Specifying Verifying Concurrent Systems Using Z

Article · March 1997

DOI: 10.1007/3-540-58555-9_105 · Source: CiteSeer

Andy Evans
Xactium

Specifying & Verifying Concurrent Systems Using Z [*]

Andy S. Evans

Faculty of Information & Engineering Systems,
Leeds Metropolitan University,
Beckett Park, Leeds.

Abstract. At present, few guidelines exist for applying the Z notation to concurrent systems, while existing approaches to proving properties of Z specifications of concurrent systems place emphasis on the use of additional formalisms such as temporal logic. This paper proposes a practical and rigorous approach to specifying and verifying concurrent systems using the Z notation alone. Guidelines are given for specifying common features of concurrent systems in Z. A simple lift controller systems is then specified. Finally, a formal proof method based on a simple (weak) fairness rule is presented and is illustrated by means of proofs of safety and liveness properties of the lift controller system.

1 Introduction

The Z notation [1] has proved itself to be a powerful and usable notation for specifying and verifying software systems. Although it has been mainly used for the specification of sequential systems, Z has also been successfully used to specify aspects of concurrent systems. For example, Duke et al [2] and Fergus and Ince [3] have applied Z to the specification of concurrent communication protocols. Nevertheless, few guidelines exist at present for applying Z to concurrent systems. Furthermore, existing approaches to proving safety and liveness properties of concurrent Z specifications have placed emphasis on the use of additional formalisms such as temporal logic [2, 3], TLA [4], or the refinement of the specification into a process based notation such as CSP [5].

This paper proposes a practical and rigorous approach to the specification and verification of concurrent systems using Z. In this approach, Z is used to specify the state and operations of a concurrent system. Emphasis is placed on simplicity; notions of process, channel, and message communication are modelled using state variables. In order to model and reason with concurrency, a simple assumption is made about the way the operations of the concurrent system being modelled are selected and executed. Given this rule, a powerful but simple proof system can be defined in Z, which can be used to prove safety and liveness properties of Z specifications of concurrent systems.

The proof system presented here has been adapted from assertional programming and reasoning methods developed by Misra and Chandy [6], Lam and

Presented at Formal Methods Europe, Barcelona, Oct 24-28, 1994.

Shankar [7], and Tel [8]. Assertional methods are a widely used and powerful approach to specifying and verifying concurrent programs. Based on a state-transition model of concurrency they offer a surprisingly simple, yet rigorous approach to verifying properties of concurrent programs, such as safety and liveness.

In assertional methods, concurrent 'programs' are described in terms of program fragments or guarded commands. However, we believe that Z offers a far more abstract and elegant notation for this purpose. Also, the schema calculus and proof theory of Z appears to be very well suited to the 'operational' style of proof advocated by assertional methods. Thus, in this paper we aim to show that, contrary to popular belief, concurrent systems can be readily specified and verified using standard Z notation and proof theory alone.

This paper is organized as follows: section 2 describes the above approach in more detail and gives some general guidelines for specifying concurrent systems in Z. Section 3 provides an illustration of a Z specification of a lift controller system. Following this, a set of inference rules for proving safety and liveness properties based on standard Z proof theory is presented, along with some examples of their application to the lift controller system.

2 Specifying Concurrent Systems in Z

This section presents some guidelines for specifying concurrent systems in Z. It describes how a number of common features of concurrent and distributed systems, such as processes, communication and non-determinism can be specified in Z.

2.1 Modelling the State

We consider that an entire concurrent system can be described and is totally determined by its state [8]. To specify this in Z, a state schema is written in which the component variables of the concurrent system are declared. In addition, notions of process, channel, and message communication (common features of concurrent systems) are modelled using state variables. For example: processes can be modelled by a schema type containing the set of variables describing its state. The state of an asynchronous message channel can be modelled as a *bag* of the messages that are in transit in the channel. If the channel is required to pass messages sequentially, a *sequence* state variable is used. Time may also be modelled and reasoned with provided it is also included in the system state [7].

2.2 Operations as State Changes

The operations of a concurrent system describe the way that the state of the system may change. Messages may be sent or received and variables assigned new values. Operations are specified using operation schemas. These consist of a set of declarations (including the system state), a set of pre-conditions (which define the set of valid states that the operation may be applied to), and a set of post-conditions that defines the set of states that may be reached by the system if the operation changes.

Operations may be deterministic or non-deterministic. Deterministic operations have the property that the state after the operation is completely determined by the state before the operation and the operation itself. Non-deterministic operations have the property that the state before the operation determines a range of possible states after the operation. Although the processes of a concurrent system are likely to be deterministic, the introduction of non-determinism often helps in simplifying the system at the specification stage.

2.3 Concurrency

In order to model and reason with concurrency, a simple assumption is made about the way the operations are selected and executed. The system starts in a state that satisfies the initial conditions of the system. At each step in the execution of the system an operation is selected and executed *non-deterministically* according to the following (weak) *fairness* rule: any operation that is *continuously* enabled will eventually occur.

In this model, two or more operations can be thought of as being concurrent if they are *enabled* at the same time. An operation is said to be enabled if its preconditions are satisfied by the current state of the system. However, the execution of the system is still viewed as a sequence of discrete, instantaneous operations. If, at any time, a number of operations are enabled at the same time, then any of the operations may be the next to occur [2]. This feature nicely captures the fact that in real life, when several things are happening concurrently, the apparent ordering of the occurrence of events is not unique. Rather, any sequence of events may occur, reflecting the fact that the ordering of events is not determined by the model, but by the events that occur in the modelled system.

2.4 Communication

Communication is a fundamental aspect of all concurrent systems. Such communication generally falls into two types: asynchronous communication and synchronous communication. Both types of communication can be specified in Z as follows:

Asynchronous Communication

In asynchronous point-to-point message communication a process sends a message to another process by placing the message in a channel that links the two processes. A message can be placed on a channel provided there is some empty

[2] This is more generally termed as an 'interleaving model of concurrency'. It has the advantage of being able to capture all the same information as a truly concurrent model, but greatly simplifies reasoning.

space in the channel to hold the message. In a truly asynchronous (unbounded) channel, it is assumed that each channel has an unlimited amount of space so that any number of messages may be placed in the channel.

Asynchronous communication channels can be specified in Z by a sequence variable. For example, the following state schema declares the variable c to represent an asynchronous message channel, which links the two processes *SendMessage* and *ReceiveMessage*:

$[MSG]$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Asynchronous}} \\
\hline
c : \operatorname{seq} MSG \\
\hline
\end{array}$$

Operations to send and receive messages along this channel can be specified as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SendMessage}} \\
\hline
\Delta \mathit{Asynchronous} \\
m? : MSG \\
\hline
c' = c \frown \langle m? \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ReceiveMessage}} \\
\hline
\Delta \mathit{Asynchronous} \\
m! : MSG \\
\hline
c \neq \langle \rangle \\
c' = \mathit{tail}\ c \\
m! = \mathit{head}\ c \\
\hline
\end{array}$$

Synchronous Communication

In synchronous communication, for example as used in CSP [9], one process is the sender and the other is a receiver. Communication takes place only if the receiving process is waiting for the communication; this is termed a *rendezvous*. Synchronous communication can be modelled in Z as follows. Let the state variables s and r represent the (possibly empty) output and input channels of a sending and receiving process respectively:

$SMSG ::= msg\langle\langle MSG \rangle\rangle \mid empty$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Synchronous}} \\
\hline
s, r : SMSG \\
\hline
\end{array}$$

A rendezvous operation can now be specified, which ensures that the two processes may only communicate provided that the sender is waiting to send ($s \neq \mathit{empty}$) and the receiver is waiting to receive ($r = \mathit{empty}$):

$$\begin{array}{|l}
\hline \mathit{Rendezvous} \\
\Delta \mathit{Synchronous} \\
\hline
s \neq \mathit{empty} \\
r = \mathit{empty} \\
s' = \mathit{empty} \\
r' = s \\
\hline
\end{array}$$

3 Case Study - A Simple Lift Controller System.

A lift controller system [3] is required to control a single lift in a multi-storey building. Requests for the lift can either be made by customers waiting on different floors of the building, or by customers travelling in the lift. The task of the controller is to service these requests by moving the lift to the required floors. The lift must only service requests in the direction it is currently travelling in. When there are no longer any pending requests in this direction, it will change direction to service any requests in the opposite direction.

First, the basic types of the system are introduced:

$$\begin{array}{|l}
\mathit{number\_of\_floors} : \mathbb{N} \\
\hline
\mathit{number\_of\_floors} \geq 2
\end{array}$$

There should be at least two floors in the building for the lift to service.

$$\begin{array}{l}
\mathit{FLOOR\_NUMBER} == 1 \mathrel{..} \mathit{number\_of\_floors} \\
\mathit{DIRECTION} ::= \mathit{up} \mid \mathit{down} \\
\mathit{STATE} ::= \mathit{stopped} \mid \mathit{moving} \\
\mathit{DOOR} ::= \mathit{open} \mid \mathit{closed}
\end{array}$$

$$\mathit{REQUEST\_TYPE} ::= \mathit{up\_request} \mid \mathit{down\_request} \mid \mathit{passenger\_request}$$

FLOOR_NUMBER is a type representing the possible floors used by the lift. The direction of the lift can either be *up* or *down*. The lift is either *stopped* or *moving* and the lift door is either *open* or *closed*. A request is either a request for a lift to go up or down or a passenger request.

$$\begin{array}{|l}
\hline \mathit{Lift} \\
\mathit{position} : \mathit{FLOOR\_NUMBER} \\
\mathit{direction} : \mathit{DIRECTION} \\
\mathit{state} : \mathit{STATE} \\
\mathit{door} : \mathit{DOOR} \\
\hline
\end{array}$$

[3] This is a simplified version of Jackson's hi-rider elevator system [10].

The lift has a floor number, a direction, a state and a door status.

$$
\begin{array}{|l}
\hline
\textit{Request} \\
\textit{type} : \textit{REQUEST\_TYPE} \\
\textit{floor} : \textit{FLOOR\_NUMBER} \\
\hline
\end{array}
$$

A request identifies a type of request and the floor number requested.

$$
\begin{array}{|l}
\hline
\textit{Lift\_System} \\
\textit{lift} : \textit{Lift} \\
\textit{requests} : \mathbb{P}\,\textit{Request} \\
\hline
\end{array}
$$

The state of the lift system consists of a single lift and the set of requests that have been made and are pending for the lift.

$$
\begin{array}{|l}
\hline
\textit{Initialize\_Lift\_System} \\
\textit{Lift\_System}' \\
\hline
\textit{lift}'.\textit{position} = 1 \land \\
\textit{lift}'.\textit{direction} = \textit{up} \\
\textit{lift}'.\textit{state} = \textit{stopped} \\
\textit{lift}'.\textit{door} = \textit{open} \\
\textit{requests}' = \emptyset \\
\hline
\end{array}
$$

The lift is initially at the bottom floor of the building, the door is open and there are no requests.

$$
\begin{array}{|l}
\hline
\textit{MakeRequests} \\
\Delta \textit{Lift\_System} \\
r? : \mathbb{F}_1\,\textit{Request} \\
\hline
\textit{requests}' = \textit{requests} \cup r? \\
\textit{lift}' = \textit{lift} \\
\hline
\end{array}
$$

At any moment in time a number of requests may be made for the lift. This is specified by the operation *MakeRequests*, which adds the new requests to the set of requests currently pending in the lift system. It is assumed that this operation is deterministically selected by the users of the lift. Note that this operation nicely illustrates how two independent components of the concurrent system can communicate with each other via the shared variable *requests*.

$$
\begin{array}{l}
\textit{Move\_Lift\_Up} \\
\Delta \textit{Lift\_System} \\
\hline
\textit{lift.door} = \textit{closed} \\
\neg\, (\exists\, r : \textit{requests} \bullet r.\textit{floor} = \textit{lift.position}) \\
\exists\, r : \textit{requests} \bullet \\
\qquad r.\textit{floor} > \textit{lift.position} \land \\
\qquad (r.\textit{type} = \textit{passenger\_request} \lor r.\textit{type} = \textit{up\_request}) \\
(\textit{lift.direction} = \textit{up} \lor \\
(\textit{lift.direction} = \textit{down} \land \\
\neg\, (\exists\, r : \textit{requests} \bullet \\
\qquad r.\textit{floor} < \textit{lift.position} \land \\
\qquad (r.\textit{type} = \textit{passenger\_request} \lor r.\textit{type} = \textit{down\_request})))) \\
\\
\textit{lift}'.\textit{position} = \textit{lift.position} + 1 \\
\textit{lift}'.\textit{direction} = \textit{up} \\
\textit{lift}'.\textit{state} = \textit{moving} \\
\textit{lift}'.\textit{door} = \textit{closed} \\
\textit{requests}' = \textit{requests}
\end{array}
$$

Provided there are pending requests for the lift above its current position and the lift is currently travelling upwards, or the lift is travelling downwards and there are no remaining downwards requests, then this operation will move the lift up by one floor. This operation will only proceed if there are no requests to be serviced at the current floor and the lift door is closed.

$$
\begin{array}{l}
\textit{Open\_Door} \\
\Delta \textit{Lift\_System} \\
\hline
\textit{lift.door} = \textit{closed} \\
\exists\, r : \textit{requests} \bullet r.\textit{floor} = \textit{lift.position} \\
\\
\textit{lift}'.\textit{position} = \textit{lift.position} \\
\textit{lift}'.\textit{direction} = \textit{lift.direction} \\
\textit{lift}'.\textit{state} = \textit{stopped} \\
\textit{lift}'.\textit{door} = \textit{open} \\
\textit{requests}' = \textit{requests}
\end{array}
$$

This operation opens the lift door when there are requests to be serviced and the lift door is closed.

$$
\begin{array}{l}
\textit{Close\_Door} \\
\hline
\Delta \textit{Lift\_System} \\
\hline
\textit{lift.door} = \textit{open} \\
\textit{lift.state} = \textit{stopped} \\
\exists\, r : \textit{requests} \bullet r.\textit{floor} \neq \textit{lift.position} \\
\\
\textit{requests}' = \textit{requests} \setminus \\
\quad \{ r : \textit{requests} \mid \\
\qquad r.\textit{floor} = \textit{lift.position} \} \\
\textit{lift}'.\textit{position} = \textit{lift.position} \\
\textit{lift}'.\textit{direction} = \textit{lift.direction} \\
\textit{lift}'.\textit{state} = \textit{moving} \\
\textit{lift}'.\textit{door} = \textit{closed} \\
\hline
\end{array}
$$

This operation closes the lift door if there are requests at other floors that need servicing and the lift door is open. Existing request that have been serviced by the lift are now removed from *requests*.

$$
\begin{array}{l}
\textit{Move\_Lift\_Down} \\
\hline
\Delta \textit{Lift\_System} \\
\hline
\textit{lift.door} = \textit{closed} \\
\neg\, (\exists\, r : \textit{requests} \bullet r.\textit{floor} = \textit{lift.position}) \\
\exists\, r : \textit{requests} \bullet \\
\quad r.\textit{floor} < \textit{lift.position} \land \\
\quad (r.\textit{type} = \textit{passenger\_request} \lor r.\textit{type} = \textit{down\_request}) \\
(\textit{lift.direction} = \textit{down} \lor \\
(\textit{lift.direction} = \textit{up} \land \\
\neg\, (\exists\, r : \textit{requests} \bullet \\
\quad r.\textit{floor} > \textit{lift.position} \land \\
\quad (r.\textit{type} = \textit{passenger\_request} \lor r.\textit{type} = \textit{up\_request})))) \\
\\
\textit{lift}'.\textit{position} = \textit{lift.position} - 1 \\
\textit{lift}'.\textit{direction} = \textit{down} \\
\textit{lift}'.\textit{state} = \textit{moving} \\
\textit{lift}'.\textit{door} = \textit{closed} \\
\textit{requests}' = \textit{requests} \\
\hline
\end{array}
$$

Provided there are pending requests for the lift below its current position and the lift is currently travelling downwards, or the lift is travelling upwards and there are no remaining upwards requests, then this operation will move the lift down by one floor. This operation will only proceed if there are no requests to be serviced on the current floor and the lift door is closed.

4 Proving Safety and Liveness Properties

This section presents a logic for the verification of concurrent Z specifications, which has been adapted from the assertional proof methods of Misra and Chandy [6], Shankar and Lam [7] and Tel [8]. The logic is based on proofs of the form $DOp \vdash P \Rightarrow Q'$. This states that if a deterministic schema operation [4], DOp, is applied to an operation in which predicate P holds, then predicate Q' will hold after the application of DOp [5]. Notice that Q' is dashed to represent the truth of the predicate Q in the post state of the operation.

An essential part of the logic is the fairness assumption that continuously enabled operations will eventually be executed. Safety and liveness properties are proved simply by showing that some or all of the operations in the system will result in the desired properties becoming true. As a simple example, suppose that a concurrent system has a deterministic operation, Op for which $Op \vdash true \Rightarrow Q'$ holds. It can be claimed at any point during the execution of the system that Q' will eventually hold. This is because operation Op will be executed eventually – from the fairness rule – and thereby establish Q'.

Most properties one may wish to prove of a concurrent system can be expressed using proofs of this form. However, it it cumbersome to do so all the time. Instead, a number of commonly used inference rules can be defined in terms of these proofs. These are presented below.

4.1 Proving Safety Properties

Traditionally, there are two major classes of properties of concurrent systems that one may wish to prove - *safety* (invariance) properties and *liveness* (progress) properties (a comprehensive introduction can be found in [11]).

Informally, the safety properties of a system mean that the system does nothing bad, for example engage in prohibited actions or sequences of actions. Formally, a safety property is expressed in terms of an *invariant* - a property that must always hold during the 'execution' of the system. Note that proving an invariant is an important step in the proof of liveness properties.

4.2 Safety Properties

Safety properties are expressed by the inference rule **unless** (first introduced by Misra and Chandy [6]), which is interpreted in Z as follows. For any specification, S, where $Op_1 \mathinner{..} Op_n$ represent all the possible deterministic operations of S, then:

[4] Although the proof theory described in this paper is based on deterministic operations, non-deterministic operations can be readily included within the theory by viewing them as describing an abstract class of deterministic operations. It is thus simply a case of demonstrating that all or a number of the possible non-deterministic choices satisfy the required inference rule.

[5] A possibly more precise way of stating this would be $DOp \vdash P(s) \Rightarrow Q(s')$, where s represents the undashed (pre-) state of DOp and s' represents the dashed (post-) state of DOp.

$$\frac{Op_1 \vdash P \land \neg\, Q \Rightarrow P' \lor Q' \,..\, Op_n \vdash P \land \neg\, Q \Rightarrow P' \lor Q'}{P\ \mathbf{unless}\ Q}$$

This states that provided the system is in a state that satisfies P and $\neg\, Q$, then after the application of any of the enabled operations in the system either P will remain true or Q will become true. Examples of safety properties that can be expressed using **unless** are that a message is in a channel unless it has been received, and that a philosopher remains hungry unless they are eating.

Two more important concepts are also defined using **unless**. For a given specification:

$$P\ \mathbf{unless}\ \mathit{false} \Leftrightarrow P\ \mathbf{is\ stable}$$

$$(\mathit{initialisation\ schema} \vdash Q') \land Q\ \mathbf{is\ stable} \Leftrightarrow Q\ \mathbf{is\ invariant}$$

A stable predicate remains true once it has become true, while an invariant is always true during the lifetime of the system.

Example 1

A important(!) safety property of the lift controller system is that when the lift door is open the lift cannot move. This can be expressed as:

$$(\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped})\ \mathbf{is\ invariant}$$

Expanding (from the definition of **is invariant**) gives:

$$\begin{array}{l}
\mathit{Initialize\_Lift\_System} \vdash (\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped}) \land \\
Op_1 \vdash \\
\quad (\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped}) \Rightarrow \\
\quad (\mathit{lift'.door} = \mathit{open} \Rightarrow \mathit{lift'.state} = \mathit{stopped}) \\
.. \\
Op_n \vdash \\
\quad (\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped}) \Rightarrow \\
\quad (\mathit{lift'.door} = \mathit{open} \Rightarrow \mathit{lift'.state} = \mathit{stopped})
\end{array}$$

where $Op_1 \,..\, Op_n$ represent all the deterministic operations of the Lift Controller System.

Proof

Clearly, *Initialize_Lift_System* preserves the property as initially both the lift doors are open and the lift is stationary.

Taking the conclusion of the operation proofs and simplifying as follow:

$$(\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped}) \Rightarrow$$
$$(\mathit{lift'.door} = \mathit{open} \Rightarrow \mathit{lift'.state} = \mathit{stopped}) \qquad \text{[conclusion]}$$

$$\neg\,(\mathit{lift.door} = \mathit{open} \Rightarrow \mathit{lift.state} = \mathit{stopped}) \vee$$
$$(\mathit{lift'.door} = \mathit{open} \Rightarrow \mathit{lift'.state} = \mathit{stopped}) \qquad \text{[law: } P \Rightarrow Q \Leftrightarrow \neg\, P \vee Q\text{]}$$

$$(\mathit{lift.door} = \mathit{open} \wedge \mathit{lift.state} = \mathit{moving}) \vee$$
$$(\mathit{lift'.door} = \mathit{open} \Rightarrow \mathit{lift'.state} = \mathit{stopped}) \qquad \text{[law: } \neg\,(P \Rightarrow Q) \Leftrightarrow P \wedge \neg\, Q\text{]}$$

$$(\mathit{lift.door} = \mathit{open} \wedge \mathit{lift.state} = \mathit{moving}) \vee \qquad \text{[law: } P \Rightarrow Q \Leftrightarrow \neg\, P \vee Q\text{]}$$
$$(\mathit{lift'.door} = \mathit{closed} \vee \mathit{lift'.state} = \mathit{stopped}) \qquad [\dagger 1]$$

Clearly, we need only show that all the operations of the system satisfy the second part of the disjunction in †1, as the initial condition does allow the lift door to be open while the lift is moving. This property can be shown to hold for each of the operations in the system as follows:

Move_Lift_Up - Satisfies †1 as the operation results in the doors being closed. A similar argument applies to *Move_Lift_Down*.

Open_Lift_Door - Satisfies †1 as the operation results in the lift being stopped.

Close_Lift_Door - Satisfies †1 as the operation results in the lift doors being closed.

In general, when carrying out a proof of **unless**, it suffices to show that if $P \wedge \neg\, Q$ is true in the pre-state of the operation then $P' \vee Q'$ is true in the post-state (rather than proving all cases of the implication as above). If $P \wedge \neg\, Q$ is not true, then clearly the operation will not be enabled and cannot execute (from the fairness rule), thereby preserving $P \wedge \neg\, Q$.

4.3 Proving Liveness Properties

If P and Q are predicates, then the liveness property $P \leadsto Q$ is defined to be true if, whenever a state is reached in which P is true, then eventually a state will be reached in which Q is true. Most liveness properties that one wishes to prove about systems are expressible in this form. For example, termination is expressed by letting P assert that the system is in its starting state and letting Q assert that the system has terminated. An essential part of proving liveness properties is to prove a related safety property. This is to show that nothing bad can happen before demonstrating that something good does indeed happen.

Two logical inference rules, **ensures** and **leads-to** (also written as $\leadsto$) are used to prove progress properties:

4.4 Ensures

Briefly, P **ensures** Q states that P remains true as long as Q is not true (i.e., P **unless** Q) and that there is a continuously enabled operation in the specification whose application results in Q becoming true, starting in any state that satisfies $P \land \neg Q$.

P **ensures** Q is defined as follows. For any specification S, where Op is a deterministic operation in S then:

$$\frac{P\,\mathbf{unless}\,Q \qquad P \land \neg Q\,\mathbf{enables}\,Op \qquad Op \vdash P \land \neg Q \Rightarrow Q'}{P\,\mathbf{ensures}\,Q}$$

The inference rule **enables** is used to state that Op is enabled by $P \land \neg Q$. It is defined as follows:

$$\frac{\mathbf{decl}\,Op \vdash P \Rightarrow \mathbf{pre}\,Op}{P\,\mathbf{enables}\,Op}$$

Here, **decl** is used to introduce just the declaration part of Op as the hypothesis of the proof that the predicate P implies the pre-condition of Op.

Example 2

Show that the lift door will open when the lift arrives at a floor with a request pending:

$(\exists\, r : requests \bullet r.floor = lift.position) \land lift.door = closed$
ensures
$lift'.door = open$

Expanding gives:

$((\exists\, r : requests \bullet r.floor = lift.position) \land lift.door = closed$
unless
$lift'.door = open) \land$
$Op \vdash$
$\quad(\exists\, r : requests \bullet r.floor = lift.position) \land$
$\quad lift.door = closed$
$\quad\Rightarrow$
$\quad lift'.door = open)$

The first (**unless**) statement is straightforward to prove: All progress operations (except the operation *Open_Lift_Door*) are not satisfied by the predicate $(\exists\, r : requests \bullet r.floor = lift.position) \land lift.door = closed$, and therefore result in the operation being disabled and ignored. The second statement states that there exists an operation in the system that will open the lift door when the lift arrives at a floor with a request pending. Clearly, this is satisfied by the operation $Open\_Lift\_Door$ as $(\exists\, r : requests \bullet r.floor = lift.position) \wedge lift.door = closed$ is its pre-condition and results in the lift door being opened.

4.5 Leads-to ($\leadsto$)

Almost all progress properties of our specifications are stated using the **leads-to** operator ($\leadsto$). A given specification has the property $P \leadsto Q$ if and only if it can be derived by a finite number of applications of the following rules:

$$\frac{P \textbf{ ensures } Q}{P \leadsto Q}$$

$$\frac{P \leadsto Q, Q \leadsto R}{P \leadsto R}$$

Thus, complicated liveness properties can be proved by splitting the proof up into a number of linked **ensures** relations and proving each one separately.

Induction

$$\frac{\forall\, n : n \in W \bullet P \wedge N = n \leadsto (P \wedge N < n) \vee Q}{P \leadsto Q}$$

Induction proofs are used to prove **leads-to** properties where an action, or actions, cause some type of incremental state change, e.g incrementing a counter, and it is required to prove that a number of increments will eventually result in a certain value. The above rule is adopted from [6]. Let W be some well ordered set, for example a sequence of numbers, and N be a well founded system state number, for example another counter. The meaning of the induction rule is that from any state in which P holds, the system will eventually reach a state in which Q holds, or it reaches a state in which P holds and the value of N is lower. Since N cannot decrease forever, eventually Q must become true.

Example 3

$$\begin{array}{l}
\exists_1\, r : requests \bullet \\
\quad r.position > lift.position \wedge \\
\quad lift.direction = up \wedge lift.state = moving \wedge \\
\quad lift.door = closed \\
\quad \leadsto \\
\quad lift.position = r.floor
\end{array}$$

That is, if there exists a single request for a lift on a floor above the current position of the lift, eventually the lift will move to that floor.

Substituting the Induction rule, with n representing a number from the well ordered set of floor_numbers, and the function N represented by the position of the lift the following is obtained:

$$\begin{array}{l}
\exists_1\, r : requests \bullet \forall\, n : 1 \mathrel{..} number\_of\_floors \bullet \\
\quad r.position > lift.position \wedge \\
\quad lift.direction = up \wedge lift.state = moving \wedge \\
\quad lift.door = closed \wedge r.position = m \\
\quad \textbf{ensures} \\
\quad lift'.position = r.floor \vee \\
\quad lift'.position = n + 1
\end{array}$$

This states that there exists an operation in the system, which when there exists a single request for a lift on a floor above the current position of the lift, will result in the lift moving up by 1 or moving to the floor of the request. Clearly, because the number of floors in the building is finite, eventually the lift will reach the same floor as the request. The operation that guarantees the progress of the lift upwards is *Move_Lift_Up* and it can be readily shown that this operation satisfies the above rule.

5 Conclusion

This paper proposes a practical approach to the specification and verification of concurrent systems using Z. Concurrency is modelled and reasoned with by making a simple fairness assumption about the way the operations of the concurrent system being specified are executed. Inference rules for proving safety and liveness properties are presented and applied to prove properties of a simple lift controller system.

The main aim of this paper is to show that Z is far more suitable for specifying and verifying concurrent systems than was previously thought. The author believes that a wide variety of concurrent systems could be specified and verified using this approach, especially as the operational style of proof method presented here should scale up well to large systems. In particular, the following areas of application of this work would appear to be very promising: (1) formalisation and proof of the informal graphical techniques used in real-time systems analysis; (2) distributed system specification, and (3) the specification and verification of concurrent object-oriented systems. The author has also used the approach to integrate Z with Coloured Petri Nets, as a means to both visualising concurrent Z specifications and improving on existing proof techniques for Petri Nets [12].

Acknowledgements

The author would especially like to thank Mike Hinchey, Tony Bryant and Sheila for their many helpful comments and suggestions.

References

1. Spivey J.M., The Z Notation (2nd Edition), Prentice Hall, 1993.
2. Duke R. et al., Protocol Specification and Verification Using Z, Protocol Specification, Testing and Verification VIII, North Holland 1988, p 33-46.
3. Fergus E. and Ince D., Z Specifications and Modal Logic, Proceedings of Software Engineering 90, Brighton, Ed. Patrick Hall, Cambridge University Press, July 1990.
4. Lamport, L., 'TLZ', Proceedings of the 8th Z Users Meeting, Cambridge, Springer Verlag, 1994.
5. Woodcock J.C.P and Morgan C., Refinement of State-Based Concurrent Systems, Procs. of VDM 90, Springer Verlag, p 341-351, 1990.
6. Chandy K.M and Misra J., Parallel Program Design, Addison Wesley, 1988.
7. Shankar A.U and Lam.S.S., Time Dependent Distributed Systems: proving safety, liveness and real-time properties, Distributed Computing 2, p 61-79, Springer Verlag, 1987.
8. Tel G., Topics In Distributed Algorithms, Cambridge University Press, Chapter 3, 1991.
9. Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall, 1985.
10. Jackson, M.A., System Development, Prentice Hall International Series in Computer Science, 1983.
11. Lamport L., A Simple Approach to Specifying Concurrent Systems, CACM, 32, p 32-45, 1989.
12. Evans, A.S., Visualising Concurrent Z Specifications, Proceedings of the 8th Z Users Meeting, Springer Verlag, Cambridge, June 1994.

This article was processed using the LaTeX macro package with LLNCS style