15th Australian Computer
Science Conference
Jan 1992

Specifying Concurrent Systems Using Object-Z

Graeme Smith and Roger Duke

Software Verification Research Centre
Department of Computer Science
University of Queensland

ABSTRACT

Although concurrency is essentially an implementation issue, it is often advantageous to specify a system in terms of its concurrent components. In this paper we examine the specification of concurrent systems using Object-Z. Object-Z is an extension to the formal specification language Z which supports an object-oriented specification style. The use of Object-Z is illustrated through two case studies: Conway's game of Life and a simple card game. We use the former example to compare Object-Z with the process-based language CSP and the latter to discuss the issue of specifying fairness in concurrent systems.

1 Introduction

Formal specification is concerned with system functionality, i.e. what a system can do without regard for how it does it. As is well understood, system functionality can be captured at various levels of abstraction. At the top level, the interface between the system and the environment is specified giving an external (user) model of the system. At lower levels, internal aspects concerned with the design and implementation of the system may also be incorporated into the model. A hierarchy of views is common in the specification of communication systems where specifications are given of the system's service as well as its underlying protocol.

Concurrency is essentially a design and implementation issue: a system may be designed as a composition of distinct components implemented concurrently and synchronising on particular events. Communication systems, for example, being distributed in nature are implemented concurrently. Other systems may be implemented concurrently for reasons such as efficiency. Consequently, it is important for any formal development technique to be able to accommodate concurrency. Indeed, for many systems, even functionality at the highest level is best specified in terms of the underlying system components and their interactions (e.g. see Duke *et al.*[6]). That is, clarity of a specification can often be enhanced by structuring the specification so as to reflect the concurrency of the system itself.

State-based specification languages such as Z[7,17,18] and VDM[10] are ideal for specifying top-level functionality. For example, consider a simple buffering system that can store up to two items. A top-level functional specification can readily be given using Z as follows. This specification is generic in that the type, T, of the items is not specified.

$$\begin{array}{|l}
\multicolumn{1}{l}{System[T]} \\ \hline
s : \mathrm{seq}\ T \\ \hline
\# s \leqslant 2 \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Init[T]} \\ \hline
System[T] \\ \hline
s = \langle\, \rangle \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{In[T]} \\ \hline
\Delta System[T] \\
in? : T \\ \hline
\# s < 2 \\
s' = s \frown \langle in? \rangle \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Out[T]} \\ \hline
\Delta System[T] \\
out! : T \\ \hline
s \neq \langle\, \rangle \\
s = \langle out! \rangle \frown s' \\ \hline
\end{array}$$

This specification (deliberately) abstracts away from implementation detail and gives no indication of system design: decisions such as whether this system should incorporate concurrency need to be taken independently. Suppose however that, for reasons of no concern to us here, the decision was taken to design this system as a concurrent composition of two one-place buffers as illustrated in Figure 1.

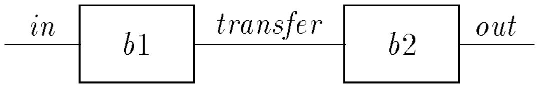

Figure 1: Simple buffer system

The one-place buffers $b1$ and $b2$ are assumed to be initially empty. The event *in* places an item from the system's environment into buffer $b1$, the event *transfer* transfers an item from buffer $b1$ to buffer $b2$ and the event *out* outputs an item from buffer $b2$ to the system's environment. Events *in* and *out* are restricted to the components $b1$ and $b2$ respectively, whereas the event *transfer* is performed by $b1$ and $b2$ in synchronisation.

Several specification languages have been designed specifically to capture the concurrency implicit in systems like the simple buffer of Figure 1. Process-based specification languages such as CSP[8] and CCS[15], for example, model concurrent systems as a collection of processes which interact via a 'handshake' mechanism. For example, the buffer system of Figure 1 could be modelled in CSP as follows.

$$\begin{array}{l}
B1 = in?x \rightarrow transfer!x \rightarrow B1 \\
B2 = transfer?y \rightarrow out!y \rightarrow B2 \\
SYSTEM = B1 \mid\mid B2
\end{array}$$

The concurrency operator '||' is used to construct the process *SYSTEM* which is composed of the component processes $B1$ and $B2$. Any trace of the process *SYSTEM* is an interleaving of traces of $B1$ and $B2$ subject to the restriction that synchronisation occurs on common-named events.

Despite the advantages of process-based languages when specifying concurrent systems in terms of their components, they are not ideally suited to the specification of higher-level functionality.

It is the aim of this paper to show that the state-based specification language Object-Z[3,5] is well suited for modelling concurrent systems at both levels of abstraction: in terms of their top-level functionality and in terms of their implementational structure. In Section 2 we introduce the Object-Z notation through a specification of the buffer system of Figure 1. In Section 3 we use Object-Z to specify Conway's game of Life[2] both functionally and in terms of a lower-level concurrent design. We compare this concurrent specification with one in CSP. In Section 4 we specify a simple card game consisting of a number of players. Each player is modelled as an object and the specification of the game involves specifying how the players interact. The game is based on a case study proposed by ISO[9] for the evaluation of specification languages for modelling open distributed systems. The card game is an example of a system whose high-level functionality is most readily modelled in terms of the system's concurrent components (i.e. the players). This system also raises the issue of fairness in concurrent systems[12,13]; we consider this issue and show how it may be handled elegantly in Object-Z.

2 Object-Z

Object-Z is an extension of Z which includes a special *class* construct to encapsulate a state schema with the operations which may effect that state and is designed to support an object-oriented[14] specification style. A fundamental idea of object orientation is that a system can be thought of as a collection of distinct, interacting components called objects. Hence, there is a strong relationship between concurrent and object-oriented systems. This relationship has been recognised by many researchers involved in the development of object-oriented programming languages[19].

A specification in Object-Z of the simple buffer system of Figure 1 is given below.

$Buffer[T]$

$$
\begin{array}{|l}
buf : \mathbb{P}\, T \\
\hline
\# buf \leqslant 1
\end{array}
$$

I_{NIT}

$$
\begin{array}{|l}
buf = \varnothing
\end{array}
$$

In

$$
\begin{array}{|l}
\Delta(buf) \\
item? : T \\
\hline
buf = \varnothing \\
buf' = \{item?\}
\end{array}
$$

Out

$$
\begin{array}{|l}
\Delta(buf) \\
item! : T \\
\hline
buf = \{item!\} \\
buf' = \varnothing
\end{array}
$$

The class *Buffer* has one (nameless) state schema, an initial state schema and two operations *In* and *Out*. The operations differ from Z operation schemas in that they have, in addition to the declaration and predicate parts, a Δ-list which lists those class variables whose values may change when the operation is applied to an object of the class.

In the class *System* below, $b1$ and $b2$ denote objects of the class *Buffer*.

$$
\begin{array}{|l}
\hline
\textit{System}[T] \\
\quad
\begin{array}{|l|}
\hline
b1, b2 : \textit{Buffer}[T] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textsc{Init} \\
b1.\textsc{Init} \wedge b2.\textsc{Init} \\
\hline
\end{array} \\
\quad \textit{In} \mathrel{\widehat=} b1.\textit{In} \\
\quad \textit{Transfer} \mathrel{\widehat=} b1.\textit{Out} \parallel b2.\textit{In} \\
\quad \textit{Out} \mathrel{\widehat=} b2.\textit{Out} \\
\hline
\end{array}
$$

The notation $b1.\textsc{Init}$ denotes initialisation of object $b1$ according to the $\textsc{Init}$ schema in *Buffer*, and the notations $b1.\textit{In}$ and $b1.\textit{Out}$ denote the operations which change the state of $b1$ according to the *Buffer* operations *In* and *Out* respectively. These operations include an implicit $\Delta(b1)$ so that $b2$ is not changed. The notations involving $b2$ have similar meanings.

When two or more operation schemas are conjoined, the Δ-list of the resulting operation schema unites all of the variables in the Δ-lists of the constituent schemas. The parallel operator '||', used in the definition of *Transfer* above, is a binary operator used to achieve inter-object communication. The operator conjoins two operation schemas but also identifies (equates) inputs in either schema with outputs in the other schema having the same type and basename (i.e. apart from the '?' or '!'). Identified inputs and outputs are hidden in the resulting schema, i.e. they are not available in the environment of objects of the class.

The use of the Object-Z parallel operator to capture synchronisation differs from the use of the concurrency operator in CSP since the former is placed between actual operations, or events, whereas the latter is placed between the specifications of processes, or objects. The CSP approach is more concise; however the Object-Z approach allows more flexibility when naming operations as operations that synchronise don't have to have the same name.

3 The game of Life

Systems can be specified either from a top-level functional point of view or in terms of their internal structure. In this section we specify Conway's game of Life functionally using Object-Z and then in terms of a concurrent implementation using both Object-Z and CSP. The lower-level specifications are used to compare the techniques for composing component specifications and capturing state information in these two languages.

The game of Life is a simulation of the evolution of an infinite array of cells. Each cell has eight neighbours (including its four diagonal neighbours). In each generation of the evolution, each cell is either *alive* or *dead*. The status of a cell in the next generation depends on its present status and the present status of its neighbours as follows:

- a live cell with less than two live neighbours will die (of loneliness),
- a live cell with two or three live neighbours will remain alive,
- a live cell with four or more live neighbours will die (of overcrowding), and
- a dead cell with exactly three neighbours will become alive, otherwise it will remain dead.

The functionality of the game of Life can be captured by the following Object-Z specification. Each position in the cartesian plane is associated with a status, i.e. the status of the cell at that position. A single operation *Change* relates the before and after status of each cell according to the rules listed above.

Life

$$Status == alive \mid dead$$

$$cells : \mathbb{Z} \times \mathbb{Z} \rightarrow Status$$

Change

$$\Delta(cells)$$

$$
\begin{array}{l}
\forall (i,j) : \mathrm{dom}\ cells \bullet \\
\quad \mathbf{let}\ living\_neighbours == \\
\qquad \{(k,l) : \mathrm{dom}\ cells \mid cells(k,l) = alive \\
\qquad\qquad \wedge \mid i-k \mid \leqslant 1 \wedge \mid j-l \mid \leqslant 1 \\
\qquad\qquad \wedge \neg\ (i = k \wedge j = l)\} \bullet \\
\qquad \#living\_neighbours \notin \{2,3\} \Rightarrow cells'(i,j) = dead \\
\qquad \#living\_neighbours = 2 \Rightarrow cells'(i,j) = cells(i,j) \\
\qquad \#living\_neighbours = 3 \Rightarrow cells'(i,j) = alive
\end{array}
$$

This specification has no bias towards a particular implementation. It could be implemented either as an array of cells whose components are updated sequentially, or as a collection of cell processes (objects) operating concurrently. If we choose the latter approach then it is desirable, as part of the design process, to specify the system in terms of its components.

This could be done in CSP as shown in the following specification. Each cell broadcasts its status to its neighbours and when all the cells have broadcasted they synchronise on the event which allows them to change their status. Each cell is identified by its position in the cartesian plane and has three parameters:

- *status* – a variable denoting the cell's status (either alive or dead),
- *num_living* – a natural number from 0 to 8 denoting the number of living neighbours that have broadcasted their status since the last status change, and
- *broadcasted* – a boolean variable denoting whether or not the cell has broadcasted its status since the last status change.

Since CSP models concurrency by interleaving events, the size of the cell array is restricted to be finite; if it were infinite the composite process would never perform a change event since it would first have to engage in an infinite number of broadcast events. To avoid boundary problems we assume this finite array is a torus. Alternatively, we could have partitioned the infinite array of cells into a finite number of infinite sets where the cells in each partition synchronise on their broadcast events. We have chosen the former approach for reasons of simplicity.

The constants *width* and *height* are natural numbers denoting the width and height of the cell array respectively.

$$
\begin{aligned}
& CELL_{i,j}(status, num\_living, false) \\
& \quad = broadcast.i.j.status \rightarrow CELL_{i,j}(status, num\_living, true)
\end{aligned}
$$

$$
\begin{aligned}
& CELL_{i,j}(status, num\_living, broadcasted) \\
& \quad = \mathbf{if}\ k \in \{(i-1) \bmod width, i, (i+1) \bmod width\} \\
& \qquad\quad \wedge\ l \in \{(j-1) \bmod height, j, (j+1) \bmod height\} \\
& \qquad\quad \wedge\ \neg\,(k = i \wedge l = j)\ \mathbf{then} \\
& \qquad broadcast.k.l.alive \rightarrow CELL_{i,j}(status, num\_living + 1, broadcasted) \\
& \qquad \Box \\
& \qquad broadcast.k.l.dead \rightarrow CELL_{i,j}(status, num\_living, broadcasted)
\end{aligned}
$$

$$
\begin{aligned}
& CELL_{i,j}(status, num\_living, true) \\
& \quad = \mathbf{if}\ alive < 2 \vee alive > 3\ \mathbf{then} \\
& \qquad change \rightarrow CELL_{i,j}(dead, 0, false) \\
& \qquad \mathbf{else\ if}\ alive = 2\ \mathbf{then} \\
& \qquad\quad change \rightarrow CELL_{i,j}(status, 0, false) \\
& \qquad\quad \mathbf{else}\ change \rightarrow CELL_{i,j}(alive, 0, false)
\end{aligned}
$$

$$
LIFE = \mathop{||}_{i<width,\ j<height} CELL_{i,j}(status, 0, false)
$$

The system consists of the concurrent composition of all cells whose i and j indices are natural numbers from 0 to $width - 1$ and 0 to $height - 1$ respectively. The initial status of each cell is undefined.

A similar specification can be given in Object-Z as follows where

$$Status == alive \mid dead$$

denotes the possible values of a cell's status. The variables in class *Cell* have the same interpretation as the corresponding variables in the CSP specification.

Cell

$$
\begin{array}{l}
status : Status \\
num\_living : 0 \mathinner{..} 8 \\
broadcasted : Boolean
\end{array}
$$

INIT

$$
\begin{array}{l}
num\_living = 0 \\
\neg\ broadcasted
\end{array}
$$

Broadcast

$$
\begin{array}{l}
\Delta(broadcasted) \\
status! : Status \\
\hline
\neg\ broadcasted \\
status! = status \\
broadcasted'
\end{array}
$$

Receive

$$
\begin{array}{l}
\Delta(num\_living) \\
status? : Status \\
\hline
status? = alive \Rightarrow num\_living' = num\_living + 1 \\
status? = dead \Rightarrow num\_living' = num\_living
\end{array}
$$

Change

$$
\begin{array}{l}
\Delta(status, num\_living, broadcasted) \\
\hline
broadcasted \\
(num\_living < 2 \lor num\_living > 3) \Rightarrow status' = dead \\
num\_living = 2 \Rightarrow status' = status \\
num\_living = 3 \Rightarrow status' = alive \\
num\_living' = 0 \\
\neg\ broadcasted'
\end{array}
$$

The state of class *Life* consists of a function mapping the set of possible cell positions to objects of class *Cell*.

$$
\begin{array}{l}
\textit{Life} \\
\hline
\quad \mathit{height}, \mathit{width} : \mathbb{N} \\[1ex]
\hline
\quad \mathit{cells} : (0\mathinner{\ldotp\ldotp} \mathit{width}-1) \times (0\mathinner{\ldotp\ldotp} \mathit{height}-1) \rightarrow \mathit{Cell} \\
\hline
\\
\textsc{Init} \\
\hline
\quad \forall\, c : \mathrm{ran}\ \mathit{cells} \bullet c.\textsc{Init} \\
\hline
\\
\Phi \mathit{SelectCell} \\
\hline
\quad \Delta(\mathit{cells}) \\
\quad i, j : \mathbb{N} \\
\quad c, c' : \mathit{Cell} \\
\hline
\quad (i, j) \in \mathrm{dom}\ \mathit{cells} \\
\quad c = \mathit{cells}(i, j) \\
\quad \mathit{cells}'(i, j) = c' \\
\hline
\\
\Phi \mathit{NeighboursReceive} \\
\hline
\quad \Delta(\mathit{cells}) \\
\quad i, j : \mathbb{N} \\
\quad \mathit{status}? : \mathit{Status} \\
\hline
\quad \mathbf{let}\ \mathit{neighbours} == \\
\qquad\qquad \{(k, l) : \mathbb{N} \times \mathbb{N} \mid k \in \{(i-1) \bmod \mathit{width}, i, (i+1) \bmod \mathit{width}\} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ l \in \{(j-1) \bmod \mathit{height}, j, (j+1) \bmod \mathit{height}\} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \neg\ (k = i \wedge l = j)\} \bullet \\
\qquad \forall (k, l) : \mathit{neighbours} \bullet \exists\, n, n' : \mathit{Cell} \bullet \\
\qquad\qquad\quad n = \mathit{cells}(k, l) \\
\qquad\qquad\quad n.\mathit{Receive} \\
\qquad\qquad\quad \mathit{cells}'(k, l) = n' \\
\qquad (\mathit{neighbours} \cup \{(i, j)\}) \ntriangleleft \mathit{cells}' = (\mathit{neighbours} \cup \{(i, j)\}) \ntriangleleft \mathit{cells} \\
\hline
\\
\mathit{Broadcast} \mathrel{\widehat{=}} (\Phi \mathit{SelectCell} \bullet c.\mathit{Broadcast}) \parallel \Phi \mathit{NeighboursReceive} \\
\\
\mathit{Change} \\
\hline
\quad \Delta(\mathit{cells}) \\
\hline
\quad \forall (i, j) : \mathrm{dom}\ \mathit{cells} \bullet \exists\, c, c' : \mathit{Cell} \bullet \\
\qquad\qquad c = \mathit{cells}(i, j) \\
\qquad\qquad c.\mathit{Change} \\
\qquad\qquad \mathit{cells}'(i, j) = c' \\
\hline
\end{array}
$$

The class *Life* has two framing schemas identified by names beginning with the symbol Φ. These schemas don't correspond to actual operations but are combined with other schemas to construct operations. They are generally used when we have an aggregate of objects, such as the function *cells*, and wish to change one or a limited number of these objects leaving the rest unchanged.

The framing schema $\Phi SelectCell$ chooses a cell with position (i, j) and updates its value. The schema doesn't specify how its value is updated nor what happens to the rest of the cells in the array. The framing schema $\Phi NeighboursReceive$ specifies that all the neighbours of the cell with position (i, j) perform a receive operation and that all other cells in the array, apart from the cell with position (i, j), are unchanged.

The operation *Broadcast* uses these framing schemas to specify the selected cell performing a *Broadcast* operation in synchronisation with all its neighbours performing a *Receive* operation. Synchronisation between the selected cell and its neighbours is captured by the parallel operator and between the individual neighbours by the predicate of $\Phi NeighboursReceive$. The latter method of capturing synchronisation is also used in the operation *Change* which specifies all of the cells performing a *Change* operation. This operation cannot occur until all cells are in a state where the precondition of the *Cell* operation *Change* is true, i.e. it may only occur after all cells have broadcasted.

Comparing Specification Styles

The specification of the cell components in CSP and Object-Z are quite similar apart from the fact that the CSP specification requires, as part of the cell's state, its position in the cartesian plane. This difference arises because of the different approaches to composing the concurrent components. The CSP approach, based on a distributed concurrency operator, allows little state information at the system level and so this information is captured within the components. The Object-Z approach, based on the parallel operator and schema predicates, associates each cell with its position by a system-level function. Although the CSP approach is more concise, the ability to distribute state information between the system and component specifications in Object-Z aids in handling complexity in large systems.

Another distinction between the two specifications is the way they represent state information and the typing of state variables. Object-Z captures the cell's state information in the state schema of the class and all state variables are explicitly typed. Types which are not an integral part of the language, such as *Status*, have to be declared explicitly. CSP, on the other hand, introduces parameters and extensions to the name of the process in the form of subscripts to capture state information. State variables in CSP are not explicitly typed; the values they can acquire must be inferred (informally) from the specification and accompanying explanatory text.

Both approaches are low-level, i.e. they correspond closely in form to an actual implementation. The CSP specification is structured much like a program written in a concurrent programming language such as occam[11] whereas the Object-Z specification is structured much like a program written in an object-oriented programming language like Eiffel[14].

4 A simple card game

Often the best way to specify the functionality of a system is in terms of concurrent components. In this section we provide an example of such a system: a simple card game. We use this example to look at specifying fairness in Object-Z.

The game is played by a collection of players. The game begins with each player having the same even number of cards from a given pack. A move consists of a player selecting another player and offering the selected player some cards. The selected player must reply by accepting the cards and offering, from his or her original hand, the same number of cards in response. Players discard all their pairs of cards with the same face value. The game continues so long as there are two or more players still holding cards in their hands.

Let *Card* denote the set of all cards to be used in the game and *Name* denote the set of names (unique identifiers) for players of the game. We define $Even_1 == Even - \{0\}$ where $Even \subset \mathbb{N}$ denotes the set of even numbers, and let $dealt : Even_1$ be a constant denoting the number of cards initially held by each player.

The class *Player* specifies a player. The predicates of the operation schemas have been elided for reasons of simplicity.

Player

$name : Name$

$hand : \mathbb{P}\ Card$

$\textsc{Init}$
$\#hand = dealt$

SelectPartnerSwap
$\Delta(hand)$
$partner! : Name$
$offer! : \mathbb{P}\ Card$
$respond? : \mathbb{P}\ Card$
$\ldots$

SelectedAsPartnerSwap
$\Delta(hand)$
$partner? : Name$
$offer? : \mathbb{P}\ Card$
$respond! : \mathbb{P}\ Card$
$\ldots$

DiscardPairs
$\Delta(hand)$
$\ldots$

The operation *SelectPartnerSwap* corresponds to the player selecting a partner and swapping some cards, the operation *SelectedAsPartnerSwap* corresponds to the player being selected as a partner and swapping some cards, and the operation *DiscardPairs* corresponds to the player discarding all matching pairs of cards.

We can now specify the class *Game* denoting the game.

Game

$$players : \mathbb{P}\, Player$$

$$\forall\, p_1, p_2 : players \mid p_1 \neq p_2 \bullet \quad p_1.name \neq p_2.name \quad p1.hand \cap p2.hand = \varnothing$$

INIT

$$\forall\, p : players \bullet p.\mathrm{INIT}$$

ΦSelectPlayers

$$\Delta(players)$$
$$p_1, p_1', p_2, p_2' : Player$$

$$\{p_1, p_2\} \subseteq players$$
$$p_1 \neq p_2$$
$$players' = (players - \{p_1, p_2\}) \cup \{p_1', p_2'\}$$

$$Swap \mathrel{\widehat=} \Phi SelectPlayers \bullet (p_1.SelectPartnerSwap \parallel p_2.SelectedAsPartnerSwap)$$

Discard

$$\Delta(players)$$

$$\forall\, p : players \bullet \exists\, p' : players' \bullet p.Discard$$

The game consists of a set of distinct players. The framing schema Φ*SelectPlayers* updates the value of two players leaving the rest unchanged. The operation *Swap* uses this schema to specify two players exchanging cards. The operation *Discard* specifies the discarding of matching pairs of cards by each player.

Specifying fairness

An important property of concurrent systems is fairness. Fairness ensures that each concurrent component makes progress whenever possible. For example, we may wish to extend the specification of the card game to guarantee that each player holding cards will eventually be involved in a swap with some other player. Although languages like CSP can capture fairness with the introduction of an auxiliary 'supervising' process, they have no means for capturing fairness at a higher level of abstraction.

Object-Z, on the other hand, allows us to restrict the set of possible implementations of a system in a given environment through the use of *history invariants*. (A formal treatment of history invariants in Object-Z is given by Duke and Duke[4].)

Although history invariants could be expressed in classical predicate logic, the readability of such invariants benefits from the use of temporal logic notation. A description of the temporal logic operators used in Object-Z is given below (where P is some predicate). Formal definitions may be found in Moszkowski[16].

$\Box P$	P always holds.
$\Diamond P$	P eventually holds.
$\bigcirc P$	P holds after the next event.

In writing history invariants we let the keyword **op** denote the name of the current operation, and refer to class and operation variables directly.

For example, we can specify that each player participates fairly in the card game by extending the specification of the class *Game* as follows.

Game

as before

$$\forall p : players \bullet \Box(p.hand \neq \varnothing \Rightarrow \Diamond(\mathbf{op} = Swap \wedge p \in \{p_1, p_2\}))$$

The history invariant states that whenever a player's hand is not empty, that player will eventually be involved in a swap. Such a history invariant only holds when an object of the class is placed within a willing environment, e.g. it isn't required to hold in an environment that never permits occurrences of the operation *Swap*. Otherwise, we would be placing unrealisable restrictions on the system (see Abadi and Lamport[1] for a discussion of this issue).

5 Conclusions

Although concurrency is essentially an implementation issue, we often wish to incorporate it in the specification of a system's functionality. This allows us to:

- create a hierarchy of views aiding the refinement of a specification to a concurrent implementation, and
- enhance the clarity of a specification where the system is not easily described except in terms of its concurrent components.

The first point was illustrated by considering Conway's game of Life. This example was specified functionally using Object-Z and in terms of a concurrent implementation using both CSP and Object-Z. To model concurrent composition, CSP uses an operator between processes whereas Object-Z uses an operator between operations or events. Consequently, CSP specifications at the system-level are more concise, however, the individual component specifications are often more complex.

The second point was illustrated by the specification of a simple card game in Object-Z. This specification was extended so that the individual players in the game participated fairly. Such fairness constraints can be captured in Object-Z by temporal logic history invariants.

Acknowledgements

We wish to thank David Duke, Paul King and Gordon Rose for their useful comments and suggestions. Graeme Smith is supported by an OTC postgraduate scholarship.

References

[1] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lect. Notes in Comput. Sci.*, pages 1–41. Springer-Verlag, 1990.

[2] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways for your Mathematical Plays*. Academic Press, 1982.

[3] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. North-Holland, 1990.

[4] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lect. Notes in Comput. Sci.*, pages 242–262. Springer-Verlag, 1990.

[5] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1991.

[6] R. Duke, G. Rose, and A. Lee. Object-oriented protocol specification. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 325–338. North-Holland, 1990.

[7] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, 1987.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[9] ISO/IEC JTC1/SC21/WG7. *Information Retrieval, Transfer and Management for OSI*, November 1989. Working Document N4887.

[10] C. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice-Hall, 1986.

[11] G. Jones and M. Goldsmith. *Programming in occam 2.* International Series in Computer Science. Prentice-Hall, 1988.

[12] M. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1(3):213–228, 1989.

[13] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: the ethics of concurrent termination. In S. Even and O. Kariv, editors, *Proc. 8th Intl. Colloquium on Automata, Languages and Programming (ICALP'81)*, volume 115 of *Lect. Notes in Comput. Sci.*, pages 264–277. Springer-Verlag, 1981.

[14] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice-Hall, 1988.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1980.

[16] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[17] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* International Series in Computer Science. Prentice-Hall, 1990.

[18] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[19] A. Yonezawa and M. Tokoro, editors. *Object-oriented concurrent programming.* MIT Press, 1987.