Enhancing the Established Strategy for Constructing a Z Specification

John A. van der Poll[a] Paula Kotzé[b]

School of Computing, PO Box 392, University of South Africa, 0003, South Africa

[a]vdpolja@unisa.ac.za [b]kotzep@unisa.ac.za

Abstract

*The Established Strategy for drawing up a **Z** specification document focuses on a more abstract activity of specification, namely, how to combine schemas but is largely silent about how to construct a schema. Schema construction may benefit from the application of certain heuristics for establishing its content. While formal specification can be seen as a subsection of software engineering and design heuristics in this area are firmly in place, corresponding principles and strategies for constructing a formal specification have been relatively rare. In this paper we examine a number of formal specifications written in **Z** as well as some design principles from software engineering and areas of general design. On the strength of these, we propose a preliminary set of heuristics for the construction of a formal specification and show how these may be embedded in an enhanced strategy for specification work. We illustrate how one such enhancement, namely the use of primitives, allows a specifier to discharge an important proof obligation arising from a formal specification, where otherwise a proof is not easily arrived at.*

Keywords: *Automated reasoning, Established Strategy, formal specification, heuristics, OTTER, primitives, set theory,* Z

Computing Review Categories: *D.2.4, F.3.1, F.4.1, H.1.2, H.5.2, I.2.3*

1 Introduction

There exists a well-known pseudo algorithm, called the Established Strategy (ES) [2, 21] for drawing up a **Z** [28] specification. In essence this strategy prescribes the order in which the various parts that make up a specification document are to be developed and how schemas are to be combined to produce robust operations. This strategy may be looked upon as a set of guidelines or design principles for giving a common structure to all **Z** specifications. Design principles are 'generalizable abstractions intended to orient designers towards thinking about different aspects of their design' [22, p. 20]. Such principles tend to be written in a prescriptive manner, suggesting to designers what to support and what to avoid. When design principles are used in practice they are commonly referred to as *heuristics* [22], emphasising that something has to be done with the guidelines when they are applied to a given problem. They need to be interpreted in the design context, drawing on past experience.

General design heuristics are fairly common in the field of software engineering: Measures for high-level design are formulated in [3] while design guidelines from the early work on *structured design* are given by, for example, [39] and [11]. These guidelines were later carried forward by advocates of the object-oriented approach [38], [12] and [4].

The field of human-computer interaction (HCI) also advocates a wide variety of usability principles and heuristics for the design of user interfaces, for example [13], [6], [18] and [22]. Principles for industrial and/or graphic design and the design of artefacts for human use, are presented in [19] and [26].

Formal specification techniques have been around for a substantial period of time, and have been used in various domains, ranging from small scale requirements specifications up to full scale implementations [5] often derived through a process of formal refinement. Although details on the syntax and semantics of formal specification languages and systems are fairly well established [27], design heuristics for drawing up formal specifications have been relatively sparse.

A preliminary set of heuristics to advance the utility of a specification was reported on in [30] and the aim of the current paper is to show how these may be embedded in the definition of a new strategy for **Z**. By enhancing the current strategy we will be taking one further step in the process of presenting a specification in an intelligible way and giving a common structure to specification documents. A secondary contribution of this paper is to establish some relationships among the various heuristics.

The layout of the paper is as follows: In Section 2 we introduce the current strategy for presenting a specification document and discuss the benefits to be gained by appropriately augmenting the strategy through the introduction of suitable heuristics. Our

heuristics are aimed mainly at facilitating schema construction and are introduced in Section 3. A new strategy, enhanced with the heuristics defined previously is presented in section 4. In Section 5 we construct a small specification according to our enhanced strategy. The paper is concluded with an analysis and pointers for future work.

2 Current Established Strategy

The Established Strategy for setting up a **Z** specification is presented in [2] and [31] and has been distilled mainly from work by John Wordsworth [35, 36].

The ES embodies the following sequence of steps for constructing a specification:

- Define all global constants and basic types, and give a natural language description of these.
- Present the abstract state space, using the constants and basic types above.
- Give an initial state of the system and prove that such a state exists.
- Introduce partial definitions of each of the system operations, together with a short informal description of each.
- Calculate the precondition of each abstract operations on the state and check that the precondition is explicit in the operation's predicate; if not, modify the operation accordingly.
- Draw up a table showing all the partial operations together with their inputs, outputs and preconditions for correct operation.
- Define all schemas that present error conditions.
- Use the **Z** schema calculus to make all the partial operations total.
- Provide any additional information to assist the reader of the specification, e.g. give a summary of all the robust operations at the end.

The question of why we may want to enhance the current ES for constructing a formal specification boils down to asking what the specification is to be used for. An advantage of using a formal notation during the specification phase is that the specifier can reason about the specification formally. Reasoning about the properties of a specification is an important activity early in the process of constructing a reliable program [34]. For example, we can show that certain undesirable properties are absent from the specification. A formal specification may also be used as the starting point of a subsequent refinement phase [16], and a well-designed specification could possibly be more easily refined to code than an ad hoc specification.

A further use of a formal specification is for educating users about the proposed system, creating the need for the specification to be readable. To this end Gravell [8] proposes a number of 'readability' principles for constructing a formal specification.

The above ES goes some way towards presenting a specification document in an intelligible way but is largely silent about schema content, neither does it propose any standard for specifying the interaction among the various operations making up a specification document. Established software design principles such as *cohesion* [38, 1] are also not officially part of this strategy, neither is the use of certain accepted HCI design principles, for example, *make things visible to the user* [19]. The results of an analysis of a number of specifications in the literature support these claims and serve as motivation for improving a specification document as follows:

1. Incorporate some established software engineering design principles normally present in the final product already at the specification phase.
2. Apply a number of HCI and general design principles in the construction of a formal specification.
3. Facilitate the initial stages of a subsequent refinement process.
4. Structure the specification so as to facilitate the process of automatic proof.

In the next section we examine a number of specifications in the literature and introduce a preliminary set of heuristics to address (amongst others) the above four points during the construction of a specification. The core of these design principles were developed in [30] but no indication was given as to how these may enhance the ES for drawing up a specification.

3 Design Guidelines

3.1 Format of a Precondition

Our first design principle aims to facilitate a correct refinement from a non-deterministic specification and our vehicle is that of a rudimentary UNIX-like filing system [17, p. 64]. We have the basic types

$$[FID,\ CID\ ,SYL]$$

where FID denotes the set of all file identifiers, CID represents all channel identifiers and SYL is the set of all syllables used to make up file names (see below).

The expanded abstract state space is given by:

$$
\begin{array}{|l}
\hline
FS \\
\hline
fstore : FID \pfun FILE \\
cstore : CID \pfun CHAN \\
nstore : NAME \pfun FID \\
dnames : \mathbb{P}\, NAME;\ usedfids : \mathbb{P}\, FID \\
\hline
Front(\!| dnames \cup \mathrm{dom}\ nstore |\!) \subseteq dnames \\
usedfids = \\
\quad \mathrm{ran}\ nstore \cup \{chan : \mathrm{ran}\ cstore \bullet chan.fid\} \\
usedfids \subseteq \mathrm{dom}\ fstore \\
\hline
\end{array}
$$

where

- $FILE = \mathrm{seq}\ BYTE$, (i.e. a sequence of bytes) for $BYTE = 0\,..\,255$,
- $NAME = \mathrm{seq}\ SYL$ (file names are sequences of syllables) and
- $CHAN$ is given by

$$
\begin{array}{|l}
\hline
CHAN \\
\hline
fid : FID \\
posn : \mathbb{N} \\
\hline
\end{array}
$$

A detailed discussion of FS presented by [17, pp. 45 - 78] is beyond the scope of this paper. The following operation opens a file [17, p. 74]:

$$
\begin{array}{|l}
\hline
open \\
\hline
\Delta FS \\
name? : NAME;\ cid! : CID \\
fid, fid' : FID \\
report! : REPORT \\
\hline
(name? \in \mathrm{dom}\ nstore \land cid! \notin \mathrm{dom}\ cstore \land \\
\quad fid = fid' = nstore(name?) \land \\
\quad (\exists\, CHAN' \bullet posn' = 0 \land fid' = fid \land \\
\qquad cstore' = cstore \oplus \{cid! \mapsto \theta CHAN'\}) \land \\
\quad nstore' = nstore \land report! = OK) \\
\qquad \lor \\
(name? \notin \mathrm{dom}\ nstore \land \theta FS' = \theta FS \land \\
\quad report! = NoSuchName) \\
\qquad \lor \\
(\mathrm{dom}\ cstore = CID \land \theta FS' = \theta FS \land \\
\quad report! = NoFreeCids) \\
\hline
\end{array}
$$

The precondition $cid! \notin \mathrm{dom}\ cstore$ is the negation of $\mathrm{dom}\ cstore = CID$ and vice versa, in the sense that the system attempts to obtain a new unused channel identifier (i.e. $cid!$), and if successful, the condition $cid! \notin \mathrm{dom}\ cstore$ holds. Otherwise there are no free identifiers left and $\mathrm{dom}\ cstore = CID$ prevails.

The partial preconditions of operation *open* are:

$$(name? \in \mathrm{dom}\ nstore \land cid! \notin \mathrm{dom}\ cstore) \quad (1)$$

$$(name? \notin \mathrm{dom}\ nstore) \quad (2)$$

$$(\mathrm{dom}\ cstore = CID) \quad (3)$$

The total precondition of *open*, namely (1) $\lor$ (2) $\lor$ (3) is a tautology but not a partition since two of these conditions overlap. Often in a specification this non-determinism is deliberate because it allows implementers flexibility. However, if preconditions overlap in this way, then a sequence of automatic refinement steps could generate an incorrect structure:

if *precondition*1 **then** $S1$
elseif *precondition*2 **then** $S2$
elseif *precondition*3 **then** $S3$
endif (4)

The semantics of (4) requires the preconditions to be pairwise disjoint, leading to our first design heuristic:

> *Heuristic #1*: Ensure that the precondition to a total operation is a partition whenever non-determinism is not required.

3.2 Communication with the User

There is a further aspect to the above discussion as far as feedback to the user is concerned: Consider the scenario where there is no free channel available (i.e. $CID = \mathrm{dom}\ cstore$) and the input file name, $name?$, is incorrect (i.e. $name? \notin \mathrm{dom}\ nstore$). Suppose further that owing to the above non-determinism the message '*NoFreeCids*' is displayed, informing the user to wait for a channel to become available before proceeding. However, once a channel is released by another process, the user can try to reconnect again, just to be faced with the message '*NoSuchName*'. One could argue that this message should have been displayed together with the message about the channel, so that the user could have corrected the problem in the meantime, instead of simply having to wait for a free channel. This leads to our second design heuristic:

> *Heuristic #2*: Maximise communication with the user of the system.

The above heuristic agrees with an important HCI principle proposed by Norman [19, p. 140]:

'Narrow the gulfs of execution and evaluation. Make things visible, both for execution and evaluation'.

A further motivation for Norman's principle above is found in a simplified version of a library system often used in specification work. Starting with the basic types $NAME$, denoting the names of all library users and $BOOK$, representing all library books we could have the following state space:

$$
\begin{array}{|l}
\hline
Simple\_Library \\
\hline
borrowers : NAME \pfun BOOK \\
current : \mathbb{P}\, NAME \\
\hline
current = \mathrm{dom}\ borrowers \\
\hline
\end{array}
$$

The component *current* may be redundant in the given state, since we could use dom *borrowers* instead, but it serves an important purpose in the sense that it identifies all library users who at any point in time have one or more books on their name. Hence, it 'makes things visible' in line with the current heuristic.

Heuristic #2 may also be viewed as a special case of heuristic #1 since its application boils down to partitioning a comprehensive precondition into atomic parts for maximal user feedback. For schema *open* we have:

$$(name? \in \mathrm{dom}\ nstore \land cid! \notin \mathrm{dom}\ cstore)$$
`Correct operation`
$$(name? \in \mathrm{dom}\ nstore \land \mathrm{dom}\ cstore = CID)$$
`File OK but no free channel`
$$(name? \notin \mathrm{dom}\ nstore \land \mathrm{dom}\ cstore = CID)$$
`File error and no free channel`
$$(name? \notin \mathrm{dom}\ nstore \land cid! \notin \mathrm{dom}\ cstore)$$
`File error but channel available`

The reader may question the utility of the 3rd and 4th messages above. In the case of a file error and also no free channel, a user may decide that the system is too busy to continue and, therefore, postpose further working till a later stage. Converseley, with an available channel the user may want to utilise a quiet system optimally.

Next we propose a preliminary standard format for total operations in **Z**.

3.3 Signature of an Operation

Operations in **Z** often accept as domain elements the state and external input and deliver as range elements the state and additional output (e.g. *open* above). In the light of heuristic #2 we notationally separate the message from other output as formulated in the following design heuristic for a *user-level* operation, i.e., an operation which communicates with the user.

> *Heuristic #3*: Define every user-level operation, say **f**, based on the general format:

$$\mathbf{f} : Input \times State \rightarrow State \times Output \times Message \quad (5)$$

Definition (5) is stated in a preliminary form and it will be refined through further heuristics below.

This design principle may seem somewhat obvious, nevertheless it has the benefit of giving a common structure to all **Z** operations and this in turn allows a specifier to recognise a familiar structure when inspecting a schema.

Note how this heuristic also supports heuristic #2 in the sense that a familiar structure helps to 'make things visible'.

3.4 Undefined Output

Our next design principle is concerned with output generated in an error case (i.e. when the precondition to a partial operation is not satisfied). Consider the following definition of a simple data base [34] where *Key* and *Data* are basic types.

File
$$contents : Key \pfun Data$$

A robust operation to read a file is:

Read
$$contents, contents' : Key \pfun Data$$
$$k? : Key$$
$$d! : Data$$
$$r! : Message$$
$$\begin{array}{l} (\ k? \in \mathrm{dom}\ contents \land \\ \quad d! = contents\ k? \land \\ \quad contents' = contents \land \\ \quad r! = okay\ ) \\ \quad\quad \lor \\ (\ k? \notin \mathrm{dom}\ contents \land \\ \quad contents' = contents \land \\ \quad r! = key\_not\_in\_use\ ) \end{array}$$

Note that $d!$ is unspecified under the error condition $k? \notin \mathrm{dom}\ contents$. Woodcock and Davies [34, p. 222] claim that an output variable 'can take any value' if the precondition is not satisfied. However, a possible interpretation of this claim is that the value $d!$ could be given a value $contents\ k$, for any $k \in \mathrm{dom}\ contents$ which is undesirable.

Instead, we could specify that the value of an output variable like $d!$ above is *undefined* in the error case. This can be achieved by insisting that all sets from which output may be generated be 'lifted' to make provision for undefined values, similar to techniques used in programming language semantics [24]. If we denote an undefined value by $\bot$, then we extend the set *Data* to $Data_\bot = Data \cup \{\bot\}$.

This observation leads to:

> *Heuristic #4*: Ensure that all sets from which output may be generated are extended to allow for undefined values.

Heuristic #4 supports heuristic #2 (maximise user communication) since it advocates the value of an output component to be explicitly undefined in the error case instead of just being silent about its content.

The set *Message*, representing the set of all messages, is of course an exception to heuristic #4, since we simply use an appropriate string to describe an error condition. Therefore, we do not make the message part of the general *Output* parameter in (5) above, since a specifier may prefer to write this definition as:

$$\mathbf{f} : Input \times State \rightarrow State \times Output_\bot \times Message \quad (6)$$

In line with the current heuristic, we make the undefined nature of $d!$ explicit in the last disjunct in operation *Read* by adding $d! = \bot$. We also replace the declaration $d! : Data$ with $d! : Data_\bot$.

3.5 Cohesion

Our next heuristic stems from the well-known software engineering principle of *cohesion*. Bahrami [1] defines cohesion as a measure of the 'single-purposeness' of an object. High cohesion is desirable and low cohesion is considered bad design, since low cohesion implies the grouping together of unrelated activities. Yourdon [38] states that a module has good cohesion if its purpose can be expressed by 'a simple sentence containing a single verb and a single object'.

The highest and most desirable kind of cohesion is *functional* cohesion [20], which is the kind of cohesion described by [38] and which we advocate in the design of a formal specification. The natural language definition given by Yourdon above is unfortunately too imprecise and we refine the idea below.

For the purpose of achieving high functional cohesion in a formal specification, we propose breaking up every operation in the specification into a sequence of *primitive* operations. Scheurer [23, p. v] puts forward the following thesis in his preface: 'Set theory, based on logic, is a universal language in which all problems may be formulated and solved.' (Naturally any such problem must be specifiable in some notation and must have a solution that can be realised.) Since **Z** is based on first-order logic and a strongly typed fragment of Zermelo-Fraenkel (ZF) set theory [7], we propose to define every primitive as manipulating just *one* component of the abstract state space of our system, using an operation or definition from standard set theory.

The above ideas on cohesion crystallise into the following heuristic:

> *Heuristic #5*: Maintain high cohesion in a formal specification by defining every operation on the state as a sequence of primitives such that every primitive manipulates at most one state component using a standard set-theoretic operation or definition.

The use of primitives in this context is illustrated in Section 5 below. Heuristic #5 has an important benefit when reasoning about the properties of a specification: In Section 5.8 we show how this heuristic facilitates an important proof obligation that arises from the interaction between one primitive and another primitive which reverses the effect of the first primitive.

3.6 Explicit Preconditions and Relationships

In this section we return to the golden thread running through [19], namely to 'make things visible' and to show at every step which actions are applicable or allowable. For example, on page 183 in [19] Norman writes:

> 'In each state of the system, the user must readily see and be able to do the allowable actions. The visibility acts as a suggestion, reminding the user of possibilities and inviting the exploration of new ideas and methods.'

This idea is also touched on in [17] through an analysis of the expanded version of an operation to remove a file in UNIX:

destroyFS

$$
\begin{array}{l}
fstore, fstore' : FID \pfun FILE \\
cstore, cstore' : CID \pfun CHAN \\
nstore, nstore' : NAME \pfun FID \\
usedfids, usedfids' : \mathbb{P}\, FID \\
fid? : FID \\
\hline
fid? \in \dom fstore \land \\
usedfids \subseteq \dom fstore \land usedfids' \subseteq \dom fstore' \land \\
usedfids = \\
\quad \ran nstore \cup \{chan.fid \mid chan \in \ran cstore\} \land \\
usedfids' = \\
\quad \ran nstore' \cup \{chan.fid \mid chan \in \ran cstore'\} \land \\
fstore' = \{fid?\} \ndres fstore \land fid' = fid \land \\
posn' = posn \land cstore' = cstore \land \\
nstore' = nstore
\end{array}
$$

Recall that the state, namely, *FS* was defined in Section 3.1. (The complete definition of *destroyFS* [17] also mentions directory names, but these definitions are beyond the scope of our discussion.) The precondition of *destroyFS* is given by

$$fid? \in \dom fstore \land usedfids \subseteq \dom fstore \land$$
$$usedfids = \ran nstore \cup \{chan.fid \mid chan \in \ran cstore\}$$

where *fid*? represents the identifier of the file that is to be deleted and *usedfids* the set of file identifiers currently in use (e.g. open files).

A question arises from the definition of *destroyFS*: Can a file be deleted while in use? The answer is no, as we show next:

1. $usedfids' \subseteq \dom fstore'$ [postcondition of *destroyFS*]
2. $usedfids' = usedfids$ [$nstore' = nstore \land cstore' = cstore$]
3. $usedfids \subseteq \dom fstore'$ [From 1. and 2.]
4. $fid? \notin \dom fstore'$ [$fstore' = \{fid?\} \ndres fstore$]

5. $fid? \notin usedfids$ [From 3. and 4.]

Hence, a file cannot be destroyed while in use. Therefore, the condition $fid? \notin usedfids$ is actually a further precondition of the correct operation of *destroyFS*. One could argue that the absence of this condition from *destroyFS* violates the above visibility principles advocated by Norman, that is, to explicitly show all the conditions which need to hold for an action to be applicable. In fact, Morgan and Sufrin [17] call *destroyFS* a 'dishonest' definition.

This leads us to a preliminary version of our next design heuristic which is in further support of heuristic #2:

> *Heuristic #6.0*: Ensure that all operations are honest by listing all preconditions explicitly.

For another kind of 'dishonesty' in specification work we return to a classic specification of a telephone exchange by Morgan [15]:

Specify a telephone system whereby subscribers may engage in telephonic conferences. No phone may be used in more than one discussion group at a time. A subscriber may, however, engage in any number of these discussion groups. Each group is uniquely identified by a docket, assigned by the system when the first request for the group is initiated.

A conversation is a set of subscribers, that is, those who are participating in the conversation:

CONVERSATION

$$\mathbb{P}\, SUBSCRIBER$$

A request for a conversation has two components:

REQUEST

$$subscriber : SUBSCRIBER$$
$$conversation : CONVERSATION$$

Component *subscriber* represents who made the request and *conversation* is what was requested.

A connection provided by the telephone system is defined by:

CONNECTION

$$phones : \mathbb{P}\, PHONE$$
$$subscribers : \mathbb{P}\, SUBSCRIBER$$
$$using : SUBSCRIBER \nrightarrow PHONE$$

$$\mathrm{dom}\, using = subscribers$$
$$\mathrm{ran}\, using = phones$$

The *phones* component represents the set of phones that are connected; *subscribers* represents the conversation which the connected phones collectively support and *using* records for each subscriber in a conversation which phone the subscriber is using.

The state of this system is given by

TS

$$sites : SUBSCRIBER \leftrightarrow PHONE$$
$$requests : DOCKET \nrightarrow REQUEST$$
$$connections : DOCKET \nrightarrow CONNECTION$$

$$\mathsf{disjoint}\ (\mathrm{ran}\, connections).phones \land$$
$$(\forall\, d)(\forall\, req)(\forall\, con)$$
$$\quad ((d, req) \in requests \land (d, con) \in connections$$
$$\qquad \longrightarrow con.subscribers \subseteq req.conversation) \land$$
$$\bigcup((\mathrm{ran}\, connections).using) \subseteq sites$$

where *SUBSCRIBER*, *PHONE* and *DOCKET* are basic types. Consider next an expanded version of operation *plug_in*, whereby a subscriber makes himself or herself available at a telephone by plugging in to the telephone:

plug_in

$$sites, sites' : SUBSCRIBER \leftrightarrow PHONE$$
$$requests, requests' : DOCKET \nrightarrow REQUEST$$
$$connections : DOCKET \nrightarrow CONNECTION$$
$$connections' : DOCKET \nrightarrow CONNECTION$$
$$me? : SUBSCRIBER$$
$$phone? : PHONE$$

$$sites' = sites \cup \{me? \mapsto phone?\}$$
$$requests' = requests$$

Note that there is no relationship given between the before and after state values of *connections*. The reason is because operation *plug_in* may possibly initiate a new connection. To illustrate, suppose that subscriber $sub1$ is currently plugged in to phone $ph1$ and wants to talk to subscriber $sub2$. Subscriber $sub1$ attempts to connect to $sub2$ by making a call:

call

$$\Delta\, TS$$
$$me? : SUBSCRIBER$$
$$request? : REQUEST$$
$$docket! : DOCKET$$

$$request?.subscriber = me?$$
$$docket! \notin \mathrm{dom}\, requests$$
$$sites' = sites$$
$$requests' = requests \oplus \{docket! \mapsto request?\}$$

Schema *call* puts in a request on behalf of $sub1$ (i.e. $sub1 = me?$) and if $sub2$ is currently plugged in to a free phone, then the connection is initiated. If, however, subscriber $sub2$ is not plugged in to any free phone, then $sub1$ has to wait until such time as $sub2$ (via operation *plug_in*) plugs in to a free phone (say $ph2$), whereafter the two subscribers are connected to each other (i.e. the state component *connections* is changed).

The above discussion leads us to a revised version of heuristic *#6.0*:

Heuristic #6: Ensure that all operations are honest by

1. listing all preconditions explicitly, and
2. showing the relationship between each changed state component and its after state value, unless such relationship is 'easily provable' from the specification. (Note that the relationship between *connections* and *connections′* is not easily provable since the specification is written at a very abstract level.)

In the spirit of item 2 in heuristic #6 one may be tempted to furthermore require an operation to list all individual postconditions of the operation, simply to make the operation even more honest. This will be valuable to a user since the consequences of an operation would be directly visible. However, this is in general not very feasible since it may lead to overspecification and such consequences are best left as proof obligations to be derived and discharged by the specifier.

3.7 Undo Changes in State Components

Consider next the well-known HCI principle of 'undo' [19, p. 131]:

> Make it possible to reverse actions – to 'undo' them – or make it harder to do what cannot be reversed.

This philosophy suggests the following principle for specification work:

> *Heuristic #7*: Specify an undo counterpart for every operation that changes the state. The idea is to reverse the effect of a state change.

One could argue that heuristic #7 is not really concerned with the actual writing of a formal specification. Nevertheless, if an undo operation is not part of a specification document, then that operation will not be coded into the final software.

Note however:

1. We propose an undo only if it is feasible to do so. For example, if an incorrect value is used in specifying an after state value then we simply 'redo' the operation using the correct value instead of actually 'undoing' the erroneous result.

2. We may have to remember some information in order to specify an undo. For example, suppose we delete an employee record using some key, only to discover that it was the wrong record. For the subsequent undo operation we still have the key available (since it would be communicated back to the user — see heuristic #2 above), but the particulars of the employee (e.g. name, address, etc.) would be lost. To obviate this problem we introduce a component additional to the state space, and call it an *environment*. In the environment we put all auxiliary information, e.g. the detail of a deleted employee.

The use of an environment suggests the following redefinition of (6):

$$\mathbf{f} : Input \times Env \times State \longrightarrow Env \times State \times Output_{\perp} \times Message \qquad (7)$$

3.8 Placing Control Statements

Z allows for the use of an **if then else** construct [28, p. 64] and our next heuristic addresses the question of where in the predicate part of a schema this construct ought to be used. One of the Coad-Yourdon object-oriented guidelines is called 'keep methods simple' [38], and under that heading a claim is made that if the method involves a lot of code or contains IF-THEN-ELSE statements or CASE statements, then it is a strong indication that the method's class has been poorly factored — i.e. procedural code is being used to make decisions that should have been made in the inheritance hierarchy.

For specification work the above guideline translates into limiting control statements to the top level operations (which include our user-level operations). Our primitives therefore do not make any decisions, leading to:

> *Heuristic #8*: Put the control statements in a formal specification as high up in the hierarchy as possible. In particular, put these statements in the top-level operations and not in the primitives.

3.9 Specifying a Control Module

Our last heuristic stems from an observation made of the structure of a number of **Z** specifications in the literature, as well as the golden thread of visibility advocated in [19].

The Established Strategy for presenting a **Z** specification prescribes the use of a table summarising the names of all the partial operations together with their respective inputs, outputs and preconditions [31]. Although such a summary goes a long way in showing when operations are applicable, it does not include error conditions and it also does not entirely show how all the *total* operations are linked together in, for example, an interactive version of such a system. In particular it does not adhere to the visibility concept, coupled with executions and evaluations as stated in [19, p. 140]:

> Narrow the gulfs of execution and evaluation. Make things visible, both for execution and evaluation. On the execution side make the options readily available. On the evaluation side, make the results of each action apparent.

As a first step in adhering to the above call for making the results of an action apparent, we introduce to a specification document a *control module* together with messages to the user of the system, confirming which user-level operation has been chosen.

This leads us to our final design heuristic:

> *Heuristic #9*: Specify a control module which shows when the user-level operations in a formal specification are invoked.

Note how this heuristic again supports heuristic #2, encouraging user communication. The use of a control module is illustrated in Section 5 where we specify a small system according to our enhanced strategy presented next.

4 The Enhanced Strategy

In this section we show how the heuristics developed above can be embedded to enhance the Established Strategy for setting up a **Z** specification.

Algorithm 4.1: Producing an intelligible specification.
Input: A natural language requirements definition.
Output: A **Z** specification adhering to the ES in Section 2 and the heuristics of the previous section.

Method:

***Step* 1:** Define all global constants and basic types. Extend all types from which output is generated to allow for undefined output. Give a natural language description of these types.

***Step* 2:** Present the abstract state space, using the constants and (extended) basic types above.

***Step* 3:** Give an initial state of the system and prove that such a state can be realised.

***Step* 4:** Present the environment, again using the above constants and basic types.

***Step* 5:** Introduce robust definitions for each of the system operations, say **f**, where

$$\mathbf{f} : Input \times Env \times State \longrightarrow Env \times State \times Output_{\perp} \times Message$$

as follows:

5.1 Determine an appropriate sequence of primitives for each robust operation.

5.2 Specify the operation through the primitives and thereafter specify each primitive accordingly. For each primitive explicitly show the condition defining the after state value of the state or environment component to be changed by the primitive. Ensure that primitives are devoid of any control statements.

5.3 Give a short informal description of each robust operation.

***Step* 6:** Determine the precondition of each robust operation on the state and check that the precondition is explicit in the operation's predicate as follows:

6.1 List all preconditions for invoking primitives explicitly.

6.2 In the case that non-determinism is not a specific requirement, check that the calculated precondition is a partition. If not, revisit *Step 5* above and modify the operation accordingly.

6.3 Complete the operation by maximising user communication. This is done by partitioning the larger precondition for the generation of messages into atomic parts.

***Step* 7:** Specify an undo counterpart for every robust operation that changes the state. This is done through the use of primitives as before.

***Step* 8:** Specify the control module which shows when each user-level operation (which is also a robust operation) is invoked.

***Step* 9:** Draw up a table showing all the robust operations together with their inputs, outputs, preconditions for correct operation and error cases.

***Step* 10:** Provide any additional information to assist the reader of the specification, e.g. give a summary of all the robust operations at the end.

Note how the enhanced strategy avoids the use of two problematic schema calculus constructs, namely, schema conjunction ($\wedge$) and disjunction ($\vee$). These constructs are normally used to combine partial operations with success and error schemas to produce robust operations [28]. However, schema conjunction and disjunction are known to occasionally generate ill-formed structures [10, 9].

In the following section we show how the new strategy may be used to draw up a **Z** specification of a familiar library system.

5 A Library System

A specification following the format of the new strategy is constructed. Amongst other things we employ steps 5 and 7 above to illustrate the use of primitives and an undo operation using an environment. In Section 5.8 we show how the use of primitives allows

us to discharge an important proof obligation that arises from the interaction between an operation and its undo.

Consider a library system where users may register, borrow books from the library, and return these at a later date. A book is uniquely identified by an ISBN (for simplicity we assume that our library stocks at most one copy of a book). Other information pertaining to a book includes the title, author, publisher and the year published. A user is uniquely identified by an identity code. Other relevant information includes a user name and an address.

5.1 Basic Types

A simple way to extract the basic types from the small requirements definition above is to underline all nouns in the definition and consider these for becoming basic types. Care has to be taken, however, not to identify a subset of another set, or an aggregate made up of other basic types as additional basic types.

Our basic types are:

$$[ISBN, \text{Title}, \text{Author}, \text{Publisher}, \text{Year}, \text{ID}, \text{Name}, \text{Address}, Option_\perp, Message]$$

The set $Option_\perp$ is not evident from the above requirements but we use it in the specification of our control module, namely, *LibCntrl* in Section 5.4 below. Similarly, the set *Message* is a standard type used in our specifications (see Step 6.3 of the enhanced strategy).

At this stage a natural language description of the above types would normally be given. However, to save space we rely on the reader's intuition for a grasp of these types.

5.2 State Space and Initial State

The state of the library is given by

Library

$$\begin{aligned} &books : ISBN \pfun Title \times Author \times Publisher \times Year \\ &users : ID \pfun Name \times Address \\ &available : \mathbb{P}\, ISBN \\ &borrowed : ISBN \pfun ID \\ &date : ISBN \pfun Date \\ \hline &available \cup \operatorname{dom} borrowed \subseteq \operatorname{dom} books \\ &available \cap \operatorname{dom} borrowed = \varnothing \end{aligned}$$

We assume that the library contains reference works that are available but cannot be borrowed by a user, hence the use of a subset ($\subseteq$) relationship instead of equality in the first predicate above.

An initial state is defined as a library with no books, no users and an empty date set:

InitLibrary

$$\begin{aligned} &Library' \\ \hline &books' = \varnothing \land users' = \varnothing \land date' = \varnothing \end{aligned}$$

A proof obligation arises from the initial state:

$$\vdash \exists\, Library' \bullet InitLibrary \tag{8}$$

Formula (8) claims that a state can be realised such that it satisfies the requirements of *InitLibrary*. Theorem 8 is easily discharged by the popular and widely used first-order, resolution-based theorem prover, OTTER [14, 37], provided we give to the reasoner the following ZF [7] axiom describing the existence of an empty set:

$$(\exists\, B)(\forall\, x)\,(x \notin B) \tag{9}$$

5.3 Definition of the Environment

A user and a book is uniquely identified by an identity code and an ISBN respectively. Therefore, we need to remember an identity code or an ISBN in the case of a subsequent undo operation. Since the requirements do not specify the removal of books or users from the system, we need not keep the detail of a deleted book or user in the environment. Therefore, our environment is defined as:

LibEnv

$$\begin{aligned} &id : ID \\ &isbn : ISBN \end{aligned}$$

5.4 Library Control Module

Our enhanced strategy prescribes the definition of a control module to show when the user-level operations are invoked. User-level operations to register a new user (say *Register*), borrow a book (*Borrow_Book*) and return a book (*Return_Book*) are evident from the above natural language description. An operation to initialise the system is normally done at system creation only, hence we assume it is not accessible to any user thereafter.

First, we define a *control environment* with a single variable to, in the case of an undo, remember the previous operation selected.

CEnv

$$previous : Option_\perp$$

Variable *previous* is a member of an extended type, since initially no previous operation exists and in our model an undo operation cannot be preceded by another undo, therefore it may be undefined. Our control module is:

LibCntrl

$\Delta CEnv$
$choice? : Option_{\perp}$
$mes! : Message$

$previous' = choice? \land$
$mes! =$ **if** $choice? =$ `Register` **then**
Operation *Register* selected
else if $choice? =$ `Borrow` **then**
Operation *Borrow_Book* selected
else if $choice? =$ `Return` **then**
Operation *Return_Book* selected
else if $choice? =$ `Undo` **then**
if $previous =$ `Register` **then**
Undo previous Register operation
else if $previous =$ `Borrow` **then**
Undo previous Borrow operation
else if $previous =$ `Return` **then**
Undo previous Return operation
else Previous Undo operation invalid
else Invalid selection

The hard coding of the names of the operations in the control module is simply a matter of taste since a specifier may prefer to use a table-driven or parameterised approach to determine the operation selected by the user. For example, to test for a normal operation we can specify $choice? \in \{\texttt{Register}, \texttt{Borrow}, \texttt{Return}\}$ instead of using 3 different predicates. Also, the use of '**else if**' in this way is non-standard in **Z** but the use of an **if** P **then** E_1 **else** E_2 construct is [28].

Next we specify one of our user-level operations.

5.5 A User-Level Operation

Schema *Borrow_Book* below issues a book to a user.

Borrow_Book

$\Delta Library;\ \Delta LibEnv$
$id? : ID;\ isbn? : ISBN$
$mes! : Message$

$\theta LibEnv' =$
if $id? \in \mathrm{dom}\ users \land isbn? \in available$ **then**
$Env\_id(id?, Env\_isbn(isbn?, \theta LibEnv))$
else $\theta LibEnv$
$\theta Library' =$
if $id? \in \mathrm{dom}\ users \land isbn? \in available$ **then**
$Borrow(UnAvail(\theta LibEnv', \theta Library))$
else $\theta Library$
$mes! =$
if $id? \in \mathrm{dom}\ users \land isbn? \in available$ **then**
Book $isbn?$ borrowed by user $id?$
else if $id? \notin \mathrm{dom}\ users \land isbn? \in available$ **then**
Invalid user $id?$ but book $isbn?$ available
else if $id? \in \mathrm{dom}\ users \land isbn? \notin available$ **then**
Book $isbn?$ unavailable but user $id?$ valid
else Invalid user $id?$ and book $isbn?$ unavailable

Borrow_Book is designed in line with steps 5 and 6 of the enhanced strategy, since it:

- follows the layout given for a robust operation,
- presents the preconditions as partitions,
- is built up through a sequence of primitives to specify after state values for state and environment components, and explicitly shows the condition under which the primitives are invoked,
- places the control statements in the schema and not in the primitives,
- maximises communication with the user.

5.6 Definition of Primitives

The primitives in *Borrow_Book* are defined using ordinary set theory. *Env_isbn* places an isbn in the environment to be used in the event of an undo and *Env_id* performs a similar function for a user id.

Primitive *Env_isbn*

$Env\_isbn : ISBN \times LibEnv \longrightarrow LibEnv$
is given by
$Env\_isbn(isbn?, env) = env'$, where
$env'.isbn = isbn?$

Primitive *Env_id*

$Env\_id : ID \times LibEnv \longrightarrow LibEnv$
is given by
$Env\_id(id?, env) = env'$, where
$env'.id = id?$

Primitive *UnAvail* makes a book unavailable while *Borrow* issues the book to a user:

Primitive *UnAvail*

$UnAvail : LibEnv \times Library \longrightarrow LibEnv \times Library$
is given by
$UnAvail(env, library) = (env, library')$, where
$library'.available = library.available - \{env.isbn\}$

Primitive *Borrow*

$Borrow : LibEnv \times Library \longrightarrow Library$
is given by
$Borrow(env, library) = library'$, where
$library'.borrowed =$
$library.borrowed \cup \{(env.isbn, env.id)\}$

5.7 Definition of an Undo

According to step 7 of the enhanced strategy we specify an undo counterpart for *Borrow_Book* as follows:

Undo_Borrow_Book

$\Delta Library;\ \Xi LibEnv$
$mes! : Message$

$\theta Library' =$
$Undo\_UnAvail(Undo\_Borrow(\theta LibEnv, \theta Library))$
$mes! =$ Previous borrow operation reversed

Primitive *Undo_Borrow* reverses the effect of *Borrow*:

Primitive *Undo_Borrow*

$Undo\_Borrow : LibEnv \times Library \longrightarrow LibEnv \times Library$
is given by
$Undo\_Borrow(env, library) = (env, library')$, where
$library'.borrowed = \{\, env.isbn \,\} \lhd library.borrowed$

Primitive *Undo_UnAvail* is specified in a similar way.

The ease whereby a proof obligation arising from a specification may be discharged is often a good indication of the usefulness of the specification [32]. In the next section we show how our use of primitives allows an automated reasoner to obtain a short proof of a property where a proof is otherwise not easily obtained.

5.8 Discharging a Proof Obligation

Suppose an unregistered donor donates a new book to the library and thereby becomes a registered user. In [21], the following traditional schema describing this operation is given:

Donate

$\Delta Library;\ \Delta LibEnv$
$isbn? : ISBN;\ id! : ID_\perp$
$title? : Title;\ aut? : Author;\ pub? : Publisher$
$yr? : Year;\ name? : Name;\ addr? : Address$
$mes! : Message$

$(id! \notin \mathrm{dom}\, users \land isbn? \notin \mathrm{dom}\, books \land$
$\quad isbn = isbn? \land id = id! \land$
$\quad books' =$
$\quad\quad books \cup \{isbn? \mapsto (title?, aut?, pub?, yr?)\} \land$
$\quad users' = users \cup \{id! \mapsto (name?, addr?)\} \land$
$\quad available' = available \cup \{isbn?\} \land$
$\quad borrowed' = borrowed \land date' = date \land$
$\quad mes! = OK)$
$\quad\quad \lor$
$((\mathrm{dom}\, users = ID \lor isbn? \in \mathrm{dom}\, books) \land$
$\quad \theta LibEnv' = \theta LibEnv \land$
$\quad \theta Library' = \theta Library \land$
$\quad id! = \perp \land mes! = System\ error)$

An important proof obligation often stated in **Z** texts is to show that an operation followed by its undo counterpart leaves the state unchanged, i.e.:

$$Donate \fatsemi Donate^{\sim} \vdash \Xi Library \qquad (10)$$

The OTTER reasoner has difficulty in proving (10) above, but if we rewrite schema *Donate* as a sequence of 3 primitives operations

- *Capture_book* (say), a primitive to specify
  $books' =$
  $\quad books \cup \{isbn? \mapsto (title?, aut?, pub?, yr?)\}$,
- *Register_user* (say), to specify
  $users' = users \cup \{id! \mapsto (name?, addr?)\}$,
- *Avail* (say), for $available' = available \cup \{isbn?\}$,

specify appropriate undo counterparts for each of the 3 primitives above, and perform 3 different proofs at the level of the primitives and their inverses, then OTTER easily finds a proof (for example) for

$$Capture\_book \fatsemi Capture\_book^{\sim} \vdash books' = books \qquad (11)$$

where *books* represents the component before primitive *Capture_book* and *books'* the same component after $Capture\_book^{\sim}$. A successful proof attempt of (11) is presented in the appendix.

Quick proofs are also obtained for:

$$Register\_user \fatsemi Register\_user^{\sim} \vdash users' = users$$
$$Avail \fatsemi Avail^{\sim} \vdash available' = available$$

Failing to prove (10) above is more significant than it may seem. A specifier may decide to leave schema *Donate* as it is and attempt to perform 3 different simpler proofs, one of which could be:

$$Donate \fatsemi Donate^{\sim} \vdash books' = books \qquad (12)$$

Again the theorem-prover fails to find a proof of (12). While a proof of (11) is found after just *0.06 seconds* on a Pentium IV running at a clock speed of 1.8 GHz, the theorem prover finds no proof for (12) in 30 minutes. The architecture of this last failed proof attempt is characterised by the presence of redundant information [33] in the sense that changes to the state components *users* and *available* in schema *Donate* are irrelevant to a proof of (12). It turns out that such irrelevant information leads the theorem prover astray. Of course, a specifier can remove redundant information from a proof attempt and perform a number of different proofs. But this boils down to an application of steps 5.1 and 5.2 of our enhanced strategy, i.e. defining an operation as a sequence of primitives.

5.9 Summary

We demonstrated in the previous section how the enhanced strategy may be used to construct a formal specification and how our strategy facilitates the discharging of an important proof obligation.

6 Analysis and Future Work

This paper proposed an enhanced established strategy for drawing up a **Z** specifiction. The new strategy is built around the notion of a set of design heuristics, mainly for enhancing schema content. Central to the new strategy is the use of primitives, allowing a specifier to specify a robust operation as a sequence of simpler operations, each possibly manipulating at most one schema component. We illustrated how the use of primitives facilitates the task of finding a short proof of a property of a composition where otherwise the presence of redundant information leads the theorem prover astray.

In defining a sequence of primitives in a user-level operation one may find that such a sequence sometimes becomes too long to comfortably write down on a single line. We can use more than one line, but a specifier may also consider grouping primitives which, according to the judgement of the specifier, conceptually belong together into a *master* operation [29] and instead replace such sequence in the user-level operation by the master operation. The definitions of the individual primitives are not affected.

Although it is often claimed that implementation issues should not affect specification decisions, additional implementation benefits may be realised through the use of primitives. Since every primitive manipulates at most one component of the state or environment, we can, on a multi-processor machine, assign a processor to a primitive and if some of the primitives happen to be independent, then we may achieve true concurrency. Furthermore, on a threaded single-processor machine we can program a user-level operation as a task and each primitive as a thread [25] within the task. If a thread should block during execution then the possibility exists for another thread in the same task to start executing, speeding up the execution of the task as a whole.

Step 4 of the enhanced strategy suggests the use of an environment in specification work. The environment is often used to reverse the effect of a state change. However, in our model it is possible to undo only the effect of the *previous* update operation, and not any other (update) operation before the last one. Modern software packages normally allow one to undo a number of previous operations, one after another. Typically in our model a sequence of environments could be used for this purpose and future work could concentrate on developing such sequences of environments.

Acknowledgement

We wish to thank Willem Labuschagne in the Department of Computer Science, University of Otago, New Zealand for his valuable input during the initial stages of this research.

References

[1] A. Bahrami. *Object-Oriented Systems Development using the Unified Modelling Language*. McGraw-Hill, 1999.

[2] R. Barden, S. Stepney, and D. Cooper. ***Z** in Practice*. Prentice-Hall, London, UK, 1994.

[3] L. C. Briand, S. Morasca, and V. R. Basili. Defining and Validating High-Level Design Metrics. Technical Report CS-TR-3301, College Park, MD: University of Maryland, Department of Computer Science, 1994.

[4] S. R. Chidamber and C. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476 – 493, June 1994.

[5] B. Collins, J. Nicholls, and I. Sørensen. Introducing Formal Methods: The CICS Experience with **Z**. In B. De Neumann, D. Simpson, and G. Slater, editors, *Mathematical Structures for Software Engineering*, pages 153 – 164. Clarendon Press, Oxford, 1991.

[6] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice-Hall, London, UK, 2nd edition, 1998.

[7] H. Enderton. *Elements of Set Theory*. Academic Press, Inc., 1977.

[8] A. M. Gravell. What is a Good Formal Specification? In J. Nicholls, editor, ***Z** User Workshop: Proceedings of the Fifth Annual **Z** User Meeting, Oxford, U.K., 17 - 18 December 1990*, pages 137 – 150. Springer-Verlag, 1991.

[9] I. Hayes. Interpretations of **Z** Schema Operators. In J. Nicholls, editor, ***Z** User Workshop: Proceedings of the Fifth Annual **Z** User Meeting, Oxford, U.K., 17 - 18 December 1990*, pages 12 – 26. Springer-Verlag, 1991.

[10] M. Jackson. Problem Frames and Principles of Description. Course notes: WoFACS'98, Cape Town, South Africa, July 1998.

[11] M. A. Jackson. *Principles of Program Design*. Academic Press, London, UK, 1975.

[12] T. Love. Timeless Design of Information Systems. *Object Magazine*, page 46, November-December 1991.

[13] D. J. Mayhew. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[14] W. W. McCune. *OTTER 3.3 Reference Manual*. Argonne National Laboratory, Argonne, Illinois, August 2003. ANL/MCS-TM-263.

[15] C. C. Morgan. Specification of a Communication System. In Y. Paker and J.-P. Verjus, editors, *Distributed Computing Systems: Synchronisation, Control, and Communication*, pages 93 – 108. Academic Press, 1983.

[16] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 2nd edition, 1994.

[17] C. C. Morgan and B. Sufrin. Specification of the Unix Filing System. In I. Hayes, editor, *Specification Case Studies*, pages 45 – 78. Prentice-Hall, London, UK, 2nd edition, 1993.

[18] J. Nielsen. Ten usability heuristics. Technical Report www.useit.com/papers/heuristics, UseIt, 2001.

[19] D. A. Norman. *The Design of Everyday Things*. The MIT Press, 1998.

[20] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice-Hall, New Jersey, USA, 1998.

[21] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and **Z***. Prentice-Hall, London, UK, 2nd edition, 1996.

[22] J. Preece, H. Sharp, and H. Rogers. *Interaction Design: Beyond Human-Computer Interaction*. John Wiley & Sons, 2002.

[23] T. Scheurer. *Foundations of Computing : System Development with Set Theory and Logic*. International Computer Science Series. Addison-Wesley, University Press, Cambridge, 1994.

[24] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[25] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, New York, USA, 7th edition, 2005.

[26] A. Spalter. *The Computer in the Visual Arts*. Addison-Wesley, 1999.

[27] J. Spivey. *Understanding **Z**: A Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1988.

[28] J. Spivey. *The **Z** Notation: A Reference Manual*. Prentice-Hall, London, UK, 2nd edition, 1992.

[29] J. A. van der Poll. *Automated Support for Set-Theoretic Specifications*. PhD thesis, University of South Africa, June 2000.

[30] J. A. van der Poll and P. Kotzé. What design heuristics may enhance the utility of a formal specification? In P. Kotzé, L. Venter, and J. Barrow, editors, *Enablement Through Technology - Research Conference of SAICSIT*, pages 179 – 194, Port Elizabeth, South Africa, September 2002. ACM International Conference Proceedings.

[31] J. A. van der Poll and P. Kotzé. A Multi-level Marketing Case Study: Specifying Forests and Trees in Z. *South African Computer Journal*, Issue 30:17 – 28, June 2003.

[32] J. A. van der Poll, P. Kotzé, and W. A. Labuschagne. Automated Support for Enterprise Information Systems. *Journal of Universal Computer Science*, 10(11):1519 – 1539, November 2004.

[33] J. A. van der Poll and W. A. Labuschagne. Heuristics for Resolution-Based Set-Theoretic Proofs. *South African Computer Journal*, Issue 23:3 – 17, July 1999.

[34] J. Woodcock and J. Davies. *Using **Z**: Specification, Refinement, and Proof*. Prentice-Hall, London, UK, 1996.

[35] J. Wordsworth. Practical experience of formal specification: a programming interface for communications. In J. McDermid and C. Ghezzi, editors, *2nd European Software Engineering Conference* (*ESEC*), pages 140 – 158. University of Warwick, Coventry, UK, Springer-Verlag, 1989.

[36] J. Wordsworth. A **Z** Development Method. Draft version 0.11. Technical report, IBM, UK, Hursley Park, January 1989.

[37] L. Wos. Programs that Offer Fast, Flawless, Logical Reasoning. *Communications of the ACM*, 41(6):87 – 95, June 1998.

[38] E. Yourdon. *Object-Oriented Systems Design: An Integrated Approach*. Yourdon Press, Englewood Cliffs, New Jersey, 1994.

[39] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, N.Y, 1978.

Appendix

Input to OTTER for a proof of: $Capture\_book \fatsemi Capture\_book^{\sim} \vdash books' = books$

```
set(hyper_res). %% Resolution strategies
set(factor).    %% and factoring
set(ur_res).

set(para_from).  %% Paramodulation
set(para_into).  %% and
set(order_eq).   %% ordering of equalities

set(process_input).
clear(para_from_right).
clear(para_into_right).
set(dynamic_demod_all).
set(back_demod).

assign(max_seconds,1800).  %% Assign 30 mins.

weight_list(pick_and_purge).
weight(x,5).
end_of_list.

formula_list(usable).

%% Reflexivity.
%% ------------
(all x (x = x)).

%% Definition of domain.
%% ---------------------
(all R x ( El(x,dom(R)) <-> (exists y El(ORD(x,y),R)) )).

%% Primitive Capture_book.
%% -----------------------
(all isbn title author pub yr
     e_id e_isbn books users available borrowed date
     e_id1 e_isbn1 books1 users1 available1 borrowed1 date1
 ( El(ORD(7TUP(isbn,title,author,pub,yr,ENV(e_id,e_isbn),
               STATE(books,users,available,borrowed,date)),
          ORD(ENV(e_id1,e_isbn1),
               STATE(books1,users1,available1,borrowed1,date1))),
      Capture_book) ->
  (
   -El(isbn,dom(books)) &
   (e_isbn1 = isbn) &

%% books1 = books u {isbn |-> (title,author,pub,yr)}.
%% -------------------------------------------------
   (all y z
    (El(ORD(y,z),books1) <->
     (El(ORD(y,z),books) |
      ((y = isbn) & (z = 4TUP(title,author,pub,yr)))))) &

   (users1 = users) &
   (available1 = available) &
   (borrowed1 = borrowed) &
   (date1 = date) ) )).
```

```
%% Primitive Undo_Capture_book.
%% ----------------------------
(all e_id e_isbn books users available borrowed date
     e_id1 e_isbn1 books1 users1 available1 borrowed1 date1
 ( El(ORD(ORD(ENV(e_id,e_isbn),
              STATE(books,users,available,borrowed,date)),
          ORD(ENV(e_id1,e_isbn1),
              STATE(books1,users1,available1,borrowed1,date1))),
       Undo_Capture_book) ->

  ((e_id1 = e_id) & (e_isbn1 = e_isbn) &

%% books1 = DomDiff(books,{e_isbn}).
%% ---------------------------------
   (all y z
    (El(ORD(y,z),books1) <->
     (El(ORD(y,z),books) & -(y = e_isbn)))) &

   (users1 = users) & (available1 = available) &
   (borrowed1 = borrowed) &
   (date1 = date) ) )).

%% Definition of: Capture_book ; Undo_Capture_book.
%% -----------------------------------------------
(all isbn title author pub yr
 e_id e_isbn books users available borrowed date
 e_id2 e_isbn2 books2 users2 available2 borrowed2 date2
( El(ORD(7TUP(isbn,title,author,pub,yr,ENV(e_id,e_isbn),
              STATE(books,users,available,borrowed,date)),
         ORD(ENV(e_id2,e_isbn2),
             STATE(books2,users2,available2,borrowed2,date2))),
     Comp(Capture_book,Undo_Capture_book)) ->

  (exists e_id1 e_isbn1 books1 users1 available1 borrowed1 date1
  ( El(ORD(7TUP(isbn,title,author,pub,yr,ENV(e_id,e_isbn),
               STATE(books,users,available,borrowed,date)),
           ORD(ENV(e_id1,e_isbn1),
               STATE(books1,users1,available1,borrowed1,date1))),
       Capture_book) &

    El(ORD(ORD(ENV(e_id1,e_isbn1),
               STATE(books1,users1,available1,borrowed1,date1)),
           ORD(ENV(e_id2,e_isbn2),
               STATE(books2,users2,available2,borrowed2,date2))),
       Undo_Capture_book) )) )).

end_of_list.

formula_list(sos).

%% books'' = books.
%% ----------------
-(all isbn title author pub yr e_id e_isbn books users available borrowed date
      e_id2 e_isbn2 books2 users2 available2 borrowed2 date2
  ( El(ORD(7TUP(isbn,title,author,pub,yr,ENV(e_id,e_isbn),
               STATE(books,users,available,borrowed,date)),
           ORD(ENV(e_id2,e_isbn2),
               STATE(books2,users2,available2,borrowed2,date2))),
    Comp(Capture_book,Undo_Capture_book)) ->
     (all y z (El(ORD(y,z),books2) <-> El(ORD(y,z),books))) ) ).

end_of_list.
```