Z and Hoare Logics

Antoni Diller

Abstract

Z is gaining ground in the software development community as a specification language, but there is at present no standard way of relating a **Z** specification to program code. Hoare logics have been around for about 20 years. They are well understood and widely taught as a method of proving that a program meets its specification. In this paper I look at how a software development might use both techniques and both notations to provide a path from a high-level **Z** specification to program code. Rules and conventions for combining the two notations are given and their use is illustrated by two case studies.

Antoni Diller, "Z and Hoare Logics", in J. E. Nicholls (eds.), *Z User Workshop: York 1991*, [London, Springer-Verlag, 1992], pp. 59–76.

Z and Hoare Logics

Antoni Diller
School of Computer Science
Aston Webb Building
University of Birmingham
Edgbaston
Birmingham
B15 2TT
England

A.R.Diller@cs.bham.ac.uk

Abstract

Z is gaining ground in the software development community as a specification language, but there is at present no standard way of relating a **Z** specification to program code. Hoare logics have been around for about 20 years. They are well understood and widely taught as a method of proving that a program meets its specification. In this paper I look at how a software development might use both techniques and both notations to provide a path from a high-level **Z** specification to program code. Rules and conventions for combining the two notations are given and their use is illustrated by two case studies.

1 Introduction

The use of formal specification methods in general and **Z** in particular is gaining ground in the programming community. Some evidence for this can be found in the growing number of books available on the subject. Restricting my attention to **Z**, recent years have seen the publication of the following textbooks: Ince (1988), Woodcock and Loomes (1988), Diller (1990), Potter, Sinclair and Till (1991), Lightfoot (1991) and Norcliffe and Slater (1991). Spivey (1989) is a comprehensive reference manual, the denotational semantics of **Z** is provided in Spivey (1988) and Hayes (1987) is a collection of specification case studies.

The general approach to constructing and developing a formal specification is now well established. First, a high-level specification is written employing mathematical data types the implementability of which is ignored for the time being. Then a lower-level specification is written which makes use of data types that are closer to the sorts of data type found in modern imperative programming languages (like Pascal and Modula). Various proof-obligations have to be discharged in order for the lower-level specification to be a correct refinement or reification of the

higher-level specification.[1] This process of decomposition may have to be repeated several times. Eventually, a fairly low-level and concrete specification is arrived at.

At present, however, there is no universally accepted method within the **Z** community of obtaining a program from a low-level **Z** specification. Spivey (1989), pp. 12–19, uses the reification approach; Morgan (1990) has devised a special refinement calculus;[2] while Wordsworth (1988) makes use of Dijkstra's guarded command language. In this paper I wish to suggest a different approach and that is to use a Hoare logic to bridge the gap between a low-level **Z** specification and the program that implements it. (Refining abstract specifications to concrete ones is still done as mentioned above.) The use of Hoare logics is well understood. (See, for example, the books by Alagić and Arbib, (1978), Gries (1981), Backhouse (1986), Baber (1987), Gumb (1989), Dromey (1989) and Kaldewaij (1990).) In this paper I show that it is straightforward to relate a program to a **Z** specification by means of a few partial correctness specifications. (The complications found in chapter 10 of Jones (1986) arising from the combination of VDM specifications and Hoare logics do not appear here. Sticking to a few simple conventions allows us to use the standard form of a Hoare logic.) As the purpose of this paper is to show how **Z** can be linked to a Hoare logic I do not attempt to compare the method presented here with alternatives. That would be a suitable task for another paper.

Thus, the model of software development that I am proposing is as follows:[3]

Specification: using **Z**.

Data development: using **Z**.

Translation: new program variables are introduced and partial correctness specifications are generated (containing dummy commands) from the schemas of the lowest-level specification.

Implementation: suitable commands (from a programming language of your choice) are worked out to satisfy the partial correctness specifications and they are proved to meet their specifications.

The plan of this paper is as follows: In section 2 I present a concise statement of the translation phase of the above model and in section 3 I list some rules that I have found useful in manipulating the partial correctness specifications that are generated in the translation phase. In section 4 I work through a small example concentrating especially on showing how partial correctness specifications are generated from the schemas of a **Z** specification and in section 5 I apply this approach to a larger example and for this I have chosen the specification of a computerised class manager's assistant found in King (1990). An appendix contains a brief account of the Hoare logic rules and axioms that I use in this paper. Throughout this paper I refer exclusively to partial correctness and I say nothing about total correctness. The reason for this is that the purpose of this paper is to show how **Z** and a Hoare logic can be combined. The method of combination that I present works both for partial correctness and also for total correctness specifications. To discuss issues of termination, however, would introduce a needless complication which would not add anything to my account of how a **Z** specification can be combined with a Hoare logic.

[1] Details can be found in Diller (1990), chapter 13, and Spivey (1989), pp. 3ff.

[2] King (1990) shows how the refinement calculus can be combined with a **Z** specification.

[3] Compare King (1990), p. 6.

2 The Translation Phase

In this paper I show how a **Z** specification can be related to a programming language by means of a Hoare logic. The essential features of the translation that I employ are as follows:

(1) Program variables must be chosen that are different from all the variables used in the **Z** specification we are trying to implement.[4]

(2) No specification variable can occur as a program variable.[5]

(3) Let Op be a **Z** schema. Then the partial correctness specification corresponding to Op is $\{PreOp\ \sigma \wedge ident\}\ \Gamma\ \{Op\ \tau\}$ where σ and τ are substitutions and $ident$ is a predicate built up as follows:

   (a) For every input variable $i?$ in Op, whose corresponding program variable is I, both σ and τ must contain $[I/i?]$.

   (b) For every output variable $o!$ in Op, whose corresponding program variable is O, τ must contain $[O/o!]$.

   (c) For every variable having a before (x) and after (x') form in Op $ident$ must contain a conjunct $X = x$ and τ must contain a substitution $[X/x']$, where X is the corresponding program variable.

The function of the predicate $ident$ in (3) is to give an initial value to program variables like X, which correspond to specification variables having both a before and after form.

The reason for the separation that I make between specification and program variables is that they are very different kinds of thing. A specification variable is a mathematical variable whose meaning remains constant in any scope in which it occurs; whereas a program variable can have many values inside a single scope.

3 Useful Rules

In working through a number of examples I have found the following rules useful. Not all of them are used in this paper. The first three rules express results about the interdeducibility of certain pairs of formulas:

$$A \wedge (A \iff B) \dashv\vdash B \wedge (A \iff B), \tag{A}$$

$$\neg A \wedge (A \iff B) \dashv\vdash \neg B \wedge (A \iff B). \tag{B}$$

If $1 \leq i, j \leq n$ and $i \neq j$ and P_j is $x = y$, then

$$P_1 \wedge \ldots \wedge P_i \wedge \ldots \wedge P_n \dashv\vdash P_1 \wedge \ldots \wedge P_i[x/y] \wedge \ldots \wedge P_n. \tag{C}$$

Note that P_j occurs on both sides of this logical equivalence.

[4]The need to clearly distinguish between program variables and specification variables in a Hoare logic is well explained by Nielson and Nielson (1992), p. 176, where specification variables are called *logical* variables. Concerning them they say, 'The role of these variables is to "remember" the initial values of the program variables.'

[5]This does not apply to *bound* specification variables like i in $(\exists i{:}\,1\,..\,NUM \bullet CL\,i = S)$.

The next group of rules concern properties of a Hoare logic. First, I have an alternative version of the rule for introducing the conditional:

$$\frac{\{P \wedge A\}\ \Gamma_1\ \{Q_1\} \quad \{P \wedge \neg A\}\ \Gamma_2\ \{Q_2\}}{\{P\}\ \textbf{if}\ A\ \textbf{then}\ \Gamma_1\ \textbf{else}\ \Gamma_2\ \{Q_1 \vee Q_2\}} \qquad (D)$$

This is proved straightforwardly from the standard rule for introducing the conditional and postcondition weakening.

If $P \dashv\vdash Q$, then both the following hold:

$$\{P\}\ \Gamma\ \{R\} \dashv\vdash \{Q\}\ \Gamma\ \{R\}, \qquad (E)$$
$$\{R\}\ \Delta\ \{P\} \dashv\vdash \{R\}\ \Delta\ \{Q\}. \qquad (F)$$

These are easily proved using postcondition weakening and precondition strengthening.

If $1 \leq i, j \leq n$ and $i \neq j$ and P_j is $x = y$, then both the following hold:

$$\{P_1 \wedge \ldots \wedge P_i \wedge \ldots \wedge P_n\}\ \Gamma\ \{Q\} \dashv\vdash \{P_1 \wedge \ldots \wedge P_i[x/y] \wedge \ldots \wedge P_n\}\ \Gamma\ \{Q\}, \quad (G)$$
$$\{R\}\ \Delta\ \{P_1 \wedge \ldots \wedge P_i \wedge \ldots \wedge P_n\} \dashv\vdash \{R\}\ \Delta\ \{P_1 \wedge \ldots \wedge P_i[x/y] \wedge \ldots \wedge P_n\}. \quad (H)$$

These follow by combining (C) with (E) and (F).

If B is a new Boolean-valued program variable not occurring in A, Γ_1 or Γ_2, then:

$$\{P\}\ \textbf{if}\ A\ \textbf{then}\ \Gamma_1\ \textbf{else}\ \Gamma_2\ \{Q\} \dashv\vdash \{P\}\ B := A; \textbf{if}\ B\ \textbf{then}\ \Gamma_1\ \textbf{else}\ \Gamma_2\ \{Q\}. \quad (I)$$

The final rule—rule (J)—states that the precondition operator distributes through schema disjunction. Let S and T be schemas and $SorT \mathrel{\hat{=}} S \vee T$. Then $PreSorT$ and $PreS \vee PreT$ are the same schema. This follows from the fact that the existential quantifier distributes through disjunction. (Note that Spivey (1989), p. 72, defines $PreS \mathrel{\hat{=}} \mathrm{pre}\, S$. Thus $PreSorT$ is the same as $\mathrm{pre}\, SorT$.)

4 A Small Example

4.1 The Specification

In order to illustrate how **Z** specifications can be related to programs by means of a Hoare logic I shall consider a very simple specification as my example. This specification makes use of only one user-defined type, namely *Report*, which is defined as follows:

$$Report ::= okay \quad | \quad out\_of\_bounds.$$

The state of this specification is given by the schema *Table*. This contains just one variable t and no predicates. The variable t is an array of ten integers.

Table

$$t : 1 \,..\, 10 \to \mathbf{Z}$$

The schemas $\Delta Table$ and $\Xi Table$ are defined in the usual way.

$$\Delta Table \mathrel{\hat{=}} Table \wedge Table',$$
$$\Xi Table \mathrel{\hat{=}} \Delta Table \mid t' = t.$$

In the initial state all the elements of the array t' are 0.

$$InitTable' \triangleq Table' \mid \forall i : 1 \mathinner{..} 10 \bullet t'(i) = 0.$$

Several operations will now be defined on this simple specification. The first, namely *CoreUpdate*, just alters the value of one element in the array. The new value is represented by $v?$ and the number $p?$ indicates which array element is being updated.

CoreUpdate

$\Delta Table$
$p? : \mathbf{N}$
$v? : \mathbf{Z}$

$p? \in 1 \mathinner{..} 10$
$t' = t \oplus \{p? \mapsto v?\}$

The schema *Success* is used to indicate the successful completion of an operation:

Success

$rep! : Report$

$rep! = okay$

The *Update* schema is then defined as follows:

$$Update \triangleq CoreUpdate \land Success.$$

The schema *OutOfBounds* specifies what happens when the variable $p?$ is out of bounds:

OutOfBounds

$\Xi Table$
$p? : \mathbf{N}$
$rep! : Report$

$p? \notin 1 \mathinner{..} 10$
$rep! = out\_of\_bounds$

The schema *DoUpdate* is then defined as:

$$DoUpdate \triangleq Update \lor OutOfBounds.$$

The second operation, namely *CoreLookUp*, finds out what the value of a particular array element is. The position of the required element is given by $p?$ and the result is placed in $v!$

CoreLookUp

$\Xi Table$
$p? : \mathbf{N}$
$v! : \mathbf{Z}$

$p? \in 1 \mathinner{..} 10$
$v! = t(p?)$

The operation *LookUp* is then defined as:

$$LookUp \triangleq CoreLookUp \land Success.$$

The schema *DoLookUp* is then defined as:

$$DoLookUp \triangleq LookUp \lor OutOfBounds.$$

The third operation, namely *CoreSum*, sums together the values of all the elements in the array and puts the result into the output variable *out*!

CoreSum

$\Xi Table$
$out!: \mathbf{Z}$

$out! = \sum i: 1 \mathinner{..} 10 \bullet t(i)$

Summation is not part of standard **Z** as defined by Spivey (1989), but its meaning is so clear that there can be no harm in using it in specifications.[6] The operations *Sum* and *DoSum* can then be defined like this:

$$Sum \triangleq CoreSum \land Success,$$
$$DoSum \triangleq Sum.$$

4.2 Obtaining Partial Correctness Specifications

Given a **Z** schema, like *Update*, it is easy to derive a partial correctness specification $\{P\}\ \Gamma\ \{Q\}$ from it that any command Γ must satisfy in order to be a correct implementation of the original **Z** schema. The Greek letter Γ here takes the place of the command that we have to find in order to implement the **Z** schema we started from.

The first step in deriving $\{P\}\ \Gamma\ \{Q\}$ is to calculate the precondition schema of *Update*. A precondition schema is obtained from a given schema by hiding all the after and output variables. Thus, *PreUpdate* is given as follows:

PreUpdate

$t: 1 \mathinner{..} 10 \to \mathbf{Z}$
$p?: \mathbf{N}$
$v?: \mathbf{Z}$

$\exists t': 1 \mathinner{..} 10 \to \mathbf{Z}; rep!: Report \bullet$
$\quad p? \in 1 \mathinner{..} 10 \land t' = t \oplus \{p? \mapsto v?\} \land rep! = okay$

This can be simplified to the following form:

PreUpdate

$t: 1 \mathinner{..} 10 \to \mathbf{Z}$
$p?: \mathbf{N}$
$v?: \mathbf{Z}$

$p? \in 1 \mathinner{..} 10$

[6] The notation used to represent summation in the schema *CoreSum* was suggested by an anonymous referee.

The next step in deriving the partial correctness specification is to chose some program variables. These must be entirely new, that is to say, they must be distinct from every variable occurring in our **Z** specification. In this paper I use the convention that program variables are written entirely in uppercase letters. For the schema *Update* in this **Z** specification I need the program variables T, P, V and REP. These correspond to the specification variables t, $p?$, $v?$ and $rep!$ As *Update* makes use of t' the program variable T also corresponds to this in a way that will soon become clear.

The precondition P of the partial correctness specification that we are after is formed by constructing the conjunction of two predicates. The first conjunct is the predicate part of *PreUpdate* with the program variables P and V substituted for the specification variables $p?$ and $v?$, respectively. The second conjunct is $T = t$. Thus, the precondition that we are after is the predicate:

$$PreUpdate[P/p?][V/v?] \land T = t.$$

The postcondition of the partial correctness specification that we after is formed by substituting the program variables P, V, REP and T for the specification variables $p?$, $v?$, $rep!$ and t', respectively. Note that specification variables having a before and after variety are treated in a special way. Thus, the postcondition we are after is:

$$Update[P/p?][V/v?][REP/rep!][T/t'].$$

Putting all this together we obtain the partial correctness specification:

$$\begin{array}{l} \{P \in 1\,..\,10 \land T = t\} \\ \quad \Gamma_1 \\ \{P \in 1\,..\,10 \land T = t \oplus \{P \mapsto V\} \land REP = okay\}. \end{array} \tag{1}$$

Here, Γ_1 represents the programming language command that we are trying to find in order to implement *Update*. A suitable command Γ_1 is not difficult to find. It is:

$$T := T \oplus \{P \mapsto V\}; REP := okay$$

The proof that this is indeed a correct implementation is straightforward using the assignment axiom twice and the sequencing rule. The command $T := T \oplus \{P \mapsto V\}$ is usually written in imperative programming languages as $T[P] := V$.[7]

We go through the same steps with the schema *OutOfBounds*. First, we form the precondition schema *PreOutOfBounds*:

PreOutOfBounds

$t: 1\,..\,10 \to \mathbf{Z}$

$p?: \mathbf{N}$

$\exists t': 1\,..\,10 \to \mathbf{Z}; rep!: Report \bullet$
$\quad p? \notin 1\,..\,10 \land t' = t \land rep! = out\_of\_bounds$

This simplifies to the following:

[7] Using the notation $T := T \oplus \{P \mapsto V\}$ for altering the component of an array allows us to use the usual Hoare logic axiom for assignment.

PreOutOfBounds

$t: 1 \mathinner{..} 10 \to \mathbf{Z}$

$p?: \mathbf{N}$

$p? \notin 1 \mathinner{..} 10$

The required partial correctness specification is:

$$\{PreOutOfBounds[P/p?] \land T = t\}$$
$$\Gamma_2$$
$$\{OutOfBounds[P/p?][REP/rep!][T/t']\},$$

which, when written out in full is:

$$\{P \notin 1 \mathinner{..} 10 \land T = t\}$$
$$\Gamma_2 \tag{2}$$
$$\{P \notin 1 \mathinner{..} 10 \land T = t \land REP = out\_of\_bounds\}.$$

And a suitable Γ_2 is $REP := out\_of\_bounds$.

From (1) and (2) we can obtain—by means of rule (D)—the following partial correctness specification:

$$\{T = t\}$$
$$\textbf{if } P \in 1 \mathinner{..} 10$$
$$\textbf{then } (T := T \oplus \{P \mapsto V\}; REP := okay)$$
$$\textbf{else } REP := out\_of\_bounds$$
$$\{(P \in 1 \mathinner{..} 10 \land T = t \oplus \{P \mapsto V\} \land REP = okay) \lor$$
$$(P \notin 1 \mathinner{..} 10 \land T = t \land REP = out\_of\_bounds)\}.$$

The precondition of this is equivalent to $PreDoUpdate \land T = t$ and the postcondition to:

$$DoUpdate[P/p?][V/v?][REP/rep!][T/t'].$$

Thus, we could have achieved the same final result had we started from the schema *DoUpdate*.

We can go through the same steps with the schema *LookUp*. First, we form the precondition schema *PreLookUp*:

PreLookUp

$t: 1 \mathinner{..} 10 \to \mathbf{Z}$

$p?: \mathbf{N}$

$\exists t': 1 \mathinner{..} 10 \to \mathbf{Z}; v!: \mathbf{Z}; rep!: Report \bullet$
$p? \in 1 \mathinner{..} 10 \land v! = t(p?) \land t' = t \land rep! = okay$

This simplifies to the following:

PreLookUp

$t: 1 \mathinner{..} 10 \to \mathbf{Z}$

$p?: \mathbf{N}$

$p? \in 1 \mathinner{..} 10$

Thus, the required partial correctness specification is:

$$\{PreLookUp[P/p?] \land T = t\}$$
$$\Gamma_3$$
$$\{LookUp[P/p?][W/v!][REP/rep!][T/t']\}.$$

Note that I have used the program variable W to correspond to the specification variable v! When written out in full this partial correctness specification is as follows:

$$\{P \in 1 \mathinner{..} 10 \land T = t\}$$
$$\Gamma_3 \tag{3}$$
$$\{P \in 1 \mathinner{..} 10 \land W = t(P) \land REP = okay \land T = t\}.$$

And a suitable Γ_3 is $W := T[P]; REP := okay$.[8] The proof that this is indeed correct is straightforward. Combining (2) and (3)—by means of rule (D)—gives us:

$$\{T = t\}$$
$$\textbf{if } P \in 1 \mathinner{..} 10$$
$$\textbf{then } (W := T[P]; REP := okay)$$
$$\textbf{else } REP := out\_of\_bounds$$
$$\{(P \in 1 \mathinner{..} 10 \land W = t(P) \land REP = okay \land T = t) \lor$$
$$(P \notin 1 \mathinner{..} 10 \land T = t \land REP = out\_of\_bounds)\}.$$

Things are slightly more interesting when we come to the operation specified by the schema *Sum*. First, we work out the precondition schema *PreSum*:

PreSum

$$t: 1 \mathinner{..} 10 \to \mathbf{Z}$$

$$\exists t': 1 \mathinner{..} 10 \to \mathbf{Z}; out!: \mathbf{Z}; rep!: Report \bullet$$
$$out! = \sum i: 1 \mathinner{..} 10 \bullet t(i) \land t' = t \land rep! = okay$$

This simplifies to the following:

PreSum

$$t: 1 \mathinner{..} 10 \to \mathbf{Z}$$

The partial correctness specification that we are after is, therefore:

$$\{PreSum \land T = t\}\ \Gamma_4\ \{Sum[OUT/out!][REP/rep!][T/t']\}.$$

Here, Γ_4 represents the programming language command that we are after which will implement the schema *Sum*. Writing out this partial correctness specification in full gives us:

$$\{T = t\}\ \Gamma_4\ \{OUT = \sum i: 1 \mathinner{..} 10 \bullet t(i) \land REP = okay \land T = t\}. \tag{4}$$

[8] When subscripting arrays I write, for example, $T[I]$, but when applying a function to an argument I use the notation $t(i)$. It would have been possible to use the same notation for both, but the practice employed is more common.

In trying to find a suitable command Γ_4 it is easiest to work in stages. First, I will obtain a command Γ_5 which satisfies the partial correctness specification:

$$\{T = t\}\ \Gamma_5\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i] \wedge T = t\}. \tag{5}$$

It is not difficult to see that the following command is a suitable Γ_5:

$$\begin{array}{l}\textbf{begin new } I;\\ \Delta\left\{\begin{array}{l} \left.\begin{array}{l} OUT := T\,[1];\\ I := 1; \end{array}\right\}\Delta_1\\ \textbf{while } I \neq 10 \textbf{ do}\\ \quad \Delta_3\left\{\begin{array}{l} I := I + 1;\\ OUT := OUT + T\,[I] \end{array}\right\}\Delta_2 \end{array}\right.\\ \textbf{end}\end{array}$$

It is straightforward to show that this command is a suitable Γ_5 by means of the usual axioms and rules of a Hoare logic as found, for example, in Hoare (1969) and Hoare (1971). I will just give a sketch of the proof here. As I do not allow any specification variables, such as t, to occur in the commands that implement them, (5) follows by specification conjunction from $\{T = t\}\ \Gamma_5\ \{T = t\}$ and:

$$\{\mathit{true}\}\ \Gamma_5\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i]\}. \tag{6}$$

So, I will concentrate my attention on establishing (6).

Using the assignment axiom twice and the sequencing rule it is easy to show that:

$$OUT = \sum i{:}\,1\,..\,I \bullet T[i],$$

is an invariant of the command Δ_3. So, by precondition strengthening we get:

$$\{OUT = \sum i{:}\,1\,..\,I \bullet T[i] \wedge I \neq 10\}\ \Delta_3\ \{OUT = \sum i{:}\,1\,..\,I \bullet T[i]\}.$$

From this, by the rule for the **while**-loop, we can infer that:

$$\{OUT = \sum i{:}\,1\,..\,I \bullet T[i]\}\ \Delta_2\ \{OUT = \sum i{:}\,1\,..\,I \bullet T[i] \wedge I = 10\},$$

and this is clearly the same as:

$$\{OUT = \sum i{:}\,1\,..\,I \bullet T[i]\}\ \Delta_2\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i]\}. \tag{7}$$

This completes that part of this proof dealing with the **while**-loop.

Using the assignment axiom twice and the sequencing rule it is straightforward to show that:

$$\{\mathit{true}\}\ \Delta_1\ \{I = 1 \wedge OUT = T[I]\}.$$

As the postcondition of this implies the precondition of (7) we have:

$$\{\mathit{true}\}\ \Delta\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i]\}.$$

From this, by means of the block rule we can infer:

$$\{\mathit{true}\}\ \Gamma_5\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i]\},$$

which is what I set out to prove. This completes this part of the proof.

The following is obviously correct (as *REP* occurs neither in the precondition nor in the postcondition):

$$\begin{aligned}&\{OUT = \sum i{:}\,1\,..\,10 \bullet T[i] \land T = t\}\\ &\ REP := okay\\ &\{OUT = \sum i{:}\,1\,..\,10 \bullet T[i] \land T = t\},\end{aligned}$$

and from that and (5) above—by the sequencing rule—we have:

$$\{T = t\}\ \Gamma_5; REP := okay\ \{OUT = \sum i{:}\,1\,..\,10 \bullet T[i] \land T = t\}.$$

This completes the correctness proof.

In real-life schemas are usually implemented by procedures (or functions). This could be done in this case. The resulting procedures might be:

procedure *DO_UPDATE* (**in** $P{:}\,\mathbf{N}; V{:}\,\mathbf{Z}$; **out** $REP{:}\,Report$);
  if $P \in 1\,..\,10$
  then $(T := T \oplus \{P \mapsto V\}; REP := okay)$
  else $REP := out\_of\_bounds$

procedure *DO_LOOKUP* (**in** $P{:}\,\mathbf{N}$; **out** $W{:}\,\mathbf{Z}; REP{:}\,Report$);
  if $P \in 1\,..\,10$
  then $(W := T[P]; REP := okay)$
  else $REP := out\_of\_bounds$

procedure *SUM* (**out** $OUT{:}\,\mathbf{Z}; REP{:}\,Report$);
  $\Gamma_5; REP := okay$

In doing this the array variable T is being treated as a global variable. To complete this example I just need to show how the initial state gets implemented. The above approach works here as well and we get:

$$\{PreInitTable' \land T = t\}\ \textbf{for}\ I := 1\ \textbf{to}\ 10\ \textbf{do}\ T[I] := 0\ \{InitTable'[T/t']\}.$$

5 A Larger Example

In this section I show how the method introduced in the last section can be applied to a slightly more complicated example. For this purpose I have chosen the specification used in King (1990). This has the added advantage that readers familiar with the refinement calculus can compare the two methods.

King (1990), pp. 11–14, presents a specification of a computerised class manager's assistant and then refines it. In the approach presented in this paper that would be done in the same way. So, in order to illustrate the differences between the two approaches, I will just make use of his lower-level specification. This comes from

pp. 12–13 and 14–16 of his paper. The specification makes use of two user-defined types:

$$[Student, Response].$$

The type *Response* is defined like this:

$$Response ::= ok \quad | \quad found \quad | \quad full \quad | \quad missing.$$

The specification uses a single global constant, namely max, which gives the maximum capacity of the class:

$$max : \mathbf{N}$$
$$\hline$$
$$max > 0$$

The state space of the system is defined by the schema *Class_1*:

Class_1

$$cl : 1 \,..\, max \rightarrow Student$$
$$ex : 1 \,..\, max \rightarrow Boolean$$
$$num : 0 \,..\, max$$
$$\hline$$
$$((1 \,..\, num) \lhd cl) \in (\mathbf{N} \rightarrowtail Student)$$

The array cl records the students who are members of the class and ex records which of them has done the exercise. The number num indicates how much of the arrays cl and ex are being used. The predicate ensures that no student occurs twice in the array cl.

The successful completion of an operation is indicated by the schema *Success*:

Success

$$resp! : Response$$
$$\hline$$
$$resp! = ok$$

The schema *Enrol_ok_1* specifies the operation of a student successfully becoming a member of the class:

Enrol_ok_1

$$\Delta Class\_1$$
$$s? : Student$$
$$\hline$$
$$s? \notin \{i : 1 \,..\, num \bullet cl\, i\}$$
$$num < max$$
$$cl' = cl \oplus \{num' \mapsto s?\}$$
$$ex' = ex \oplus \{num' \mapsto false\}$$
$$num' = num + 1$$

This operation can go wrong in two ways: either the class is full already or the student is already enrolled. These two error-conditions are specified by *Full_1* and *Found_1*, respectively.

Full_1

$\Xi Class\_1$
$resp!: Response$

$num = max$
$resp! = full$

Found_1

$\Xi Class\_1$
$s?: Student$
$resp!: Response$

$\exists i: 1 \mathinner{..} num \bullet cl\, i = s?$
$resp! = found$

The schema *Enrol_1* is the total operation of someone attempting to join the class:

$$Enrol\_1 \triangleq (Enrol\_ok\_1 \land Success) \lor Full\_1 \lor Found\_1$$

This completes the material that I have taken from King (1990). For my purposes I need a schema *Enrol_okay_1* defined in this way:

$$Enrol\_okay\_1 \triangleq Enrol\_ok\_1 \land Success$$

The partial correctness specification corresponding to the schema *Enrol_okay_1* is the following, where S, CL, EX, NUM and $RESP$ are new program variables:[9]

$$\{PreEnrol\_okay\_1[S/s?] \land CL = cl \land EX = ex \land NUM = num\}$$
$$\Gamma_6$$
$$\{Enrol\_okay\_1[S/s?][CL/cl'][EX/ex'][NUM/num'][RESP/resp!]\}.$$

Expanding this and using rule (G) results in the following:

$$\{S \notin \{i: 1 \mathinner{..} NUM \bullet CL\, i\} \land NUM < max \land CL = cl \land EX = ex \land NUM = num\}$$
$$\Gamma_6$$
$$\{S \notin \{i: 1 \mathinner{..} num \bullet cl\, i\} \land num < max \land CL = cl \oplus \{NUM \mapsto S\} \land$$
$$EX = ex \oplus \{NUM \mapsto false\} \land NUM = num + 1 \land RESP = ok\}.$$

A suitable Γ_6 is the following:

$$\begin{aligned} NUM &:= NUM + 1; \\ CL &:= CL \oplus \{NUM \mapsto S\}; \\ EX &:= EX \oplus \{NUM \mapsto false\}; \\ RESP &:= ok \end{aligned}$$

The partial correctness specification corresponding to the error schema *Full_1* is as follows:

$$\{PreFull\_1[S/s?] \land CL = cl \land EX = ex \land NUM = num\}$$
$$\Gamma_7$$
$$\{Full\_1[S/s?][CL/cl'][EX/ex'][NUM/num'][RESP/resp!]\}.$$

[9]Following King (1990) I regard the state invariant as a precondition of the entire specification.

Expanding this and using rule (G) results in the following:

$$\begin{array}{l}\{NUM = max \land CL = cl \land EX = ex \land NUM = num\} \\ \quad \Gamma_7 \\ \{num = max \land RESP = full \land CL = cl \land EX = ex \land NUM = num\}.\end{array}$$

A suitable Γ_7 is $RESP = full$.

The partial correctness specification corresponding to the error schema *Found_1* is as follows:

$$\begin{array}{l}\{PreFound\_1[S/s?] \land CL = cl \land EX = ex \land NUM = num\} \\ \quad \Gamma_8 \\ \{Found\_1[S/s?][CL/cl'][EX/ex'][NUM/num'][RESP/resp!]\}.\end{array}$$

Expanding this and using rule (G) results in the following:

$$\begin{array}{l}\{(\exists i: 1 \mathinner{..} NUM \bullet CL\, i = S) \land CL = cl \land EX = ex \land NUM = num\} \\ \quad \Gamma_8 \\ \{(\exists i: 1 \mathinner{..} num \bullet cl\, i = S) \land \\ \qquad CL = cl \land EX = ex \land NUM = num \land RESP = found\}.\end{array}$$

A suitable Γ_8 is $RESP := found$.

Putting the above results together gives us the following command as an implementation of the schema *Enrol_1*:

$$\begin{array}{l}\textbf{if } (\exists i: 1 \mathinner{..} NUM \bullet CL\, i = S) \\ \textbf{then } RESP := found \\ \textbf{else if } NUM = max \\ \qquad \textbf{then } RESP := full \\ \qquad \textbf{else } NUM := NUM + 1; \\ \qquad\qquad CL := CL \oplus \{NUM \mapsto S\}; \\ \qquad\qquad EX := EX \oplus \{NUM \mapsto false\}; \\ \qquad\qquad RESP := ok\end{array}$$

There are standard ways of implementing such searches as represented by $(\exists i: 1 \mathinner{..} NUM \bullet CL\, i = S)$.[10] Using one of these and rule (I) would result in an executable command which could be made into a procedure.

6 Conclusion

In this paper I have presented a method for combining the schemas of a **Z** specification with the partial correctness specifications of a Hoare logic. Readers familiar with my earlier paper—Diller (1991)—will notice that the translation method contained in this paper is simpler than the one used in that paper.

I think that the model of software development presented in this paper has much to recommend it. In particular, I would like to mention the following:

[10] See, for example, Backhouse (1986), pp. 173–174.

(1) The method used in this paper works perfectly well with the current standard version of **Z**. No changes need to be made to **Z** in order to make it combine smoothly with a Hoare logic.

(2) Hoare logics are well understood and widely taught. Having been around for over 20 years they are part of the intellectual toolbox of the modern programmer.[11] (The refinement calculus presented in Morgan (1990), for example, is an elegant formal system, but it will take a long time for it to become part of our intellectual furniture.)

(3) As is widely known it is impossible to write an algorithm which always succeeds whose input is a specification of a programming task and whose output is an algorithm that meets that specification, but verification condition generators can be written for Hoare logics.[12] These enable the process of program verification and falsification to be automated. Such tools would make the techniques introduced in this paper easier to use in practice.

Concerning the future development of the approach presented in this paper, I think that three avenues should be pursued. The first is that more rules should be devised to make the manipulation of the special form of partial correctness specifications generated easier. The second is that tools should be written to automate those parts of the method presented here that can be automated. For example, the translation phase could easily be automated. The third is the most important. All that I have done in this paper is to present a syntactic or notational transformation of a schema in a **Z** specification into the pre- and postcondition of a Hoare triple. Both **Z** and Hoare logics have semantics associated with them. I believe that the syntactic transformation presented here can be justified by linking the semantics of **Z** and that of a Hoare logic. This I hope to do in a future paper.

Appendix: Hoare Logic Proof Rules Used

Assignment

$$\{P[E/V]\}\ V := E\ \{P\},$$

where $P[E/V]$ stands for the result of substituting E for all the free occurrences of V in P.

Sequencing

$$\frac{\{P_0\}\ \Gamma_1\ \{P_1\} \quad \{P_1\}\ \Gamma_2\ \{P_2\}}{\{P_0\}\ \Gamma_1; \Gamma_2\ \{P_2\}}\ ;\text{-}int$$

The Conditional

$$\frac{\{P \wedge S\}\ \Gamma_1\ \{Q\} \quad \{P \wedge \neg S\}\ \Gamma_2\ \{Q\}}{\{P\}\ \textbf{if}\ S\ \textbf{then}\ \Gamma_1\ \textbf{else}\ \Gamma_2\ \{Q\}}\ \textbf{if}\text{-}int$$

[11] See, for example, Cousot (1990).

[12] See chapter 3 of Gordon (1988) for details.

While-loops

$$\frac{\{P \wedge S\}\ \Gamma\ \{P\}}{\{P\}\ \textbf{while}\ S\ \textbf{do}\ \Gamma\ \{P \wedge \neg S\}}\ \textbf{while}\text{-}int$$

For-loops There is both an axiom and a rule governing the **for**-loop. First, the axiom:

$$\{P \wedge (E_2 < E_1)\}\ \textbf{for}\ V := E_1\ \textbf{to}\ E_2\ \textbf{do}\ \Gamma\ \{P\}$$

and, second, the rule:

$$\frac{\{P \wedge (E_1 \leq V) \wedge (V \leq E_2)\}\ \Gamma\ \{P[V+1/V]\}}{\{P[E_1/V] \wedge (E_1 \leq E_2)\}\ \textbf{for}\ V := E_1\ \textbf{to}\ E_2\ \textbf{do}\ \Gamma\ \{P[E_2+1/V]\}}\ \textbf{for}\text{-}int$$

The side condition for the **for**-loop is that neither V nor any variable occurring in either E_1 or E_2 can occur on the left-hand side of an assignment in Γ.

Blocks and Local Variables The rule for blocks that I have used is from Hoare (1971) where, on p. 109, it is stated in this form:

$$\frac{\{P\}\ \Gamma[Y/X]\ \{Q\}}{\{P\}\ \textbf{begin new}\ X; \Gamma\ \textbf{end}\ \{Q\}}\ \textit{block-int}$$

where Y is not free in P or Q and it does not occur in Γ, unless Y is the same variable as X.

Postcondition Weakening

$$\frac{\{P\}\ \Gamma\ \{R\} \quad R \vdash Q}{\{P\}\ \Gamma\ \{Q\}}\ \textit{post-weak}$$

Precondition Strengthening

$$\frac{P \vdash R \quad \{R\}\ \Gamma\ \{Q\}}{\{P\}\ \Gamma\ \{Q\}}\ \textit{pre-strength}$$

Specification Conjunction

$$\frac{\{P_1\}\ \Gamma\ \{Q_1\} \quad \{P_2\}\ \Gamma\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \Gamma\ \{Q_1 \wedge Q_2\}}\ \textit{spec-conj}$$

References

Alagić, S., and M.A. Arbib, (1978), *The Design of Well-Structured and Correct Programs*, Berlin, Springer-Verlag.

Baber, R.L., (1987), *The Spine of Software*: *Designing Provably Correct Software*: *Theory and Practice, or a Mathematical Introduction to the Semantics of Computer Programs*, Chichester, Wiley.

Backhouse, R.C., (1986), *Program Construction and Verification*, Hemel Hempstead, Prentice Hall.

Cousot, P., (1990), "Methods and Logics for Proving Programs", in J. van Leeuwen, (ed.), *Handbook of Theoretical Computer Science*, vol. B, *Formal Models and Semantics*, Amsterdam, Elsevier, 1990, pp. 841–993.

Diller, A., (1990), **Z**: *An Introduction to Formal Methods*, Chichester, Wiley.

Diller, A., (1991), *Relating* **Z** *Specifications and Programs Through Hoare Logics*, Research Report CSR–91–3, School of Computer Science, University of Birmingham.

Dromey, R.G., (1989), *Program Derivation*: *The Development of Programs from Specifications*, Wokingham, Addison-Wesley.

Gilmore, S., (1991), *Correctness-oriented Approaches to Software Development*, Internal Report CST–76–91, Department of Computer Science, University of Edinburgh. (Also Report ECS–LFCS–91–147, Laboratory for Foundations of Computer Science, University of Edinburgh.)

Gordon, M.J.C., (1988), *Programming Language Theory and its Implementation*: *Applicative and Imperative Paradigms*, Hemel Hempstead, Prentice Hall.

Gries, D., (1981), *The Science of Programming*, Berlin, Springer-Verlag.

Gumb, R.D., (1989), *Programming Logics*: *An Introduction to Verification and Semantics*, Chichester, Wiley.

Hayes, I., (ed.), (1987), *Specification Case Studies*, Hemel Hempstead, Prentice Hall.

Hoare, C.A.R., (1969), "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, pp. 576–580 and 583.

Hoare, C.A.R., (1971), "Procedures and Parameters: An Axiomatic Approach", in E. Engeler (ed.), *Symposium on Semantics of Algorithmic Languages*, Berlin, Springer-Verlag, 1971, pp. 102–116.

Ince, D.C., (1988), *An Introduction to Discrete Mathematics and Formal System Specification*, Oxford, Oxford University Press.

Jones, C.B., (1986), *Systematic Software Development Using VDM*, Hemel Hempstead, Prentice Hall.

Kaldewaij, A., (1990), *Programming*: *The Derivation of Algorithms*, Hemel Hempstead, Prentice Hall.

King, S., (1990), **Z** *and the Refinement Calculus*, Technical Monograph PRG–79, Oxford University Computing Laboratory.

Lightfoot, D., (1991), *Formal Specification Using* **Z**, Basingstoke, Macmillan.

Morgan, C., (1990), *Programming from Specifications*, Hemel Hempstead, Prentice Hall.

Morgan, C., K. Robinson and P. Gardiner, (1988), *On the Refinement Calculus*, Technical Monograph PRG–70, Oxford University Computing Laboratory.

Nielson, H.R., and F. Nielson, (1992), *Semantics with Applications*: *A Formal Introduction*, Chichester, Wiley.

Norcliffe, A., and G. Slater, (1991), *Mathematics of Software Construction*, Chichester, Ellis Horwood.

Potter, B., J. Sinclair and D. Till, (1991), *An Introduction to Formal Specification and* **Z**, Hemel Hempstead, Prentice Hall.

Spivey, J.M., (1988), *Understanding* **Z**: *A Specification Language and its Formal Semantics*, Cambridge, Cambridge University Press.

Spivey, J.M., (1989), *The* **Z** *Notation*: *A Reference Manual*, Hemel Hempstead, Prentice Hall.

Woodcock, J.C.P., and M. Loomes, (1988), *Software Engineering Mathematics*: *Formal Methods Demystified*, London, Pitnam.

Wordsworth, J.B., (1988), *Specification and Refinement using* **Z** *and the Guarded Command Language*: *A Compendium*, IBM United Kingdom Laboratories Ltd., Hursley.