Towards Design Guidelines for Constructing a Formal Specification

John A. van der Poll\textsuperscript{1} and Paula Kotzé\textsuperscript{2}

Department of Computer Science and Information Systems
University of South Africa
\textsuperscript{1}vdpolj@unisa.ac.za \textsuperscript{2}kotze@unisa.ac.za

\textbf{Abstract.} Accepted software engineering design principles are well established, but design principles for constructing a formal specification have been relatively rare. In this paper we examine a number of formal specifications written in $\mathbf{Z}$ 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 guidelines for the construction of a formal specification. The purpose of these guidelines is to incorporate general design principles as well as those often used in the final software, already at the specification phase. We illustrate how one of these guidelines, 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.

1 Introduction

Software engineering design principles have been around for a couple of decades, dating back to the early work on \textit{structured design} (e.g. [1], [2] and [3]). In many instances the object-oriented design methodology replaced the earlier structured techniques as the preferred method of design. Numerous notational representations for object-oriented design were introduced, e.g. Coad et al. [4], Rumbaugh et al. [5], Booch [6] and Shlaer et al. [7].

Guidelines for object-oriented software design have been proposed by Yourdon [8], Love [9] and Coad et al. [10] while general measures for high-level design were formulated by Briand et al. [11]. Human Computer Interaction (HCI) design principles are presented in Dix et al. [12] (amongst others), while principles for general design are developed in Norman [13].

Design principles for drawing up a formal specification have been relatively sparse. Gravell [14] proposes a number of principles for constructing a specification written in $\mathbf{Z}$ (e.g. [15]), mainly to make such specifications more readable.

1.1 Why Design Guidelines?

The question of why a set of guidelines for constructing a formal specification might be useful, boils down to asking what the specification is to be used for.
A formal specification is often used as the starting point of a subsequent refinement phase Morgan [16], and a well-designed specification could possibly be more easily refined to code than an ad hoc specification. One of the advantages 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 (e.g. Woodcock et al. [17]). For instance we can show that certain undesirable properties are absent from the specification.

A specification constructed according to established design criteria could be more comprehensible to users, easier to enhance and therefore facilitate long-term maintenance, especially if principles embodied in the final software are already used at the specification phase.

Although the Established Strategy (e.g. Barden et al. [18]) goes a long way towards presenting a Z specification in an intelligible way, established software design principles (e.g. cohesion [8]) are not officially part of this strategy; neither is the use of certain general design principles (e.g. make things visible to the user [13]).

In this paper we propose a number of guidelines to support the construction of a formal specification. The aims behind these guidelines are to:

1. Incorporate some established software engineering design principles normally present in the final product already during 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 proof.

1.2 Structure of this Paper

Section 2 introduces a preliminary set of guidelines aimed at enhancing the utility of a specification. The definition of these guidelines is based on an analysis of a number of Z specifications, some software engineering and general design principles, and the consideration of some initial refinement stages. In Sect. 3 we build a small specification using some of the principles proposed below. We conclude with ideas about future work and applicability of our guidelines.

2 Design Guidelines

2.1 Format of a Precondition

Consider the state $FS$ of a UNIX-like filing system. A detailed discussion of $FS$ appears in Morgan & Sufrin [19, pp. 45 - 78]. We will use this state definition and the ‘open file’ operation below to illustrate our first three guidelines.

In $FS$, $FID$ (a set of file identifiers), $CID$ (a set of channel identifiers), and $SYL$ (a set of syllables) are all basic types; $FILE = \text{seq \mbox{\textit{BYTE}}}$, for $\text{BYTE} = 0 \ldots 255$; $NAME = \text{seq \mbox{\textit{SYL}}}$ (i.e. a sequence of syllables); and $\text{CHAN}$ is given by:
\[\text{CHAN} \]
\[
\begin{array}{l}
\text{fid} : \text{FID} \\
\text{posn} : \text{N}
\end{array}
\]

\[\text{FS} \]
\[
\begin{array}{l}
\text{fstore} : \text{FID} \rightarrow \text{FILE} \\
\text{cstore} : \text{CID} \rightarrow \text{CHAN} \\
\text{nstore} : \text{NAME} \rightarrow \text{FID} \\
\text{dnames} : \mathbb{P} \text{NAME}; \text{usedfids} : \mathbb{P} \text{FID}
\end{array}
\]
\[
\begin{array}{l}
\text{Front} \{ \text{dnames} \cup \text{dom nstore} \} \subseteq \text{dnames} \\
\text{usedfids} = \text{ran nstore} \cup \{ \text{chan} : \text{ran cstore} \bullet \text{chan.fid} \} \\
\text{usedfids} \subseteq \text{dom fstore}
\end{array}
\]

An operation to open a file is given by:

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

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

The partial preconditions of operation \text{open} are:

\[
\begin{array}{l}
\text{(name}? \in \text{dom nstore} \land \text{cid!} \notin \text{dom cstore}) \quad (1) \\
\text{(name}? \notin \text{dom nstore}) \quad (2) \\
\text{(dom cstore} = \text{CID}) \quad (3)
\end{array}
\]

The total precondition of \text{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 the following (incorrect) structure:

\[
\text{if } \text{precondition1 then } S1 \\
\text{elseif } \text{precondition2 then } S2 \\
\text{elseif } \text{precondition3 then } S3 \\
\text{endif}
\]

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

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

### 2.2 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. \(\text{CID} = \text{dom cstore}\)) and the input file name, \(\text{name}\), is incorrect (i.e. \(\text{name} \notin \text{dom nstore}\)). Suppose also that owing to the above non-determinism, the message ‘\(\text{NoFreeCids}\)’ is displayed, telling 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, only to be faced with the message ‘\(\text{NoSuchName}\)’. One could argue that this message should have been displayed together with the message about the channel, so that the user could have fixed the problem in the meantime, instead of having to wait for a free channel. This leads to our second design guideline:

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

The above guideline agrees with the following principle proposed by Donald Norman [13, p. 140]:

‘Narrow the gulls of execution and evaluation. Make things visible, both for execution and evaluation’.

### 2.3 Signature of an Operation

Next, we propose a *preliminary* version of our third guideline, derived from the structure of total operations in Z. Often operations accept as domain elements the state and external input to deliver as range elements the state and additional output (e.g. \(\text{open}\) above). In the light of guideline #2 above we notationally separate the message from other output as formulated in the following design guideline for a *user-level* operation, i.e., an operation which communicates with the user.

**Guideline #3:** Define every user-level operation, say \(\text{f}\), based on the general format:

\[
\text{f} : \text{Input} \times \text{State} \rightarrow \text{State} \times \text{Output} \times \text{Message}
\]

(5)
Definition (5) is stated in a preliminary form. Further guidelines below will refine this definition.

2.4 Value of Undefined Output

Consider the following definition of a simple file system (Woodcock et al. [17]) where Key and Data are basic types.

\[
\begin{array}{l}
\text{File} \\
\quad \text{contents : Key} \rightarrow \text{Data}
\end{array}
\]

A robust operation to read a file is:

\[
\begin{array}{l}
\text{Read} \\
\quad \text{contents, contents' : Key} \rightarrow \text{Data} \\
\quad \text{k? : Key} \\
\quad \text{d! : Data} \\
\quad \text{r! : Message} \\
\quad \begin{cases}
\quad (\text{k?} \in \text{dom contents} \land \text{d!} = \text{contents k?} \land \text{contents'} = \text{contents} \land \\
\quad \text{r!} = \text{ok}) \lor \\
\quad (\text{k?} \notin \text{dom contents} \land \text{contents'} = \text{contents} \land \\
\quad \text{r!} = \text{key not in use})
\end{cases}
\end{array}
\]

Note that \(d!\) is unspecified under the error condition \(k? \notin \text{dom contents}\). Woodcock and Davies [17, 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 \(\text{contents k}\), for any \(k \in \text{dom contents}\) which is undesirable.

Instead, we could specify that the value of an output variable like \(d!\) above is \textit{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, much like the technique used in the semantics of programming languages (e.g. Schmidt [20, page 29]). If we denote an undefined value by \(\bot\), then we extend the set \(\text{Data}\) to \(\text{Data}_\bot = \text{Data} \cup \{\bot\}\).

This observation leads to:

\textit{Guideline \#4:} Ensure that all sets from which output may be generated are extended to allow for undefined values.

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

\[
\text{f : Input} \times \text{State} \rightarrow \text{State} \times \text{Output}_\bot \times \text{Message}
\]

In line with guideline \#4, we make the undefined nature of \(d!\) explicit in the last disjunct in operation \(\text{Read}\) by adding \(d! = \bot\). We also replace the declaration \(d! : \text{Data}\) with \(d! : \text{Data}_\bot\).
2.5 Function Application

It is normally a trivial task to obtain a corresponding range element, given a function definition and a domain element, since we simply apply the function to the domain element. However, if we are given an element from the range, then it could be hard to obtain corresponding domain elements — we have to use the inverse of the function which is a relation in general. This problem is aggravated if we are presented with a second coordinate of a pair, where such a pair is the second coordinate of an enclosing, larger tuple.

Such a scenario is found in the classic specification of a telephone conference system by Carroll Morgan [21]:

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 who participate in the conversation:

\[
\text{CONVERSATION} \ni \text{SUBSCRIBER}
\]

A request for a conversation has two components:

\[
\text{REQUEST}
\]

\[
\begin{align*}
\text{subscriber} : & \text{SUBSCRIBER} \\
\text{conversation} : & \text{CONVERSATION}
\end{align*}
\]

Component \text{subscriber} represents who made the request, and \text{conversation} is what was requested.

A connection provided by the telephone system is defined by the schema:

\[
\text{CONNECTION}
\]

\[
\begin{align*}
\text{phones} : & \text{PHONE} \\
\text{subscribers} : & \text{SUBSCRIBER} \\
\text{using} : & \text{SUBSCRIBER} \rightarrow \text{PHONE}
\end{align*}
\]

\[
\begin{align*}
\text{dom using} = & \text{subscribers} \\
\text{ran using} = & \text{phones}
\end{align*}
\]

The set \text{phones} denotes the set of phones that are connected; the set \text{subscribers} represents the conversation which the connected phones collectively support; \text{using} records for each subscriber in a conversation which phone he or she is using.

The abstract state space is given by schema \text{TS}, where \text{SUBSCRIBER}, \text{PHONE} and \text{DOCKET} are basic types:
TS
sites: SUBSCRIBER → PHONE
requests: DOCKET → REQUEST
connections: DOCKET → CONNECTION

\[
\text{disjoint (ran connections).phones} \land \\
(\forall d)(\forall req)(\forall con) \\
((d, req) \in \text{requests} \land (d, con) \in \text{connections}) \\
\rightarrow \text{con. subscribers} \subseteq \text{req. conversation}) \land \\
\bigcup ((\text{ran connections).using}) \subseteq \text{sites}
\]

Operation *ding_dong*, defined by Morgan [21], takes a phone as input and returns the docket associated with the conversation which the phone is part of.

\[
\text{ding_dong} \equiv TS \\
\text{phone?} : \text{PHONE} \\
\text{docket!} : \text{DOCKET} \\
\text{phone?} \in \text{connections(docket!).phones} \tag{D1}
\]

Predicate (D1) above suggests a rather complex algorithm to be generated during refinement: The component docket! is not available beforehand, but only after the *ding_dong* operation while the only input, phone?, is a range element.\(^1\)

The above complication arises because input is accepted from the range of a function, and we could simplify the problem of finding the desired domain element by adding a component to TS that takes phone? as input.

This leads us to the following guideline:

*Guideline #5:* Where appropriate, ensure that no input involving a function is accepted solely as an element of the range of the function.

### 2.6 Cohesion

Our next guideline stems from the well-known software engineering principle of *cohesion*. Bahrami [23] 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 [8] 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* (see e.g. Pfleeger [24]), which is the kind of cohesion described by Yourdon [8] and which we advocate in the design of a formal specification. The natural language

\(^1\) This specification style is related to the idea of *equal opportunity* ([22]) whereby something displayed by the system can be used by the user as input.
definition given by Yourdon [8] 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. Thierry Scheurer [25] puts forward the following thesis: 'Set theory, based on logic, is a universal language in which all problems may be formulated and solved.' Since the \( Z \) specification language is based on first-order logic and a strongly typed fragment of Zermelo-Fraenkel set theory, 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 guideline:

\textbf{Guideline \#6:} 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 Sect. 3 below. Guideline \#6 has an important benefit when reasoning about the properties of a specification: In Sect. 3.6 we show how this guideline facilitates an important proof obligation that arises from the interaction between one primitive and another primitive which reverses the effect of the first primitive.

### 2.7 Undo Changes in State Components

For our next design guideline we turn to the well-known HCI principle of 'undo' as advocated by Donald Norman [13, p. 131]:

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

The above philosophy suggests the following guideline for specification work:

\textbf{Guideline \#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 the above principle 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 a calculation 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 design guideline #2 above), but the particulars of the employee (e.g. name, address, etc.) would be lost. To obviate this problem we propose 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):

\[ f : Input \times Env \times State \rightarrow Env \times State \times Output \times Message \quad (7) \]

### 2.8 Placing Control Statements

One of the Coad-Yourdon [8] object-oriented guidelines is called ‘Keep methods simple’, 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 methods’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 our final guideline:

**Guideline #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.

To illustrate some of these guidelines, a small specification is constructed in Sect. 3 below.

### 3 A Library System

In this section we construct a small specification based on the above guidelines. Amongst other things we illustrate the use of primitives, an undo operation utilising an environment, and in Sect. 3.6 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 them at a later date. A book is uniquely identified by an ISBN\(^2\). 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 address.

---

\(^2\) For simplicity we assume that the library stocks at most one copy of a book.
3.1 Definition of the State Space

The state of the library is given by

\[
\text{Library} \\
\begin{array}{c}
\text{books : } ISBN \rightarrow \text{Title } \times \text{Author } \times \text{Publisher } \times \text{Year} \\
\text{users : } ID \rightarrow \text{Name } \times \text{Address} \\
\text{available : } \mathbb{P} ISBN \\
\text{borrowed : } ISBN \rightarrow ID \\
\text{date : } ISBN \rightarrow Date
\end{array}
\]

\[
\begin{array}{c}
\text{available } \cup \text{dom borrowed } \subseteq \text{dom books}^3 \\
\text{available } \cap \text{dom borrowed } = \emptyset
\end{array}
\]

where ISBN, Title, Author, Publisher, Year, ID, Name and Address are all basic types.

3.2 Definition of the Environment

Our system is concerned with books borrowed by users. A user is uniquely identified by an identity code and a book is uniquely identified by an ISBN. Therefore, we need to remember an identity code or an ISBN in the case of a subsequent undo operation\(^4\). Our environment is defined as:

\[
\text{LibEnv} \\
\begin{array}{c}
\text{id : } ID \\
\text{isbn : } ISBN
\end{array}
\]

One of our user-level operations, namely \texttt{Borrow\_Book} for issuing a book to a user, is specified in the next section.

3.3 A User-Level Operation

Consider schema \texttt{Borrow\_Book} below. Schema \texttt{Borrow\_Book} adheres to design guidelines \#2, \#6 and \#8 above by:

- maximising communication with the user,
- using a sequence of primitives to update state and environment components, and
- placing the control statements in the schema and not in the primitives.

\footnote{We assume the library contains reference works that are available but cannot be borrowed by a user.}

\footnote{Our simple specification does not require the removal of books or users from the system, hence we need not keep the detail of a deleted book or user in the environment.}
\_Borrow\_Book^5
\Delta \text{Library}; \Delta \text{LibEnv}
\text{id}? : \text{ID}; \text{isbn}? : \text{ISBN}
\text{mes!} : \text{Message}

\* \* Definition of environment : \* \*
\[ \theta \text{LibEnv}' = \begin{cases} 
& \text{id}? \in \text{dom users} \land \text{isbn}? \in \text{available} \text{ then} \\
& \text{Env}_\text{id}(\text{id}?, \text{Env}_\text{isbn}(\text{isbn}?, \theta \text{LibEnv})) \\
& \text{else} \ \theta \text{LibEnv} 
\end{cases} \]

\* \* Definition of state space : \* \*
\[ \theta \text{Library}' = \begin{cases} 
& \text{id}? \in \text{dom users} \land \text{isbn}? \in \text{available} \text{ then} \\
& \text{Borrow}(\text{UnAvail}(\theta \text{LibEnv}', \theta \text{Library})) \\
& \text{else} \ \theta \text{Library} 
\end{cases} \]

\* \* Definition of messages : \* \*
\[ \text{mes!} = \begin{cases} 
& \text{id}? \in \text{dom users} \land \text{isbn}? \in \text{available} \text{ then} \\
& \text{Bookisbn} \text{? borrowed by user id}? \\
& \text{else if} \ \text{id}? \notin \text{dom users} \land \text{isbn}? \in \text{available} \text{ then} \\
& \text{Invalid user id}? \\
& \text{else if} \ \text{id}? \in \text{dom users} \land \text{isbn}? \notin \text{available} \text{ then} \\
& \text{Bookisbn} \text{? unavailable} \\
& \text{else Invalid user id}? \text{ and book isbn}? \text{ unavailable} 
\end{cases} \]

3.4 Definition of Primitives

The primitives in \_Borrow\_Book are defined using ordinary set-theoretic notation. \text{Env}_\text{isbn} places an isbn in the environment to be used in the event of an undo and \text{Env}_\text{id} performs a similar function for a user id:

\- \text{Env}_\text{isbn} : \text{ISBN} \times \text{LibEnv} \rightarrow \text{LibEnv} \text{ is given by}
\text{Env}_\text{isbn}(\text{isbn}?, \text{env}) = \text{env}', \text{ where env}'.\text{isbn} = \text{isbn}?

\- \text{Env}_\text{id} : \text{ID} \times \text{LibEnv} \rightarrow \text{LibEnv} \text{ is given by}
\text{Env}_\text{id}(\text{id}?, \text{env}) = \text{env}', \text{ where env}'.\text{id} = \text{id}?

Primitive \text{UnAvail} makes a book unavailable while \text{Borrow} issues the book to a user:

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

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

^5 \text{Note that the use of `else if' in this way is non-standard in Z but the use of an if P then E₁ else E₂ construct is — see Spivey [15]}.
3.5 Definition of an Undo

To undo the effect of operation Borrow_Book we specify:

\[
\text{undo\_borrow\_book} \quad \begin{align*}
\Delta \text{Library}; \Xi \text{LibEnv} \\
\text{msg}! : \text{Message}
\end{align*}
\]

\[\text{\theta Library}' = \text{undo\_unavail}(\text{\theta LibEnv}, \theta \text{Library})\]

\[\text{msg}' = \text{previous borrow operation reversed}\]

Primitive undo\_borrow reverses the effect of Borrow:

- \text{undo\_borrow} : \text{LibEnv} \times \text{Library} \rightarrow \text{LibEnv} \times \text{Library} is given by

\[\text{undo\_borrow(env, library)} = (\text{env, library}')\]

where

\[\text{library}'\text{.borrowed} = \{ \text{env.isbn} \} \quad \text{library.borrowed}\]

Primitive undo\_unavail is specified in a similar fashion.

3.6 Discharging a Proof Obligation

Suppose an unregistered donor donates a new book to the library and thereby becomes a registered user. The following traditional schema describes this operation (see e.g. Potter et al. [26]).

\[
\text{donate} \quad \begin{align*}
\Delta \text{Library}; \Delta \text{LibEnv} \\
\text{isbn?} : \text{ISBN} \\
\text{id?} : \text{ID}_\perp \\
\text{title?} : \text{Title}; \text{aut?} : \text{Author}; \text{pub?} : \text{Publisher}; \text{yr?} : \text{Year} \\
\text{name?} : \text{Name}; \text{addr?} : \text{Address} \\
\text{msg}! : \text{Message}
\end{align*}
\]

\[\text{(id! \notin \text{dom users} \land isbn? \notin \text{dom books} \land}
\]

\[\text{isbn = isbn?} \land \text{id} = \text{id!} \land
\]

\[\text{books'} = \text{books} \cup \{ \text{isbn?} \mapsto (\text{title?}, \text{aut?}, \text{pub?}, \text{yr?}) \} \land
\]

\[\text{users'} = \text{users} \cup \{ \text{id!} \mapsto (\text{name?}, \text{addr?}) \} \land
\]

\[\text{available'} = \text{available} \cup \{ \text{isbn?} \} \land
\]

\[\text{borrowed'} = \text{borrowed} \land \text{date'} = \text{date} \land
\]

\[\text{msg}' = \text{OK} \}

\lor

\[\text{((dom users = ID \lor isbn? \in \text{dom books}) \land}
\]

\[\theta \text{LibEnv} = \theta \text{LibEnv} \land
\]

\[\theta \text{Library}' = \theta \text{Library} \land
\]

\[\text{id!} = \bot \land
\]

\[\text{msg}' = \text{System error}\}
\]
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.:

\[
\text{Donate} \supset \text{Donate}^\sim \vdash \Xi \text{Library}
\]  

(8)

The popular and widely used first-order, resolution-based theorem prover, OTTER (see e.g. McCune [27], Wos [28]), has difficulty in proving (8) above.

If, however, we rewrite schema \text{Donate} as a sequence of 3 primitives operations

- \text{Capture\textunderscore book} (say), a primitive to specify
  \[\text{books}' = \text{books} \cup \{\text{isbn} \mapsto (\text{title}, \text{auth}, \text{pub}, \text{yr})\}\],
- \text{Register\textunderscore user} (say), to specify
  \[\text{users}' = \text{users} \cup \{\text{id} \mapsto (\text{name}, \text{addr})\}\],
- \text{Avail} (say), for
  \[\text{available}' = \text{available} \cup \{\text{isbn}\}\],

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

\[
\text{Capture\textunderscore book} \supset \text{Capture\textunderscore book}^\sim \vdash \text{books}' = \text{books}
\]

(9)

where \text{books} represents the component before primitive \text{Capture\textunderscore book} and \text{books}' the same component after \text{Capture\textunderscore book}^\sim.

Quick proofs are also obtained for:

\[
\text{Register\textunderscore user} \supset \text{Register\textunderscore user}^\sim \vdash \text{users}' = \text{users}
\]
\[
\text{Avail} \supset \text{Avail}^\sim \vdash \text{available}' = \text{available}
\]

Failing to prove (8) above is more significant than it may seem. A specifier may decide to leave schema \text{Donate} as it is and attempt to perform 3 different simpler proofs, one of which could be:

\[
\text{Donate} \supset \text{Donate}^\sim \vdash \text{books}' = \text{books}
\]

(10)

Again the theorem-prover has difficulty in proving (10) above, since it fails to find a proof in 30 minutes running on a Pentium III with a clock speed of 600 MHz, while a proof of (9) is found after just 0.15 seconds on the same machine.

The architecture of this last failed proof attempt is characterised by the presence of redundant information (see e.g. van der Poll & Labuschagne [29]) in the sense that changes in the state components \text{users} and \text{available} in schema \text{Donate} are irrelevant to a proof of (10). It turns out that such irrelevant information leads the theorem prover astray. Of course, the specifier can remove the redundant information from the proof attempt, which then boils down to using primitives in line with Guideline #6.

3.7 Summary

We demonstrated in this section how all our guidelines can be applied in constructing a formal specification. A proof obligation involving a primitive operation and its undo primitive was successfully discharged.
4 Analysis and Future Work

This paper presented a number of guidelines for drawing up a formal specification. Central to these ideas is the use of primitives, allowing a specifier to apply a 'divide-and-conquer' approach to the specification of an operation. We illustrated how these primitives facilitate 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.

Additional implementation benefits may be realised through the use of primitives. Since every primitive change at most one component of the state or environment, on a multi-processor machine we can assign a processor to a primitive and if some of the primitives happen to be independent, then we may achieve true concurrency. (Having said this, we note that it is generally accepted that implementation issues are not to influence decisions made at the specification phase.) Also, on a threaded single-processor machine we can program a user-level operation as a task and each primitive as a thread within the task (see Silberschatz et al. [30] for a discussion of operating system threads). 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.

An important issue emerges from the use of an environment in this work: 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. For example, in some word-processing packages (e.g. some later versions of WordPerfect®) it is possible to undo a number of previous operations, one after another. Typically a sequence of environments could be used for this purpose. Future work could concentrate on incorporating such sequences of environments.

References