuse to service a floor, or fail to stop at the top floor of a building.

This can be attributed to two limitations of JSD, which must be overcome if it is to be used successfully with real-time systems:

- (1) With JSD there is no guarantee that a message sent on a data-stream will be received by the destination process

within a given time interval, as infinite buffering on data-streams is assumed. In addition, a destination process might never read from a particular data-stream, as it may be suspended awaiting input on some other data-stream.

- (2) In real-time systems, we would normally expect that when a process inspects the state of another process in the system, it inspects the *current* state of that process. For example, a flight-control system might refuse to lower the landing-gear if, on inspecting the process monitoring altitude, it (erroneously) believes the aircraft to be too high.

In JSD, however, the State-Vector Inspection mechanism returns some state that the inspected process has been in, not necessarily the current state. That is to say, out-of-date state-vector inspections are admitted by the JSD method.

For real-time applications such behaviour is completely unacceptable, and the behaviour admitted by the JSD model does not always correspond with our intuitions about real-time systems. However, a formal investigation using an alternative formal language indicates that these difficulties may be overcome. The approach enables us to determine (relatively simple) modifications to the JSD model, which results in a structured method that may be used very successfully with certain classes of real-time system, in particular with control systems.

5 Summary of changes to JSD

As the result of a formal investigation of JSD using RPT^+ , we can identify a number of changes which will make JSD more appropriate for the particular classes of system we're dealing with.

These changes to JSD can be summarized as:

1. Processes are considered to be receptive [6], always willing to accept input, even if that can result in undesirable future behaviour.
2. Buffering on data-streams has been removed. Data-streams can now also carry various kinds of messages (whereas JSD assumes all data-streams to be allocated to messages of a specific type) as long as these are permitted by the relevant alphabet declarations.
3. No special notation for different communication mechanisms is required, as all communication is over standard data-streams.
4. No “magical” merge processes are required, simplifying later implementation.
5. State-vector inspections are now implemented by data-streams and return the current state, rather than some state that the inspected process has been in during its lifetime.
6. Processes have an interpretation in the RPT^+ process algebra.
7. JSD Structure Diagrams have been augmented with firing and termination conditions, from which pre- and post-conditions of operations may be determined.
8. Structure Diagrams may be systematically transformed to a Z specification. Following the principles of formal specification, such a specification should be far more useful than the structured text that is traditionally derived from JSD Structure Diagrams in an *ad hoc* manner.
9. Timing constraints and fairness conditions can be expressed over the operations (leaves) of a Structure Diagram.

We believe that such a coupling of structured and formal methods results in a highly usable development method.

In a conference paper, we are constrained in the amount of material that can be presented. Therefore, we will concentrate on a substantial case study that illustrates some of the ideas, and hopefully will generate some discussion, leaving the significant technical details for a forthcoming technical report.

6 Case Study : The ‘Hi-Ride’ Elevator System

The case study in this paper describes the translation of a JSD specification (both the System Specification Diagram and relevant Structure Diagrams) into an equivalent formal specification, expressed in RPT^+ and TLZ. The aim is to illustrate the usefulness of the technique in addressing both the time-ordering of events and the system state, while maintaining a simple graphical representation.

The example used is Jackson’s well-known ‘Hi-Ride’ elevator [20]. This example has been used to examine the potential use of JSD with real-time and control systems [9], and in formal investigations of JSD with VDM [5], CSP [31] and the Theory of Asynchronous Processes [1]. Variants of the case study have also been used to examine the potential of Z for use in the specification of real-time systems (e.g. [8]). Our method applies to many classes of system, but, like others, we find this elevator example to be very appropriate for expository purposes.

6.1 Problem Statement

We borrow the problem statement directly from Jackson [20].

The Hi-Ride Elevator Company is installing elevators in a small building of six floors. There are two elevators, each serving all the floors. At each floor, except the top floor, there is a button which users can press to summon an elevator to take them upwards; at each floor, except the ground floor, there is a similar button for downwards travel. Inside each elevator there are six buttons marked with the floor numbers. There is a pair of doors at each floor, and another pair on each elevator. The elevators are raised and lowered by cables which are wound and unwound by motors positioned above the top floor. At each floor, in each elevator shaft, there is one of the company’s famous sensors, operated by a small wheel attached to the elevator: when the elevator is within 15 cm of the home position of that floor, the sensor is depressed by the wheel, and closes an electrical switch. Other, different, sensors are provided which sense whether the various pairs of doors are in the fully open or fully closed position The computer system will schedule the travel of the elevators according to the users’ requests for service, and will produce commands for the motors and the lights which are associated with the buttons.

We will consider the more sophisticated version of the elevator controller, whereby unnecessary upward and downward legs of a journey may be avoided. We will not consider the opening and closing of doors until we consider the individual Structure Diagrams.

However, we will consider only a single elevator system; a second elevator can be added easily afterwards with appropriate labelling.

6.2 JSD Specification

The System Specification Diagram of the elevator system is as shown in Figure 1. This is drawn directly from [20]. The entities we employ in our method are therefore those identified by Jackson as being relevant (that is, those entities about which information is available and suitable for modelling). We do not, however, require to model level 0 processes (explicitly denoting real-world entities) as in Jackson’s approach, although we certainly could do so applying the same principles as employed here.

We must also consider the Structure Diagrams of each of the three main process types, namely `BUTTON`, `ELEVATOR` and `ELCONTROL`. These are also drawn from [20], except that they have been augmented with firing and termination conditions, and an extra level has been introduced for readability. Later we will consider their translation to TLZ.

The Structure Diagram of the `BUTTON` process is as shown in Figure 2.

The Structure Diagram of the `ELEVATOR` process is as shown in Figure 3. For ease of readability, the firing and termination conditions have been labelled **A** to **F**, and are stated here.

Figure 1: System Specification Diagram for the elevator system.

Figure 2: Structure Diagram of the BUTTON process.

Figure 3: Structure Diagram for the ELEVATOR process.

<p>A $floor = ground \wedge$
$dir = up$</p>	<p>B $floor' = ground + 1 \wedge$
$dir' = dir$</p>
<p>C $(floor = m? + 1 \wedge$
$dir = down)$
$\vee$
$(floor = m? - 1 \wedge$
$dir = up)$</p>	<p>D $floor' = m \wedge$
$dir' = dir$</p>
E $floor = m$	<p>F $(floor = top \wedge$
$dir = up$
$\Rightarrow dir' = down)$
$\wedge$
$(floor = ground \wedge$
$dir = down$
$\Rightarrow dir' = up)$
$\wedge$
$(floor < top \wedge$
$ground < floor$
$\Rightarrow dir' = dir)$
$\wedge$
$floor = floor'$</p>

As already stated, we consider the more sophisticated controller of Jackson [20], the Structure Diagram of which is given in Figure 4. As before, we label the firing and termination conditions **A** to **V**.

Figure 4: Structure Diagram of the elevator controller.

- A** $motor = off \wedge$
 $motordir = up \wedge$
 $(u \cup d \cup e) \subseteq \{ground\}$
- B** $motor' = on \wedge$
 $motordir' = motordir \wedge$
 $ground \notin (u' \cup e') \wedge$
 $(u' \cup d' \cup e') \neq \{\}$
- C** $f? < top \wedge$
 $f? \in (u \cup e) \wedge$
 $motordir = up \wedge$
 $motor = on$
- D** $motordir' = motordir \wedge$
 $motor' = off \wedge$
 $f? \notin u' \wedge$
 $f? \notin e'$
- E** $motor = off \wedge$
 $motordir = up \wedge$
 $f? \notin (u \cup e)$
- F** $\nexists w \in (u \cup d \cup e) \bullet w > f? \Rightarrow motordir' = down \wedge$
 $\exists w \in (u \cup d \cup e) \bullet w > f? \Rightarrow motordir' = motordir \wedge$
 $motor' = on$
- G** $motor = on \wedge$
 $motordir = up \wedge$
 $f? < top \wedge$
 $f? \notin (u \cup e) \wedge$
 $\exists w \in (u \cup d \cup e) \bullet w > f?$
- H** $motor' = motor \wedge$
 $motordir' = motordir$
- I** $(\nexists w \in (u \cup d \cup e) \bullet w > f? \vee f? = top) \wedge$
 $motordir = up \wedge$
 $motor = on$
- J** $motordir' = up$
 $motor' = off$
- K** $(\nexists w \in (u \cup d \cup e) \bullet w > f? \vee f? = top) \wedge$
 $motordir = up \wedge$
 $motor = off$
- L** $motordir' = down \wedge$
 $motor' = on$
- M** $f? > ground \wedge$
 $f? \in (d \cup e) \wedge$
 $motordir = down \wedge$
 $motor = on$
- N** $motordir' = motordir \wedge$
 $motor' = off \wedge$
 $f? \notin d' \wedge$
 $f? \notin e'$
- O** $motor = off \wedge$
 $motordir = down \wedge$
 $f? \notin (d \cup e)$
- P** $((\nexists w \in (u \cup d \cup e) \bullet w < f? \wedge$
 $\exists w \in (u \cup d \cup e) \bullet w > f?)$
 $\Rightarrow motordir' = up)$
 $\vee$
 $motordir' = motordir) motor' = on$
- Q** $motor = on \wedge$
 $motordir = down \wedge$
 $ground < f? \wedge$
 $f? \notin (d \cup e)$
- R** $motor' = motor \wedge$
 $motordir' = motordir$
- S** $f? = ground \wedge$
 $motordir = down \wedge$
 $motor = on$
- T** $motordir' = motordir \wedge$
 $motor' = off$
- U** $f? = ground \wedge$
 $motordir = down \wedge$
 $motor = off$
- V** $motordir' = up \wedge$
 $motor' = motor$

```

BUTTON ( i : floor ) IS
  LITERALS
    any
    yes
    no
  REFERENCES
    SHOWNOREQ
    SHOWREQ
  INPUTS
    bb.i.press
    eb.any
    eb.i
  PROCESS BUTTON(i) IS BUTTON(i,no)
  PROCESS BUTTON(i,no) IS
    bb?i.press → SHOWNOREQ(i,yes)
    | eb?any → ⊥
  PROCESS BUTTON(i,yes) IS
    eb?i → SHOWNOREQ(i,no)
    | bb?i.press → BUTTON(i,yes)

```

Figure 5: $BUTTON(i)$ — button process at floor i

In the above firing and termination conditions, $motor = on$ indicates that the motor is operational, while $motor = off$ indicates that the motor is not moving the elevator. Thus, these two conditions are functions of the events $motor.start$ and $motor.stop$ in the RPT^+ specification of the elevator controller.

6.3 Formal Specifacaton of the SSD

We now address the derivation of a formal specification corresponding to the original System Specification Diagram. While the formal specification contains many more process definitions than are represented in the SSD, it should be pointed out than many of these are auxillary or subsidiary definitions to those in the SSD. This represents an attempt to structure the specification into realistically-sized modules, rather than present an unintelligible monolithic specification.

$BUTTON(i)$, as shown in Figure 5 begins with no outstanding requests ($BUTTON(i, no)$), and receives `press` events along the corresponding channel `bb.i`. In response it sends signals to the lighting subsystem on channel `l` (behaving as processes `SHOWNOREQ` and `SHOWREQ` as shown in Figure 6 and Figure 7) and behaves as a process with an outstanding request for service ($BUTTON(i, yes)$).

As requests are satisfied, the $BUTTON$ process receives notification of this from the elevator controller process along channel `eb`.

We may now define the various sets of buttons — `UP`, corresponding to buttons outside the elevator for requesting upwards travel, `DOWN` for requesting downwards travel, and `INTERNAL`, the set of buttons inside the elevator to request travel to a particular floor.

$$\begin{aligned}
 UP &= \parallel_{ground \leq i < top} i : u : BUTTON \\
 DOWN &= \parallel_{ground < i \leq top} i : d : BUTTON
 \end{aligned}$$

```

SHOWNOREQ ( i : floor; r : req ) IS
  LITERALS
    any
  REFERENCES
    SHOWREQ
  INPUTS
    bb.i.press
    eb.any
  OUTPUTS
    l.i.on
  PROCESS SHOWNOREQ(i,req) IS
    l!i.on → SHOWREQ(i,req)
    | bb?i.press → SHOWNOREQ(i,req)
    | eb?any → ⊥

```

Figure 6: $SHOWNOREQ(i, req)$ — button light with no outstanding requests at floor i

```

SHOWREQ ( i : floor; r : req ) IS
  LITERALS
    any
    request
  REFERENCES
    BUTTON
  INPUTS
    bb.i.press
    eb.any
  OUTPUTS
    be.i.request
  SHOWREQ(i,req) IS
    be!i.request → BUTTON(i,req)
    | bb?i.press → SHOWREQ(i,req)
    | eb?any → ⊥

```

Figure 7: $SHOWREQ(i, req)$ — button light with outstanding request at floor i

```

ELEVATOR() IS
  LITERALS
    up
    down
  VARIABLES
    x : { leave(m), arrive(m) } where ground ≤ n ≤ top
  REFERENCES
    MOVEELEV
    ⊥
  INPUTS
    ev.v where v ∈ {leave(m), arrive(m) | ground ≤ n ≤ top}

  PROCESS ELEVATOR() = ELEVATOR(ground,up)
  PROCESS ELEVATOR(i,up) = ev?x → (MOVEELEV(i,up)
                                if x = leave(i) else
                                ⊥)
  PROCESS ELEVATOR(i,down) = ev?x → (MOVEELEV(i,down)
                                if x = leave(i) else
                                ⊥)

```

Figure 8: ELEVATOR process.

```

INTERNAL = ||ground ≤ i ≤ top i:e:BUTTON
BUTTONS = UP || DOWN || INTERNAL

```

6.4 The Elevator and Controller

It transpires that the elevator itself is a very simple process, and the complexity is contained in the elevator controller. We now consider each of these processes.

The elevator simply outputs pairs of `leave(m)` and `arrive(n)` events on channel `ee`, reflecting changes in its real-world position. We will consider the effect this has on its state when we come to consider its Structure Diagram (and associated Z specification).

The `ELEVATOR` is defined as shown in Figure 8, starting at the ground floor, and moving upwards. The subordinate process `MOVEELEV` indicates the movement of the elevator, leaving floor `i` and eventually reaching floor `i+1` if it is moving in the direction up, or floor `i-1` if travelling in direction down.

`ELEVBODY`, and its subsidiary processes denotes the behaviour of the elevator as it travels between floors, outputting successive pairs of `leave(m)` and `arrive(m)` events on channel `ee`.

The controller performs most of the work in the system. It controls the motion of the elevator by issuing commands to the motor subsystem, receives requests from various button types and sends messages along the relevant communication channels to indicate that these have been satisfied, and finally receives acknowledgements from the buttons.

We are considering the more sophisticated controller described by Jackson. Initially the elevator is at the ground floor. The controller will raise the elevator to a higher floor when there are outstanding requests to visit a higher floor or for service at a higher floor.

```

MOVEELEV ( i : floor; dir : direction ) IS
  LITERALS
    up
    down
    any
  REFERENCES
    ELEVBODY
     $\perp$ 
  OUTPUTS
    ee.leave(i)
  INPUTS
    ev?any

  PROCESS MOVEELEV(i,up) = ee!leave(i)  $\rightarrow$  ELEVBODY(i,up)
                        | ev?any  $\rightarrow \perp$ 
  PROCESS MOVEELEV(i,down) = ee!leave(i)  $\rightarrow$  ELEVBODY(i,down)
                        | ev?any  $\rightarrow \perp$ 

```

Figure 9: MOVEELEV(*i*,*dir*) — elevator moving in direction *dir* from floor *i*.

```

ELEVBODY ( i : floor; dir : direction ) IS
  LITERALS
    up
    down
  VARIABLES
    x : { leave(m), arrive(m) } where ground  $\leq$  n  $\leq$  top
  REFERENCES
    COPY
     $\perp$ 
  INPUTS
    ev.v where v  $\in$  {leave(m), arrive(m) | ground  $\leq$  n  $\leq$  top}

  PROCESS ELEVBODY(i,up) = ev?x  $\rightarrow$  ( COPY(i,up,x)
                                if x = arrive(i+1) else
                                 $\perp$  )
  PROCESS ELEVBODY(i,down) = ev?x  $\rightarrow$  ( COPY(i,down,x)
                                if x = arrive(i-1) else
                                 $\perp$  )

```

Figure 10: ELEVBODY(*i*,*dir*) — body of elevator moving in direction *dir* from floor *i*.

COPY (*i* : floor; *dir* : direction; *x* : {arrive(*m*), leave(*n*)}) IS

LITERALS

up
down
any

REFERENCES

ELEVATOR

INPUTS

ev.any

OUTPUTS

ee.v where $v \in \{\text{leave}(m), \text{arrive}(m) \mid \text{ground} \leq n \leq \text{top}\}$

PROCESS COPY(*i*,up,*x*) = ee!*x* → (ELEVATOR(*i*+1,*x*)
if *i*+1 < top else
ELEVATOR(top,down)
| ev?any → ⊥

PROCESS COPY(*i*,down,*x*) = ee!*y* → (ELEVATOR(*i*-1,*x*)
if *i*-1 > ground else
ELEVATOR(ground,up)

Figure 11: COPY(*i*,*dir*,*x*) — repeater process with elevator travelling in direction *dir* from floor *i*, having engaged in event *x*.

```

ELCONTROL() IS
  REFERENCES
    STOPPEDAT
  PROCESS ELCONTROL() IS STOPPEDAT(ground, up, {}, {}, {}, [[]])

```

Figure 12: ELCONTROL — the elevator controller

It will continue to travel upwards while there are outstanding requests to visit or to service higher floors. On reaching the `top` floor, the controller will reverse the direction of travel and the elevator will travel downwards again. To avoid unnecessary upward journeys, the controller may reverse the direction of the elevator when there are no outstanding requests for a higher floor.

Similarly, on the downward journey, the controller may reverse the direction of the elevator if there are outstanding requests for service at, or to travel to, higher floors. However, if the elevator is travelling downwards and there are still requests outstanding at a lower floor, the direction of travel may not be reversed.

Remembering that we must consider that other requests can be made at any time, the behaviour of the controller ELCONTROL may be described appropriately in RPT^+ .

Initially the elevator starts at the `ground` floor with no outstanding requests for service, as shown in Figure 12.

While stopped at a floor, as shown in Figure 13, the controller will wait at the current floor, `f`, as long as there are additional requests to service that floor. This is caused by, for example, a passenger continually pressing the button corresponding to that floor, in which case it behaves as WAITREQ (as illustrated in Figure 14), or pressing the button for upwards travel when the elevator is travelling in the direction `up`, etc.), in which case it behaves as WAITSERV (as illustrated in Figure 15).

When there are no outstanding requests for service at the current floor, the elevator controller must continue to wait until all outstanding acknowledgements in `B` (a bag of requests which have been satisfied by the controller but for which acknowledgements sent along the relevant `be` channel are still outstanding) have been received; that is, it behaves as WAITACK, as illustrated in Figure 16. Once there are no outstanding requests or acknowledgements, the controller behaves as IDLE.

The behaviour of the elevator as defined by process IDLE, shown in Figure 17, varies depending on the direction of travel, `dir`, and the current floor, `f`: At the `ground` floor, the elevator controller services requests for that floor (behaves as process SERVICE, as illustrated in Figure 18) unless there are requests for service at, or to visit, other floors. In the latter case, the process behaves once again as STOPPEDAT, and prepares to move upwards. At the `top` floor, the elevator services that floor (which by the definition of SERVICE, Figure 18, will eventually result in downwards travel). At intermediate floors, with `dir` set to `up`, the controller will continue to service the current and higher floors as long as there are outstanding requests for service at, or to visit, higher floors; otherwise, it will behave as process REV (Figure 24), change direction and travel downwards again. With `dir` set to `down`, it will behave as REV, and travel upwards again, if there are outstanding requests at, or to visit, higher floors, but no outstanding requests at the current or lower floors. Otherwise, it will service the current floor, without changing direction.

Servicing a request, as illustrated in Figure 18, involves restarting the motor (by sending a `start` message along channel `m`), and subsequently behaving as GO, with the floor set to `f+1` if the elevator is travelling upwards, or `f-1` otherwise. Having left floor `f` (having engaged in the event `ee.leave(f)`), the elevator must arrive at the next floor in the direction of travel, i.e., behave as ARRIVING (Figure 20).

Having arrived at a floor, the process behaves as CHECK (Figure 21), and signals the motor to stop if there are requests to service that floor, in which case it subsequently behaves as TURN. This process (illustrated in Figure 22) enforces the algorithm to reduce unnecessary travel, by forcing the process to behave as REV (Figure 24) and changing direction if there are no more requests to be satisfied in the current direction of travel.

The entire System Specification Diagram may now be described as the parallel composition of the various RPT^+ pro-

```

STOPPEDAT ( i : floor ; dir : direction ; u, d, e : set of floor ;
           B : bag of floor ) IS

REFERENCES

    WAITREQ
    WAITSERV
    WAITACK
    IDLE

PROCESS STOPPEDAT(f,up,u,d,e,B) IS
    WAITREQ(f,up,u,d,e,B)
    if f ∈ u else
        WAITSERV(f,up,u,d,e,B)
    if f ∈ e else
        WAITACK(f,up,u,d,e,B)
        if f ∈ B else
            IDLE(f,up,u,d,e,B)

PROCESS STOPPEDAT(f,down,u,d,e,B) IS
    WAITREQ(f,down,u,d,e,B)
    if f ∈ d else
        WAITSERV(f,down,u,d,e,B)
    if f ∈ e else
        WAITACK(f,down,u,d,e,B)
        if f ∈ B else
            IDLE(f,down,u,d,e,B)

```

Figure 13: STOPPEDAT — the controller stopped at floor *i*, travelling in direction *dir*, and with *u* upwards requests, *d* downwards requests, *e* floor-visit requests, and *B* outstanding light requests.

```

WAITREQ(f : floor; dir : direction;
        u,d,e : set of floor; B : bag of floor ) IS

LITERALS

    any

VARIABLES

    m : floor

REFERENCES

    STOPPEDAT
     $\perp$ 

INPUTS

    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    ee.any

OUTPUTS

    u.eb.f
    d.eb.f

PROCESS WAITREQ(f,up,u,d,e,B) IS
    u.eb!f  $\rightarrow$  STOPPEDAT(f,up,u- $\{f\}$ ,d,e,B+[[f]])
    | u.be?n.request  $\rightarrow$  STOPPEDAT(f,up,u $\cup\{m\}$ ,d,e,B), m  $\diamond$  f
    | u.be?f.request  $\rightarrow$  STOPPEDAT(f,up,u,d,e,B-[[f]])
    | d.be?n.request  $\rightarrow$  STOPPEDAT(f,up,u,d $\cup\{m\}$ ,e,B)
    | e.be?n.request  $\rightarrow$  STOPPEDAT(f,up,u,d,e $\cup\{m\}$ ,B), m  $\diamond$  f
    | e.be?f.request  $\rightarrow$  STOPPEDAT(f,up,u,d,e,B-[[f]])
    | ee?any  $\rightarrow$   $\perp$ 

PROCESS WAITREQ(f,down,u,d,e,B) IS
    d.eb!f  $\rightarrow$  STOPPEDAT(f,down,u,d- $\{f\}$ ,e,B+[[f]])
    | d.be?n.request  $\rightarrow$  STOPPEDAT(f,down,u,d $\cup\{m\}$ ,e,B), m  $\diamond$  f
    | d.be?f.request  $\rightarrow$  STOPPEDAT(f,down,u,d,e,B-[[f]])
    | u.be?n.request  $\rightarrow$  STOPPEDAT(f,down,u $\cup\{m\}$ ,d,e,B)
    | e.be?n.request  $\rightarrow$  STOPPEDAT(f,down,u,d,e $\cup\{m\}$ ,B), m  $\diamond$  f
    | e.be?f.request  $\rightarrow$  STOPPEDAT(f,down,u,d,e,B-[[f]])
    | ee?any  $\rightarrow$   $\perp$ 

```

Figure 14: WAITREQ — the elevator controller at floor f , travelling up or down.

```

WAITSERV(f : floor; dir : direction; u,d,e : set of floor;
         B : bag of floor ) IS

VARIABLES
    m : floor

REFERENCES
    STOPPEDAT
    ⊥

INPUTS
    u.be.n.request where ground ≤ n ≤ top
    d.be.n.request where ground ≤ n ≤ top
    d.be.n.request where ground ≤ n ≤ top
    ee.any

OUTPUTS
    e.eb.f

PROCESS WAITSERV(f,dir,u,d,e,B) IS
    e.eb!f → STOPPEDAT(f,up,u,d,e-{f},B+[[f]])
    | u.be?n.request → STOPPEDAT(f,dir,u∪{m},d,e,B), m ◇ f
    | u.be?f.request → STOPPEDAT(f,dir,u,d,e,B-[[f]])
    | d.be?n.request → STOPPEDAT(f,dir,u,d∪{m},e,B)
    | e.be?n.request → STOPPEDAT(f,dir,u,d,e∪{m},B), m ◇ f
    | e.be?f.request → STOPPEDAT(f,dir,u,d,e,B-[[f]])
    | ee?any → ⊥

```

Figure 15: WAITSERV — elevator controller servicing requests at floor f , while travelling in direction dir .

```

WAITACK(f : floor; dir : direction; u,d,e : set of floor;
        B : bag of floor ) IS

VARIABLES
    m : floor

REFERENCES
    STOPPEDAT
    ⊥

INPUTS
    u.be.n.request where ground ≤ n ≤ top
    d.be.n.request where ground ≤ n ≤ top
    d.be.n.request where ground ≤ n ≤ top
    ee.any

PROCESS WAITACK(f,up,u,d,e,B) IS
    u.be?n.request → STOPPEDAT(f,up,u∪{m},d,e,B), m ◇ f
    | u.be?f.request → STOPPEDAT(f,up,u,d,e,B-[[f]])
    | d.be?n.request → STOPPEDAT(f,up,u,d∪{m},e,B)
    | e.be?n.request → STOPPEDAT(f,up,u,d,e∪{m},B), m ◇ f
    | e.be?f.request → STOPPEDAT(F,up,u,d,e,B-[[f]])

PROCESS WAITACK(f,down,u,d,e,B) IS
    d.be?n.request → STOPPEDAT(f,down,u,d∪{m},e,B), m ◇ f
    | d.be?f.request → STOPPEDAT(f,down,u,d,e,B-[[f]])
    | u.be?n.request → STOPPEDAT(f,up,u∪{m},d,e,B)
    | e.be?n.request → STOPPEDAT(f,up,u,d,e∪{m},B), m ◇ f
    | e.be?f.request → STOPPEDAT(F,up,u,d,e,B-[[f]])

```

Figure 16: WAITACK — controller waiting for acknowledgement of cleared request.

```

IDLE (f : floor; dir : direction; u,d,e : set of floor;
      B : bag of floor ) IS

REFERENCES

    SERVICE
    STOPPEDAT
    REV

PROCESS IDLE(ground,up,u,d,e,B) IS
    SERVICE(ground,up,u,d,e,B)
    if  $\exists w \in (u \cup d \cup e) \bullet w > f$  else
        STOPPEDAT(ground,up,u,d,e,B)

PROCESS IDLE(f,up,u,d,e,B) IS
    SERVICE(f,up,u,d,e,B)
    if  $\exists w \in (u \cup d \cup e) \bullet w > f$  else
        REV(f,down,u,d,e,B), ground < f < top

PROCESS IDLE(f,down,u,d,e,B) IS
    SERVICE(f,down,u,d,e,B)
    if  $\exists w \in (u \cup d \cup e) \bullet w > f$  else
        REV(f,up,u,d,e,B)
        if  $f \in u$  or  $\exists w \in (u \cup d \cup e) \bullet w > f$  else
            SERVICE(f,down,u,d,e,B), ground < f < top

PROCESS IDLE(top,down,u,d,e,B) IS
    SERVICE(top,down,u,d,e,B)

```

Figure 17: IDLE — elevator idles at floor f , while travelling in direction dir .

```

SERVICE (f : floor; dir : direction; u,d,e : set of floor;
          B : bag of floor ) IS

LITERALS

    start

VARIABLES

    m : floor

REFERENCES

    GO
    STOPPEDAT
     $\perp$ 

INPUTS

    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    ee.any

OUTPUTS

    m.start

PROCESS SERVICE(f,dir,u,d,e,B) IS
    m!start  $\rightarrow$  ( GO(f+1,up,u,d,e,B)
                  if dir = up else
                  GO(f-1,down,u,d,e,B) )
    | u.be?n.request  $\rightarrow$  STOPPEDAT(f,dir,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  STOPPEDAT(f,dir,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  STOPPEDAT(f,dir,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow$   $\perp$ 

```

Figure 18: SERVICE — elevator services requests at floor f while travelling in direction dir , and motor is started.

```

GO (f : floor; dir : direction; u,d,e : set of floor;
    B : bag of floor ) IS

VARIABLES
    m : floor

REFERENCES
    ARRIVING
    GO
     $\perp$ 

INPUTS
    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    ee.x where x  $\in$  {arrive(m), leave(m) | ground  $\leq$  n  $\leq$  top }

PROCESS GO(f,dir,u,d,e,B) IS
    ee?x  $\rightarrow$  ( ARRIVING(f,dir,u,d,e,B)
                if x = leave(f) else
                 $\perp$  )
    | u.be?n.request  $\rightarrow$  GO(f,dir,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  GO(f,dir,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  GO(f,dir,u,d,e $\cup$ {m},B)

```

Figure 19: GO — elevator has started travelling in direction `dir` at floor `f`.

```

ARRIVING (f : floor; dir : direction; u,d,e : set of floor;
          B : bag of floor ) IS

VARIABLES
    m : floor

REFERENCES
    ARRIVING
    CHECK
     $\perp$ 

INPUTS
    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    ee.x where x  $\in$  {arrive(m),leave(m) | ground  $\leq$  n  $\leq$  top}

PROCESS ARRIVING(f,dir,u,d,e,B) IS
    ee?x  $\rightarrow$  ( CHECK(f,dir,u,d,e,B)
               if x = arrive(f) else
                $\perp$  )
    | u.be?n.request  $\rightarrow$  ARRIVING(f,dir,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  ARRIVING(f,dir,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  ARRIVING(f,dir,u,d,e $\cup$ {m},B)

```

Figure 20: ARRIVING — elevator arriving at floor f , travelling in direction dir .

```

CHECK (f : floor; dir : direction; u,d,e : set of floor;
      B : bag of floor ) IS

LITERALS
    any
    stop

VARIABLES
    m : floor

REFERENCES
    TURN
    CHECK
     $\perp$ 

INPUTS
    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request  where ground  $\leq$  n  $\leq$  top
    ee.any

OUTPUTS
    m.stop

PROCESS CHECK(f,dir,u,d,e,B) IS
    m!stop  $\rightarrow$  TURN(f,dir,u,d,e,B)
    | u.be?n.request  $\rightarrow$  CHECK(f,dir,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  CHECK(f,dir,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  CHECK(f,dir,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow$   $\perp$ 

```

Figure 21: CHECK — check for outstanding requests and stop motor, if necessary.

cess specifications, i.e.

BUTTONS $\parallel$ ELEVATOR $\parallel$ ELCONTROL

6.5 Translating Structure Diagrams to TLZ

We can now consider the Structure Diagrams of each of the three main process and their translation to TLZ.

We assume a global clock, *time* of type *TIME*:

<i>Timer</i>	
<i>time</i> : <i>TIME</i>	
$0 \leq time$	

and three *TIME* constants $\delta BUTTON$, $\delta ELEVATOR$, and $\delta ELCONTROL$, which we will use to express timing constraints on the execution of operations (this is not addressed by Jackson in his example).

6.5.1 Buttons

BUTTONS is a very simple process, with just a single state component, *status*. *status* can have the values *pressed* and *released*, and so is best described as an enumerated type, *button-status*:

buttonstatus ::= *pressed* | *released*

We may then describe the state of the process as:

<i>ButtonState</i>	
<i>status</i> : <i>buttonstatus</i>	

The initial state of the process is where the button is released:

<i>InitButton</i>	
<i>ButtonState</i>	
<i>status</i> ' = <i>released</i>	

Clearly,

status : *buttonstatus* $\vdash$ *status*' = *released*

and the initial-state theorem is satisfied. In Z, the initial-state theorem is a proof obligation that must be discharged in order to demonstrate that the system can be instantiated, and that an initial state satisfying the system state invariant does indeed exist.

There are two simple operations to be considered, *press* and *release*. Although not addressed by Jackson, we will take this opportunity to add a simple timing requirement on the execution of the operations: that they take no longer than $\delta BUTTON$ time units to execute, for a reasonable value of $\delta BUTTON$.

<i>Press</i>	
$\Delta ButtonState$	
$\Delta Timer$	
<i>status</i> = <i>released</i>	
<i>status</i> ' = <i>pressed</i>	
<i>time</i> ' $\leq$ <i>time</i> + $\delta BUTTON$	

$Release$ $\Delta ButtonState$ $\Delta Timer$
$status = pressed$ $status' = released$ $time' \leq time + \delta BUTTON$

It can be clearly seen that **pre-Press** and **pre-Release** are mutually exclusive, and that the postcondition of one operation satisfies the precondition of the other. Hence, the *Press* and *Release* operations will alternate.

We may thus define *Button* as:

$$\begin{aligned}
 Button &\triangleq InitButton \\
 &\quad \wedge \square[Press \\
 &\quad \quad \vee Release]_{ButtonState} \\
 &\quad \wedge WF_{Timer, ButtonState}(Press, Release)
 \end{aligned}$$

6.5.2 The Elevator

We define the following enumerated types:

$$\begin{aligned}
 elevfloor &::= ground \dots up \\
 elevdir &::= up \mid down
 \end{aligned}$$

With these and the firing and termination conditions, we can determine the state of the elevator process as:

$ElevatorState$ $floor : elevfloor$ $dir : elevdir$

We define a valid initial state as:

$InitElevator$ $ElevatorState$
$floor' = ground$ $dir' = up$

The initial-state theorem is satisfied, since:

$$floor : elevfloor, dir : elevdir \vdash floor' = ground, dir' = up$$

We can now define the three required operations, with a constraint that movement between floors takes no longer than $\delta ELEVATOR$ time units to execute, for some reasonable value of $\delta ELEVATOR$.

$Start$ $\Delta ElevatorState$ $\Delta Timer$
$floor = ground$ $dir = up$ $floor' = ground + 1$ $dir' = dir$ $time' \leq time + \delta ELEVATOR$

$Arrive$ $\Delta ElevatorState$ $\Delta Timer$ $m? : elevfloor$
$(floor = m? + 1 \wedge dir = down)$ $\vee (floor = m? - 1 \wedge dir = up)$ $floor' = m$ $dir' = dir$ $time' \leq time + \delta ELEVATOR$

It can be clearly seen that *Start* is equivalent to *Arrive* with *ground* input for *m?*, i.e.

$$Start \equiv Arrive \wedge (m? = ground)$$

$Leave$ $\Delta ElevatorState$ $\Delta Timer$ $m? : elevfloor$
$floor = top \wedge dir = up \Rightarrow dir' = down$ $floor = ground \wedge dir = down \Rightarrow dir' = up$ $floor < top \wedge ground < floor \Rightarrow dir' = dir$ $floor' = floor$

Finally, we may define *Elevator* as the TLA specification:

$$\begin{aligned}
 Elevator &\triangleq InitElevator \\
 &\wedge \Box [Start \\
 &\quad \vee Arrive \\
 &\quad \vee Leave]_{ElevatorState} \\
 &\wedge WF_{Timer, ElevatorState}(Arrive, Leave)
 \end{aligned}$$

6.5.3 Lights and Doors

In the Hi-Ride Elevator example, Jackson [20] assumes that doors open when the elevator reaches a floor, and close when it leaves a floor. Therefore, the status of doors can be ignored in the specification. Similarly, he does not consider the status of lights (on buttons).

As a means of illustrating how easily we can extend the state of a process using Z schema composition, we now consider both the status of button lights and the elevator doors.

We define

$$lightstatus ::= on \mid off$$

$LightState$ $light : lightstatus$

We define two operations, *LightsOn* and *LightsOff*. The former turns the status of a light from *off* to *on* when the button is *pressed*, the latter turns the status of the light from *on* to *off* when the button is released.

<i>LightsOn</i>
$\Delta LightState$
$\Xi ButtonState$
$\Delta Timer$
$light = off$
$button = pressed$
$light' = on$
$time' \leq time + 1$

<i>LightsOff</i>
$\Delta LightState$
$\Xi ButtonState$
$\Delta Timer$
$light = on$
$button = released$
$light' = off$
$time' \leq time + 1$

These enable us to define new *LightPress* and *LightRelease* operations as:

$$LightPress \triangleq Press; LightsOn$$

$$LightRelease \triangleq Release; LightsOff$$

Button is then defined by the revised TLA specification:

$$Button \triangleq (InitButton \wedge [light : lightstatus \mid light = off])$$

$$\wedge \Box [LightPress$$

$$\vee LightRelease]_{(ButtonState \wedge LightState)}$$

$$\wedge WF_{Timer, (ButtonState \wedge LightState)}(LightPress, LightRelease)$$

Considering the status of doors is slightly more complex, in that we must ensure that doors are opened and closed while the elevator is stopped at a floor.

We define:

$$doorstatus ::= open \mid closed$$

$$boolean ::= true \mid false$$

<i>DoorState</i>
$door : doorstatus$

<i>FloorState</i>
$atfloor : boolean$

and define new operators *DoorOpen* and *DoorClose* as follows:

$DoorOpen$ $\Delta DoorState$ $\exists FloorState$ $\exists ElevatorState$ $\Delta Timer$
$atfloor = true$ $door = closed$ $door' = open$ $atfloor' = atfloor$ $time' \leq time + 5$

$DoorClose$ $\Delta DoorState$ $\exists FloorState$ $\exists ElevatorState$ $\Delta Timer$
$atfloor = true$ $door = open$ $door' = closed$ $atfloor' = atfloor$ $time' \leq time + 5$

We still, however, must extend the operations *Start*, *Arrive*, and *Leave* such that they modify the Boolean value *atfloor* to indicate whether or not the elevator is stopped at a floor. We can accomplish this, and capture the status of the doors by defining *DoorStart*, *DoorArrive*, and *DoorLeave* as follows:

$$\begin{aligned}
 DoorStart &\triangleq DoorClose; (Start \wedge [\Delta FloorStatus \mid atfloor = true \wedge at - floor' = false]) \\
 DoorArrive &\triangleq (Arrive \wedge [\Delta FloorStatus \mid atfloor = false \wedge at - floor' = true]); DoorOpen \\
 DoorLeave &\triangleq DoorClose; (Leave \wedge [\Delta FloorStatus \mid atfloor = true \wedge atfloor' = false])
 \end{aligned}$$

Notice that the relationship between *Start* and *Arrive*, noted earlier, no longer holds between *DoorStart* and *DoorArrive*. The *Elevator* may now be described by the revised TLA specification:

$$\begin{aligned}
 Elevator &\triangleq (InitElevator \wedge [DoorState \mid door = open] \wedge [FloorState \mid atfloor = true]) \\
 &\quad \wedge \Box [DoorStart \\
 &\quad \quad \vee DoorArrive \\
 &\quad \quad \vee DoorLeave]_{(ElevatorState \wedge FloorState \wedge DoorState)} \\
 &\quad \wedge WF_{Timer, (ElevatorState \wedge FloorState \wedge DoorState)}(DoorArrive, DoorLeave)
 \end{aligned}$$

6.5.4 The Elevator Controller

We define the following enumerated types:

$$\begin{aligned}
 motorstatus &::= on \mid off \\
 direction &::= up \mid down \\
 floor &::= ground \dots top
 \end{aligned}$$

and define the state of the elevator controller as:

ElcontrolState

motor : *motorstatus*
motordir : *direction*
u : $\mathbb{P} \text{ floor}$
d : $\mathbb{P} \text{ floor}$
e : $\mathbb{P} \text{ floor}$

where *u*, *d*, and *e* are sets denoting outstanding requests to travel upwards from a floor, downwards from a floor, or to visit a particular floor, respectively (as we saw already in the RPT^+ specification).

A valid initial state of the controller is

InitElcontrol

ElcontrolState

motor' = *off*
motordir' = *up*
u = { }
d = { }
e = { }

Again, we may derive the Z schema corresponding to each operation (leaf node of the Structure Diagram) systematically from the firing and termination conditions.

The elevator controller leaves the elevator at the ground floor as long as there are no outstanding requests, or while someone is pressing the button for the *ground* floor. Once there is a request for a floor other than the ground floor, the motor is started and the elevator will begin to move upwards:

Waitatground

$\Delta \text{ElcontrolState}$

ΔTimer

motor = *off*
motordir = *up*
 $(u \cup d \cup e) \subseteq \{\text{ground}\}$
motor' = *on*
motordir' = *motordir*
 $\text{ground} \notin (u' \cup e')$
 $(u' \cup d' \cup e') \neq \{\}$
 $\text{time}' \leq \text{time} + \delta \text{ELCONTROLLER}$

When the elevator reaches floor *f*?, while travelling upwards, any outstanding requests to visit floor *f*? or to travel upwards from *f*? must be satisfied and the motor stopped:

$Upstop$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$f? < top$ $f? \in (u \cup e)$ $motordir = up$ $motor = on$ $motordir' = motordir$ $motor' = off$ $f? \notin u'$ $f? \notin e'$ $time' \leq time + \delta ELCONTROLLER$

Notice that we write $f? \notin u'$ rather than $u' = u \setminus \{f?\}$. This means that after the operation has terminated, there are no outstanding requests for floor $f?$, but it does not prohibit the addition of requests for other floors. This is consistent with our RPT^+ specification which allowed for other requests at all times.

Once there are no more outstanding requests at floor $f?$, the motor may be restarted to allow the elevator to continue to satisfy requests at a higher floor. If there are no such requests outstanding, the direction of the motor may be reversed to allow firing of downward journeying:

$Upstart$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$motor = off$ $motordir = up$ $f? \notin (u \cup e)$ $\nexists w \in (u \cup d \cup e) \bullet w > f? \Rightarrow motordir' = down$ $\exists w \in (u \cup d \cup e) \bullet w > f? \Rightarrow motordir' = motordir$ $motor' = on$ $time' \leq time + \delta ELCONTROLLER$

Notice that there is no constraint on the final values of u , d and e , indicating that further requests may be made.

$Upstop$ and $Upstart$ occur in pairs, so we may write:

$$Upvisit \triangleq Upstop; UpStart$$

If there are no requests for floor $f?$ outstanding when the elevator passes $f?$ travelling upwards, but there are further requests for higher floors, the elevator may continue to pass upwards:

$Uppass$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$motor = on$ $motordir = up$ $f? < top$ $f? \notin (u \cup e)$ $\exists w \in (u \cup d \cup e) \bullet w > f?$ $motor' = motor$ $motordir' = motordir$

When the elevator reaches a floor above which there are no further outstanding requests, the controller should stop the motor and reverse the direction of the elevator:

$Stoptop$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$(\nexists w \in (u \cup d \cup e) \bullet w > f?$ $\vee f? = top)$ $motordir = up$ $motor = on$ $motordir' = motordir$ $motor' = off$

$Revtop$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$(\nexists w \in (u \cup d \cup e) \bullet w > f?$ $\vee f? = top)$ $motordir = up$ $motor = off$ $motordir' = down$ $motor' = on$

As before:

$$Upmostfloor \triangleq Stoptop; Revtop$$

The operations for downward motion are almost symmetric.

$Downstop$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$f? > ground$ $f? \in (d \cup e)$ $motordir = down$ $motor = on$ $motordir' = motordir$ $motor' = off$ $f? \notin d'$ $f? \notin e'time' \leq time + \delta ELCONTROLLER$

$Downstart$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$motor = off$ $motordir = down$ $f? \notin (d \cup e)$ $((\neg \exists w \in (u \cup d \cup e) \bullet w < f?$ $\wedge \exists w \in (u \cup d \cup e) \bullet w > f?)$ $\Rightarrow motordir' = up)$ $\forall motordir' = motordir)$ $motor' = on$ $time' \leq time + \delta ELCONTROLLER$

$Downvisit \triangleq Downstop; Downstart$

$Downpass$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$motor = on$ $motordir = down$ $ground < f?$ $f? \notin (d \cup e)$ $motor' = motor$ $motordir' = motordir$

$Stopground$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$f? = ground$ $motordir = down$ $motor = on$ $motordir' = motordir$ $motor' = off$
$Revground$ $\Delta ElcontrolState$ $\Delta Timer$ $f? : floor$
$f? = ground$ $motordir = down$ $motor = off$ $motordir' = up$ $motor' = motor$

$$Bottomfloor \triangleq Stopground; Revground$$

Hence, the controller may be described in TLA as:

$$\begin{aligned}
 Elcontrol &\triangleq InitElcontrol \\
 &\wedge \Box [Waitatground \\
 &\quad \vee Upvisit \\
 &\quad \vee Uppass \\
 &\quad \vee Upmostfloor \\
 &\quad \vee Downvisit \\
 &\quad \vee Downpass \\
 &\quad \vee Bottomfloor]_{ElcontrolState} \\
 &\wedge WF_{Timer, ElcontrolState}(Upvisit, Uppass, Downvisit, Downpass)
 \end{aligned}$$

6.6 Evaluation

We have presented a reworking of Jackson's 'Hi-Ride' elevator example closely following Jackson's own approach [20].

The use of RPT^+ together with TLZ overcomes the problems encountered when attempting to formalize the problem in CSP and other other CSP variants [1, 9, 31], by allowing us to address the ordering of events, while expressing timing constraints on state transitions, and permitting greater attention to state-based aspects of the system design.

Indeed, the latter has afforded a more effective treatment of changes in the state of the elevator doors and of the status of the button lights. The treatment of these aspects of the design has been greatly simplified through the use of TLZ, which is more appropriate than a process algebra for addressing such aspects. At the same time, our approach has resulted in a more formalized approach to these aspects than the more traditional ad hoc approach of JSD.

Like most other approaches to method integration [30], we find that the formal specification contains much more information than is contained in the graphical equivalent. As such, there is not a direct correspondance between the graphical view and the formal specification. This can be attributed to the fact that a formal specification enforces great rigour in the definition, and the designer is often forced to address aspects that may be left very loosely defined in

methods that are less formal. The graphical representation is, nonetheless, easier to present to non-specialists, and greatly helps in guiding the formalization.

Our method does offer some advantages over other approaches to method integration, however. As we delay consideration of data formats and explicit states until we address the Structure Diagram of each process, the ‘representational gap’, or discrepancy, between the two representations is reduced considerably. Also, the Structure Diagram (formally specified in TLZ) may be used to further guide the formal specification of the SSD, in that relevant events in the SSD are functions of the state changes denoted in the Structure Diagram.

ram.

Acknowledgements

This paper is drawn substantially from a forthcoming technical report, copies of which will be available from the author. In this paper, much of the language description and technical detail has been omitted, in favour of including more of the case study. I am grateful to an anonymous reviewer for making this suggestion.

I am grateful to Prof. C.A.R. Hoare for first motivating this work, to Dr. Mark Josephs and Dr. Larry Paulson for much guidance and helpful suggestions, and to Dr. Leslie Lamport for discussions on TLA, TLZ, and assistance with typesetting. Lastly, but by no means least, I would like to express my sincerest thanks to Prof. Tony Bryant and Dr. Lesley Semmens for their understanding and patience despite many missed deadlines with this paper.

References

- [1] J. Barnes (1990) *A Study of JSD as a Formal Design Method*, M.Sc. dissertation, Oxford University Computing Laboratory.
- [2] K. Bastiaens & J.M. Van Camphenhout (1993) A Visual Real-Time Programming Language. *Control Practice Engineering*, 1(1):59-63, February 1993.
- [3] J.R. Cameron (1986) An Overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222–240, February.
- [4] J.R. Cameron (1989) *JSP & JSD: The Jackson Approach to Software Development*. IEEE Computer Society Press, Los Alamitos.
- [5] C. Chedghey, S. Kearney & H.-J. Kugler (1987) Using VDM in an Object-Oriented Development Method for Ada Software. In *Proceedings 1st VDM-Europe Symposium: VDM — A Formal Method at Work*, Springer-Verlag Lecture Notes in Computer Science.
- [6] D.L. Dill (1989) *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, MIT Press.
- [7] C. Draper (1993) Practical Experiences of Z and SSADM. In *Proceedings ZUM’92: 7th Z User Meeting*, J.P. Bowen and J.E. Nicholls, editors. Springer-Verlag Workshops in Computing, 1993, pp 240-254.
- [8] R. Duke & G. Smith (1989) Temporal Logic and Z Specifications. *Australian Computer Journal*, 21(2):62–29, May 1989.
- [9] M.G. Hinchey (1992) JSD, RPT and the Design of Real-Time Applications. Oxford University Computing Laboratory, Programming Research Group, September.
- [10] M.G. Hinchey (1993a) The Design of Real-Time Applications. In *Proc. RTAW’93, 1st IEEE Workshop on Real-Time Applications*, New York City, 13-14 May 1993, IEEE Computer Society Press, pp 178-182.
- [11] M.G. Hinchey (1993b) A Formal Design Method for Real-Time Ada Software. In L. Collingbourne (editor) *Ada: Towards Maturity*, Computer and Communications Series 6, IOS Press, Amsterdam, 1993.

- [12] M.G. Hinchey (1994a) Visual Methods in Real-Time Programming. In W.A. Halang, editor, *Annual Review in Automatic Programming, Volume 18*, Elsevier, Oxford.
- [13] M.G. Hinchey (1994b) Formally Specifying Real-Time Ada Software. In *Proc. RTAW'94, 2nd IEEE Workshop on Real-Time Applications*, Washington, 21–22 July 1994, IEEE Computer Society Press.
- [14] M.G. Hinchey (1995a) Visual Methods in Real-Time Programming. *Control Engineering Practice*, July.
- [15] M.G. Hinchey (1995b) Structured and Formal Methods: An Investigative Framework. In D.A. Lamb, editor, *Studies of Software Design*, Lecture Notes in Computer Science, Springer-Verlag, New York.
- [16] M.G. Hinchey & S.A. Jarvis (1995) *Concurrent Systems: Formal Development in CSP*. McGraw-Hill International Series in Software Engineering, London & New York.
- [17] C.A.R. Hoare (1985) *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
- [18] M.E.C. Hull, A. Zarea-Aliabadi & D.A. Guthrie (1989) Object-Oriented Design, Jackson System Development (JSD) Specifications and Concurrency. *BCS Software Engineering Journal*, March.
- [19] M.A. Jackson (1975) *Principles of Program Design*, Academic Press, London.
- [20] M.A. Jackson (1983) *System Development*. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
- [21] M.B. Josephs, C.A.R. Hoare & H. Jifeng (1989) *A Theory of Asynchronous Processes*, Technical Report TR-6-89, Programming Research Group, Oxford University Computing Laboratory.
- [22] M.B. Josephs (1992) Receptive Process Theory. *Acta Informatica*, February 1992.
- [23] K Kronlöf, editor (1993) *Method Integration: Concept and Case Studies*. Wiley Series in Software Based Systems, London.
- [24] L. Lamport (1991) The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Palo Alto, CA 94301, 25 December 1991.
- [25] L. Lamport (1994) TLZ. Invited talk, 8th Z User Meeting, 29 – 30 June 1994, University of Cambridge.
- [26] LBMS Management Systems (1992) *An Introduction to JSD*, John Wiley and Sons, Chichester.
- [27] A. Renold (1988) Jackson System Development for Real-Time Systems. *Scientia Electra*, Vol. 34, No. 2, February.
- [28] J.J. Schwartz & J.J. Skubich (1993) Graphical Programming for Real-Time Systems. *Control Engineering Practice*, 1(1):43–49, February 1993.
- [29] J.J. Schwartz, J.J. Skubich, P. Szwed & M. Maranzana (1993) Real-Time Multitasking Design with a Graphical Tool. In *Proceedings RTAW'93, 1st IEEE Workshop on Real-Time Applications*, 13-14 May 1993, New York City, IEEE Computer Society Press, pp 8–11.
- [30] L.T. Semmens, R.B. France & T.W.G. Docker (1992) Integrating Structured Analysis and Formal Specification Techniques. *The Computer Journal*, 36(6):600-610, December 1992.
- [31] K.T. Sridhar & C.A.R. Hoare (1985) *JSD Expressed in CSP*, Technical Monograph PRG-51, Programming Research Group, Oxford University Computing Laboratory, July.
- [32] P. Zave & R.T. Yeh (1982) An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250-269, March 1982.

```

TURN (f : floor; dir : direction; u,d,e : set of floor;
      B : bag of floor ) IS

REFERENCES

  STOP
  REV
  GO

PROCESS TURN(f,up,u,d,e,B) IS
  STOP(f,up,u,d,e,B)
  if f ∈ (uUe) or f = top else
    REV(f,down,u,d,e,B)
    if f ∈ d and ¬∃ w ∈ (uUdUe) • w > j else
      GO(f+1,up,u,d,e,B)

PROCESS TURN(f,down,u,d,e,B) IS
  STOP(f,down,u,d,e,B)
  if f ∈ (dUe) or f = ground else
    REV(f,up,u,d,e,B)
    if f ∈ u and ¬∃ w ∈ (uUdUe) • w < j else
      GO(f-1,up,u,d,e,B)

```

Figure 22: TURN — reverse direction of travel.

```

STOP (f : floor; dir : direction; u,d,e : set of floor;
      B : bag of floor ) IS

LITERALS
    any
    stop

VARIABLES
    m : floor

REFERENCES
    STOPPEDAT
     $\perp$ 

INPUTS
    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request  where ground  $\leq$  n  $\leq$  top
    ee.any

OUTPUTS
    m.dir

PROCESS STOP(ground,down,u,d,e,B) IS
    m!dir  $\rightarrow$  STOPPEDAT(ground,up,u,d,e,B)
    | u.be?n.request  $\rightarrow$  STOP(ground,down,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  STOP(ground,down,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  STOP(ground,down,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow \perp$ 

PROCESS STOP(top,up,u,d,e,B) IS
    m!dir  $\rightarrow$  STOPPEDAT(top,down,u,d,e,B)
    | u.be?n.request  $\rightarrow$  STOP(top,up,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  STOP(top,up,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  STOP(top,up,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow \perp$ 

PROCESS STOP(f,dir,u,d,e,B) IS STOPPEDAT(f,dir,u,d,e,B),
                                ground < f < top

```

Figure 23: STOP — stop travel of elevator.

```

REV (f : floor; dir : direction; u,d,e : set of floor;
      B : bag of floor ) IS

LITERALS
    any
    stop

VARIABLES
    m : floor

REFERENCES
    STOP
    STOPPEDAT
     $\perp$ 

INPUTS
    u.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request where ground  $\leq$  n  $\leq$  top
    d.be.n.request  where ground  $\leq$  n  $\leq$  top
    ee.any

OUTPUTS
    m.up
    m.down

PROCESS REV(f,down,u,d,e,B) IS
    m!down  $\rightarrow$  STOPPEDAT(f,down,u,d,e,B)
    | u.be?n.request  $\rightarrow$  STOP(f,up,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  STOP(f,up,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  STOP(f,up,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow$   $\perp$ 

PROCESS REV(f,up,u,d,e,B) IS
    m!up  $\rightarrow$  STOPPEDAT(f,up,u,d,e,B)
    | u.be?n.request  $\rightarrow$  STOP(f,down,u $\cup$ {m},d,e,B)
    | d.be?n.request  $\rightarrow$  STOP(f,down,u,d $\cup$ {m},e,B)
    | e.be?n.request  $\rightarrow$  STOP(f,down,u,d,e $\cup$ {m},B)
    | ee?any  $\rightarrow$   $\perp$ 

```

Figure 24: REV — reverse direction of travel.