

Designing and Refining Specifications with Modules

Joseph M. Morris and Shahad Ahmed

Dept of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland

Abstract. Specifications have a dual existence: on the one hand they constitute a contract between the customer and the vendor, and on the other hand they are the instructions from the vendor to his programmers. From the former viewpoint, we would like to have methods for incrementally constructing specifications, and from the second viewpoint we would like to have mechanisms for systematically "calculating" an implementation of a specification. Here we seek to develop a methodology that can give formal support to both these activities within a model-oriented framework. Indeed, these two activities, which we may call designing and refining, have much in common, and to some extent we may also "calculate" parts of a specification. The basic theoretical notion underlying our approach is that programs and specifications are more or less the same thing: a programming language is the implementable subset of a specification language. The basic practical tool we outline is a notion of modules that supports the incremental construction of specifications.

1. Introduction

Specifications play a dual role. On the one hand they constitute the contract between the customer and the owner of the software company, and on the other they are the instructions from the owner to his programmers. Work on specifications has mainly been from the viewpoint of specifications as contracts, and their role in program construction is not nearly as well developed or understood. Typically a programmer will read a specification to the extent that he needs to know what is being asked of him, and then gets on with the job of programming. Perhaps he will refer back to the specification in moments of doubt, but it will not otherwise infringe on his time. But we well know by now that the specification can usefully play a much more active role in programming: it is possible, at least in principle, to derive a program from its specification by mathematical transformation. Our purpose here is to outline a specification methodology that recognises and supports the dual role of specifications in designing systems and programming by transformation.

Our basic tenet is that programs and specifications are more or less the same thing: a programming language is the implementable subset of a specification language, or viewed the other way around, a specification language is a programming language with added fancy constructs that admit ease of

expression although they may be expensive or even impossible to implement. The programming task is to make a specification in this rich language and then step by step eliminate the fancy constructs with correctness-preserving transformations — so called "refinements" — until we arrive at a program. Programs *are* specifications, although somewhat special in that we know how to execute them. Although occasionally the best specification of a program is the program itself, usually the specification will be quite un-program-like; then we have to develop the program by constructing a sequence of ever more algorithmic specifications until we arrive at one from which all the non-algorithmic constructs have been eliminated. This is pretty much what we do informally when we program by stepwise refinement.

Specifications are as complex in their structure as programs. We have to construct them very methodically in small pieces which we then combine into bigger specifications, and so on hierarchically. The specification language Z [7] has pioneered this style. Coincidentally, programming by stepwise refinement also proceeds by repeatedly translating small pieces of specification into code. As far as is practicable, we would like the constituent pieces of the specification also to be the pieces we make the subject of refinements, or put more simply, we would like the program to have much the same structure as the specification. The advantages are these. Firstly, we avoid the effort of reshaping the specification prior to deriving its implementation. Secondly, it places a tight upper bound on the amount of recoding we have to do when a part of the specification is changed. Thirdly, we can more easily reuse implementations when a piece of a specification is used more than once. We regard this latter point as important in that it encourages us to make libraries of useful specifications, together with their implementations, that can be used in more than one application. Although it seems desirable that the program should retain the structure of the specification, in fact it is not always easy to achieve, nor does it always yield an acceptably efficient implementation. Nonetheless we would like to encourage it.

The presentation will be tutorial in style. We assume some idea on the readers part of the purpose of specifications, but not any great intimacy with them. We will proceed mainly by example, without attempting to describe the mathematical underpinnings of the method. Of course it is very important that the specification language be given a formal semantics so that we can formally derive the laws of refinement, but that is a separate issue and is discussed elsewhere [1-6]. We use various typefaces in specifications and explanatory text, just to help the eye; they have no significance otherwise.

2. A simple example

Figure 1 is a specification in a Pascal-like syntax of a simple library catalogue, with explanations following. The actual type definitions of BOOK etc. aren't significant and are left open. Each copy of a book is associated with a unique identifier, so for BOOK we would probably choose the naturals, TITLE will probably be a sequence of characters, and so on. Because we are writing a specification, which doesn't have to be implemented, we can avail of any mathematical types we find useful. The values in type "set of BOOK" are all subsets of the values in BOOK. Variable *title* has type $\text{BOOK} \rightarrow \text{TITLE}$ — in words, *title* is a partial function from BOOK to TITLE; its values are sets of pairs, the left hand element of which is from BOOK and the right hand element of which is from TITLE, and such that the left hand elements are unique. So *title* just associates each book identifier with a title. Similarly, *authors* associates each book identifier with its list of authors. The set of left hand elements in a partial function *f* is called the domain of *f* and is denoted by $\text{dom}.f$; the set of right hand elements is called the range of *f* and is denoted by $\text{rng}.f$. We will often want to constrain the values a variable may take on, over and above that imposed by its type; we include such constraints in clauses beginning with a vertical line. For example, we intend that *title* should include the title of every book currently in the library and we write this as $\text{books} \subseteq \text{dom}.title$. We might have expected to see $\text{books} = \text{dom}.title$ as the constraint in this case; that is not excluded, of course, and will most likely be observed by the implementation, but it's not essential to the specification and so we choose the weaker constraint — specifications should be as unconstrained as possible while remaining consistent with the requirements. The conjunction of all such constraints is called the "invariant" of the module. We are free to use all the standard operations that come with types, such as set membership, set union, and set difference in the case of set types. The operation \oplus on ordered pairs is so-called "function overriding": the result of $\text{title} \oplus \{(b,t)\}$, for example, is a copy of *title* from which any pairs whose first element is *b* (there

```

library_catalogue module =

    type    BOOK = ... ;
            TITLE = ... ;
            AUTHOR = ... ;
            SUBJECT = ... ;

    var     books: set of BOOK;
            title: BOOK → TITLE | books ⊆ dom.title;
            authors: BOOK → sequence of AUTHOR | books ⊆ dom.authors;
            subject: BOOK → SUBJECT | books ⊆ dom.subject;

    procedure addbook(b: BOOK; t: TITLE; as: sequence of AUTHOR; s: SUBJECT);
    begin
        if b ∈ books then
            begin
                books := books ∪ {b}; title := title ⊕ {(b,t)}
                author := author ⊕ {(b,as)}; subject := subject ⊕ {(b,s)}
            end
            else message('Non-unique book identifier')
        end;

    procedure removebook(b: BOOK);
    begin
        if b ∈ books then books := books - {b}
        else message('Unknown book')
    end;

    function booksby(a: AUTHOR): set of BOOK;
    begin
        booksby := {b: b ∈ books ∧ a ∈ authors.b}
    end;

    function bookson(s: SUBJECT): set of BOOK;
    begin
        bookson := {b: b ∈ books ∧ s = subject.b}
    end;

end

```

Fig. 1: Library catalogue in Pascal-like syntax

will be at most one in the present case) have been removed and to which $\{(b,t)\}$ has been added. So $\text{title} := \text{title} \oplus \{(b,t)\}$ just records the title t of a new book b being added to the library. The set membership symbol \in is extended to sequences in the obvious way. We shall not be too specific about handling "error situations"; for simplicity we will assume some message is to be conveyed, and assume the availability of routine "message" which achieves this. Function application is denoted by an infix dot; it has the highest operator precedence.

The library catalogue meets all the requirements of a good specification: it is precise, unambiguous, and says no more than is necessary to convey what is to be achieved. We are also told that specifications should not be operational, and on this ground the reader is perhaps feeling uneasy. Doesn't the specification use an assignment statement, for example, and isn't that an operational construct? Not at

all! The assignment statement is just a convenient mathematical notation, simple and succinct. It so happens that it can also be given an operational interpretation, but that's just by the way as far as we're concerned. Note also that when removing a book from the catalogue we didn't specify any change to variables *title*, *authors*, or *subject*, and so these variables retain old information. We could have updated them if we cared to, but it is just extra work to no purpose: there is no notion of "space efficiency" of specifications, just as there is no notion of "time efficiency". The important thing is that a specification should not exclude efficient implementations, and it can be shown that the specification we have made does not. We will discuss implementations later. The specification of Figure 1 is complex in that it is composed of smaller specifications such as **addbook** and **removebook**, and these in turn are composed of still smaller specifications such as $\text{books} := \text{books} \cup \{b\}$.

Actually we do not use a Pascal-like syntax when we make specifications: the specification we would actually write is shown in Figure 2. To begin with, Pascal-like syntax is too verbose; there is a lot of writing in making specifications, and we prefer to be succinct. We make parameters of the types that need not be fixed in the module. The end of a parameter list in the definition of an operation is indicated by a period (which is quite different from the period of function application), and the end of a definition is indicated by a fat dot. The use of semicolons in Figure 1 is over-specific. It is not attractive to regard the removal of a book as consisting of four small actions carried out in some arbitrary order, for we really don't care what order they're carried out in, and they could even be carried out simultaneously. In general we avoid composing statements with semicolons, and instead use the parallel combinator \parallel : we shall describe this more precisely below, but for the moment it suffices to think of it as indicating that all the operations described by its arguments should take place in parallel. The commonly occurring form $x := x \text{ op } y$, where **op** is some operation, is abbreviated to $x:\text{op } y$; to avoid ambiguity this abbreviation is not used when **op** is $=$.

```

library_catalogue module [BOOK, TITLE , AUTHOR, SUBJECT] =

  books: set of BOOK;
  title: BOOK  $\rightarrow$  TITLE |  $\text{books} \subseteq \text{dom.title}$ ;
  authors: BOOK  $\rightarrow$  sequence of AUTHOR |  $\text{books} \subseteq \text{dom.authors}$ ;
  subject: BOOK  $\rightarrow$  SUBJECT |  $\text{books} \subseteq \text{dom.subject}$ ;

  addbook =
    b: BOOK, t: TITLE, as: sequence of AUTHOR, s: SUBJECT.
     $b \in \text{books} \rightarrow$ 
       $\text{books}:\cup \{b\} \parallel \text{title}:\oplus \{(b,t)\} \parallel \text{author}:\oplus \{(b,as)\} \parallel$ 
       $\text{subject}:\oplus \{(b,s)\}$ 
    []  $b \in \text{books} \rightarrow$ 
      message('Non-unique book identifier') •

  removebook =
    b: BOOK.
     $b \in \text{books} \rightarrow \text{books}:- \{b\}$ 
    []  $b \in \text{books} \rightarrow \text{message}(\text{'Unknown book'}) \bullet$ 

  booksby =
    a: AUTHOR.  $\{b: b \in \text{books} \wedge a \in \text{authors.b}\} \bullet$ 

  bookson =
    s: SUBJECT.  $\{b: b \in \text{books} \wedge s = \text{subject.b}\} \bullet$ 

end

```

Fig. 2: Library catalogue version 2

An if-statement is composed of a collection of so-called "guarded commands" combined together using the choice operator \square . A guarded command has the form $P \rightarrow s$ where P stands for an assertion (i.e. a boolean-valued expression) called the guard, and s stands for a specification. If P is true then $P \rightarrow s$ behaves like s , and otherwise it doesn't behave at all: in that case we hope there is some other guarded command in the collection whose guard is true. A set of guarded commands specifies a choice among any one of those operations in the set that are preceded by a guard that evaluates to true. If more than one guard is true then it is not defined which corresponding operation is chosen: it is up to the implementor. For example,

$$x \geq y \rightarrow y := y + 1 \square x \leq y \rightarrow x := x + 1$$

leaves it open as to whether $y := y + 1$ or $x := x + 1$ is chosen when $x = y$. Specifications are in this way "nondeterministic" in that they describe not one unique computation but a collection of possible implementations all of which are acceptable to the customer. Nondeterminacy is an attractive property of specifications because it allows us to be non-committal when we really don't care; the implementor can then make a choice based on convenience or efficiency. The collection of implementations admitted by a specification could even be empty for some states: this would happen, for example, if there is some state for which none of the guards in a guarded command set is true. Such a specification is semantically okay, but unimplementable for those states, and we say it is "partial" or "miraculous" — "miraculous" because the specifier appears to be expecting miracles from the implementor. For example,

$$x > y \rightarrow y := y + 1 \square x < y \rightarrow x := x + 1 \quad (*)$$

is miraculous in states satisfying $x = y$, and no implementation exists for such states. One may be tempted to take the Pascal view that (*) could be implemented in the event that $x = y$ by doing nothing, but that's not the semantics we want — if the specification gives no options then it means just that, it is not even giving the option of doing nothing. Asking an implementor to implement a specification when it is miraculous is like asking a person to choose a sweet from an empty bag — it can't be done. If we really want no action in (*) when $x = y$ then we must say so explicitly, as follows

$$x > y \rightarrow y := y + 1 \square x < y \rightarrow x := x + 1 \square x = y \rightarrow \mathbf{skip}$$

where **skip** (which we shall describe more precisely later) means "do nothing". Although final specifications will never (or should never) be miraculous, in fact miraculous specifications are very useful in the systematic construction of large specifications: we often make small specifications that happen to be miraculous, but which will later be combined with others to form a non-miraculous whole. We will see examples of this shortly.

The module of Figure 2 is still not quite how we would make it in practice: the specification we would probably write is shown in Figure 3, with explanations following. The main change that Figure 3 introduces is a stylistic one: the constituent specifications are smaller. We prefer to make lots of small specifications because it makes the task more manageable, and because it increases the chances of reusing old specifications.

In general, the constituent specifications of a module will refer freely to the variables of the module, but they may also refer to other objects. For each named operation we give arbitrary names to such extra objects by means of a piece of text that precedes the definition proper; this text is called the parameter list in the jargon of programming languages but we will use the term "signature". We let σ_s denote the signature of operation s . For example, in

AMtime =
 h : INTEGER, m : INTEGER.
 $0 \leq h < 12 \wedge 0 \leq m < 60$ •

PMtime =
 σ AMtime.
 $12 \leq h < 24 \wedge 0 \leq m < 60$ •

we could equally well have written h : INTEGER, m : INTEGER instead of σ AMtime.

library_catalogue module [BOOK, TITLE, AUTHOR, SUBJECT] =

books: set of BOOK;
title: BOOK \rightarrow TITLE | $\text{books} \subseteq \text{dom.title}$;
authors: BOOK \rightarrow sequence of AUTHOR | $\text{books} \subseteq \text{dom.authors}$;
subject: BOOK \rightarrow SUBJECT | $\text{books} \subseteq \text{dom.subject}$;

addbook0 =
 b : BOOK, t : TITLE, as : sequence of AUTHOR, s : SUBJECT.
 $\neg \text{catalogued} \rightarrow$
 $\text{books} : \cup \{b\} \parallel \text{title} : \oplus \{(b,t)\} \parallel \text{author} : \oplus \{(b,as)\} \parallel$
 $\text{subject} : \oplus \{(b,s)\}$ •

catalogued =
 b : BOOK. $b \in \text{books}$ •

addbook =
 σ addbook0. addbook0 [] oldbook •

oldbook =
 b : BOOK.
 $\text{catalogued} \rightarrow \text{message}(\text{'Non-unique book identifier'})$ •

removebook0 =
 b : BOOK.
 $\text{catalogued} \rightarrow \text{books} : - \{b\}$ •

removebook =
 σ removebook0.
 $\text{removebook0} [] \text{badbook}$ •

badbook =
 b : BOOK.
 $\neg \text{catalogued} \rightarrow \text{message}(\text{'Unknown book'})$ •

booksby =
 a : AUTHOR. $\{b : b \in \text{books} \wedge a \in \text{authors.b}\}$ •

bookson =
 s : SUBJECT. $\{b : b \in \text{books} \wedge s = \text{subject.b}\}$ •

end

Fig. 3: Library catalogue version 3

The signature is just a local extension of the name space, the actual objects associated with the names being left open; we "bind", i.e. attach objects to the names, when we use the operation. Parameter names may be bound in three ways. The first way is the familiar one of supplying an actual parameter, or argument. The meaning of a procedure or function applied to an argument is the body of the procedure or function (i.e. the right hand side of its definition with the parameter list stripped away) with the parameter replaced everywhere with the argument. For example, given

```
goodtime =
  h: INTEGER, m: INTEGER.
  0 ≤ h < 24 ∧ 0 ≤ m < 60 •
```

then `goodtime.(12, 15)` is equivalent to $0 \leq 12 < 24 \wedge 0 \leq 15 < 60$, i.e. True. The second way we can bind parameters is by quantification, and we shall see examples of that later. The final way is by implicit binding. An argument may be omitted when there is a similarly named and typed object in scope at the point of "invocation"; the similarly named object is taken as the argument. In other words, the meaning of a procedure or function when "invoked" without arguments is just its body. For example, in

```
minutes_since_midnight =
  h: INTEGER, m: INTEGER.
  goodtime → h*60+m
  [] ¬goodtime → -1 •
```

`goodtime` is "invoked" without actual arguments, so the meaning of the invocation is got by substituting its body, giving

```
minutes_since_midnight =
  h: INTEGER, m: INTEGER.
  0 ≤ h < 24 ∧ 0 ≤ m < 60 → h*60+m
  [] ¬ 0 ≤ h < 24 ∧ 0 ≤ m < 60 → -1 •
```

We could equally well have written

```
minutes_since_midnight =
  h: INTEGER, m: INTEGER.
  goodtime.(h,m) → h*60+m
  [] ¬goodtime.(h,m) → -1 •
```

but that takes a little extra writing. We have used implicit binding in Figure 3 when applying `catalogued` within `addbook0`, and again when applying `addbook0` and `oldbook` within `addbook`. `addbook`, for example, is equivalent to — and could have been defined as:

```
addbook =
  b: BOOK, t: TITLE, as: sequence of AUTHOR, s: SUBJECT.
  ¬ b ∈ books → books: ∪ {b} || title: ⊕ {(b,t)} || author: ⊕ {(b,as)} ||
  subject: ⊕ {(b,s)}
  [] b ∈ books → message('Non-unique book identifier') •
```

Of course, implicit binding is only legitimate when it is used in a context that includes objects whose name and type agree with the signature of the parametrised specification.

Note that parametrisation has low binding power, so the brackets in `x:TYPEX`. (`s [] t`) are superfluous. Observe, also, that `oldbook` is a partial specification, and that that's perfectly acceptable in the context.

3. A closer look at notation

The most important statement that the specification language has over its constituent programming language is what's called the "nondeterministic assignment". This has the form $lv:\sim P$ for lv a list of variables and P an assertion, and the following semantics: assign values to the variables in lv so that P holds after the assignment. For example $x:\sim x=0$ asks for x to be given the value 0, and $x:\sim x>0$ asks for x to be given any positive value. We may use primed variables in the P of $lv:\sim P$ to denote the values of variables before the assignment: for example, $x,y:\sim (x<y \wedge y>y')$ asks for the value of y to be increased and x to be given a value less than the new value of y . Actually we don't make much use of the nondeterministic assignment in this form in the present paper, because we can get by with a simple form of it. The simple assignment $x:=e$ is an abbreviation for $x:\sim x=e'$ where e' denotes e with all its specification variables primed. For example, $x:=x+1$ is equivalent to $x:\sim x=x'+1$, i.e give x a value equal to its old value plus one. The specification $x:\sim \text{False}$ (where x is any variable or list of variables, even the empty list ϵ) is legitimate but everywhere miraculous and so wholly unimplementable: it specifies the empty set of programs and is called **miracle**. The specification $x:\sim \text{True}$ means "give x any value you like", and of course can be implemented trivially.

The parallel assignment $x:\sim P \parallel y:\sim Q$ is defined to be equivalent to $x,y:\sim P \wedge Q$. For example, $x:=y \parallel y:=x$ is equivalent to $x:\sim x=y' \parallel y:\sim y=x'$ which is equivalent to $x,y:\sim x=y' \wedge y=x'$, i.e. exchange the values of x and y . \parallel is evidently commutative and associative, and has identity element $\epsilon:\sim \text{True}$. $\epsilon:\sim \text{True}$ specifies "do nothing" and is given the name **skip**. Note that in $x:\sim P$ there is never any need to prime variables in P that do not occur in x ; for example $x:=y+z \parallel y:\sim y>0$ is equivalent to $x,y:\sim (x=y'+z \wedge y>0)$ — there is no need to prime z because its value is not changed by the assignment. The arguments of \parallel are usually assignments but in fact may be any semicolon-free specification. As a small exercise, consider the specification $x:=1 \parallel x:=2$ (below, and elsewhere, we intersperse proof steps with their justification enclosed by double quotes):

$$\begin{aligned} & x:=1 \parallel x:=2 \\ = & \text{"definition of simple assignment"} \\ & x:\sim x=1 \parallel x:\sim x=2 \\ = & \text{"definition of } \parallel \text{"} \\ & x:\sim x=1 \wedge x=2 \\ = & \text{"logic"} \\ & x:\sim \text{False} \\ = & \text{"definition"} \\ & \mathbf{miracle} \end{aligned}$$

— $x:=1 \parallel x:=2$ is a legitimate specification, but it is impossible to implement.

For P an assertion and s a specification, the specification $P \rightarrow s$ is equivalent to s if the initial state satisfies P , and otherwise it is equivalent to **miracle**. We give \parallel an operator precedence above \rightarrow . Some laws are:

- | | | | | |
|-----|-----------------------------------|---|-------------------------------|-------------|
| (A) | $\text{False} \rightarrow s$ | = | miracle | for any s |
| (B) | $\text{True} \rightarrow s$ | = | s | |
| (C) | $P \rightarrow \mathbf{miracle}$ | = | miracle | |
| (D) | $(P \rightarrow s) \parallel t$ | = | $P \rightarrow s \parallel t$ | |
| (E) | $P \rightarrow (Q \rightarrow s)$ | = | $P \wedge Q \rightarrow s$ | |

Such laws may seem trivial and even useless, but they are as fundamental to manipulating specifications as the law $x+0 = x$ is to doing arithmetic.

For s and t any specifications, $s \sqcup t$ is a specification meaning "either s or t and I don't care which". For example $x:=0 \sqcup x:=1$ specifies "let x have the value either 0 or 1". We give \sqcup an operator precedence below \rightarrow ; so the brackets in $((R \rightarrow (x:\sim P \parallel y:\sim Q)) \sqcup s)$ are superfluous. It should be intuitively obvious that \sqcup is commutative and associative, and has **miracle** as its identity element. Note that the degree of nondeterminacy is state-dependent. For example, $x:=1 \sqcup y \neq 0 \rightarrow x:=0$ offers the choice of giving x the value 0 or 1 in the case that y differs from 0, and otherwise only the choice 1. The specification $P \rightarrow s \sqcup \neg P \rightarrow t$ is equivalent to s in the case that P holds, and otherwise t — so it is similar in style to the Pascal-style **if** P **then** s **else** t . But $P \rightarrow s$ is not similar to **if** P **then** s — the sense of the latter is captured by $P \rightarrow s \sqcup \neg P \rightarrow \text{skip}$ which is not quite the same as $P \rightarrow s$.

The invariant of a module is not simply a comment expressing a good intention, but rather shapes the semantics of the module. Without going into technicalities, the semantics are such that any action that would violate the invariant turns out to be miraculous and so the rules of refinement will never lead to an implementation — there just isn't one. In effect, we have a proof obligation to show that the invariant is maintained, but this is usually just clerical routine. In the case of Figure 3, a cursory inspection of the text suffices to convince that the invariant is maintained: Consider the constraint $\text{books} \subseteq \text{dom.title}$, for example. It could only be violated by adding a new book or discarding a title; but when a new book is added then *title* is correspondingly enlarged, while on the other hand the domain of *title* is never decreased, so the constraint is maintained.

4. An extended example

We will further illustrate the specification language and style by pursuing the library example further. We are asked to specify a library system in which we may add and remove a book from the library, list the books in a particular subject area or by a particular author, add or remove a library user, enquire about the personal details of a particular user, lend out a book and accept its return, enquire as to whether a particular book is available for loan, and list all the books on loan to a particular user. Moreover, certain operations such as adding a new user to the library are to be restricted to library staff only.

The first step in addressing a specification problem is to isolate good abstractions. This not only helps us to manage the complexity of detail that always threatens to overwhelm us when we tackle a computational problem, but also increases the chances that we'll be able to exploit existing specifications and their implementations, and further that any specification we make might be reusable in the future. It also helps to contain the damage when the requirements are changed. The library system on first inspection appears to comprise three more or less independent components: a catalogue of books, a register of users including staff, and a record of books borrowed. It seems a good idea to model these components in isolation, and then look at their composition. We shall have to be quite determined to treat these as independently as possible, for it is easy to be persuaded that they are inextricably linked together. It would seem difficult to describe the removal of a user from the register, for example, without considering whether the user has library books in his possession, and for that we need to know about the list of borrowings, whereas on the other hand it would seem that we can't describe borrowings without knowing about the register of users! Nonetheless, we shall try to isolate the various concerns.

Let us begin by describing the catalogue of books. That's easy in this case because we have an old specification that we can reuse, the one in Figure 3. But what about protection, and such problems as deleting a book that is on loan? As far as protection is concerned, that's an issue that concerns the library as a whole, and so this is not the level at which to consider it. Neither at this level should we worry about borrowed books: we should just describe the essence of a catalogue, and the specification of Figure 3 does just that.

Next we'll describe the register of users. That's very like a catalogue of books, so we can just rework the specification of Figure 3, arriving at the specification in Figure 4. We use δ in

```

library_users module [PERSON, NAME, ADDRESS, STATUS] =

  users: set of PERSON;
  name: PERSON  $\rightarrow$  NAME | users  $\subseteq$  dom.name;
  address: PERSON  $\rightarrow$  ADDRESS | users  $\subseteq$  dom.address;
  status: PERSON  $\rightarrow$  STATUS | users  $\subseteq$  dom.status;

  adduser0 =
    p: PERSON, n: NAME, a: ADDRESS, s: STATUS.
     $\neg$  registered  $\rightarrow$ 
      users:  $\cup$  {p} || name:  $\oplus$  {(p,n)} || address:  $\oplus$  {(p,a)} ||
      status:  $\oplus$  {(p,s)} •

  registered =
    p: PERSON. p  $\in$  users •

  adduser =
     $\sigma$  adduser0. adduser0 [] olduser •

  olduser =
    p: PERSON.
    registered  $\rightarrow$  message('Person already in library') •

  removeuser0 =
    p: PERSON.
    registered  $\rightarrow$  users: - {p} •

  removeuser =
     $\sigma$  removeuser0. removeuser0 [] baduser •

  baduser =
    p: PERSON.
     $\neg$  registered  $\rightarrow$  message('Not a registered user') •

  enquireuser0 =
    p: PERSON;  $\delta$  n: NAME, a: ADDRESS, s: STATUS.
    registered  $\rightarrow$  n:= name.p || a:= address.p || s:= status.p •

  enquireuser =
     $\sigma$  enquireuser0. enquireuser0 [] baduser •

end

```

Figure 4. Register of library staff and borrowers.

enquireuser0 to indicate output parameters (akin to **var** as used in Pascal parameter lists). That the invariant is maintained is evident from a brief inspection of the text.

To describe the list of borrowings we will need to know about calendars so that we can calculate the date on which a book being borrowed is due back. Presumably a calendar module is to be found in the library of specifications we will have built up; an outline of one is given in Figure 5.

```

calendar module =

    DATE = ...; "e.g. yymmdd where yy=year, mm=month, dd=day"

    today: DATE | validate.today;

    validate =
        d: DATE. ... "d is a legitimate date" •

    setdate =
        d: DATE.
        validate → today:= d
        [] baddate •

    baddate =
        d: DATE.
        ¬validate → ... •

    future =
        n: NATURAL. ... "the date n days following today" •

    update =
        today:= future.1 •
    .
    .
    .

end

```

Figure 5: A calendar module.

The borrowings file is given in Figure 6, with some explanations following.

The **include** clause is equivalent to a textual expansion of the module it names; we have been careful to keep all names distinct to avoid discussing issues of scope and visibility.

Implicit binding is used for **loansto** in the definition of **within**; #loansto is just #{b: (b,p)∈has}. The constraint of *has* uses binding by quantification (as well as binding by application to an argument). For any functional specification *f* and any quantifier *Q* appropriate to the type of *f*, we may write *Qf* to denote *f* with the names in its parameters bound by *Q*. For example, \exists **registered**, where **registered** is defined in Figure 4, denotes $\exists p: \text{PERSON}. p \in \text{users}$, i.e. that at least one person is a registered borrower. This device is used in the constraint of the partial function *has*: $\forall(\text{within.limit})$ is equivalent to $\forall p: \text{PERSON}. \# \text{loansto} \leq \text{limit}$, and substituting for **loansto** this is equivalent to $\forall p: \text{PERSON}. \# \{b: (b,p) \in \text{has}\} \leq \text{limit}$, i.e. that no person has more than *limit* books on loan.

The operation **ds** (in **return0**) stands for so-called "domain subtraction": *has ds* {b} equals *has* with all pairs whose left hand element is *b* deleted (there will be precisely one such pair in the present case). Note that the definition of **borrow** has a genuine nondeterminism in that an attempt by a user to borrow an unavailable book when he has already borrowed to his limit produces one of two error messages, but we don't know which one and we have chosen not to care. Of course, if we really did care then we would specify differently. Again, it is easy to check that the invariant is maintained.

It only remains to assemble the three pieces. Assembling such components begins with the question "what relationship between the variables do we wish to impose"; when we have answered that,

```

library_borrowings module [BOOK, PERSON] =

  include calendar;

  time: NATURAL;           "number of days that a user may retain a book"
  limit: NATURAL;         "maximum books a user can have on loan"
  has: BOOK → PERSON | ∀(within.limit); "which books are on loan, and to whom,..."
  due: BOOK → DATE | dom.has ⊆ dom.due; "... and when due back"

  within =
    n: NATURAL.
    p: PERSON. #loansto ≤ n •

  loansto =
    p: PERSON. {b: (b,p) ∈ has} •

  borrow0 =
    p: PERSON, b: BOOK.
    available ∧ within.(limit-1) →
    has:⊕ {(b,p)} || due:⊕ {(b,future.time)} •

  available =
    b: BOOK. b ∉ dom.has •

  borrow =
    σ borrow0. borrow0 [] bookout [] nomore •

  bookout =
    b: BOOK. ¬ available → message('Book out on loan') •

  nomore =
    p: PERSON.
    ¬ within.(limit-1) → message('At limit of borrowings') •

  return0 =
    b: BOOK. ¬ available → has:ds {b} •

  return =
    σ return0. return0 [] notout •

  notout =
    b: BOOK.
    available → message('Book not on loan') •

end

```

Fig. 6: Borrowings.

the extra specifying to be done will follow routinely. The catalogue and users modules are independent of one another. The borrowings module is related to the catalogue module in that the books borrowed must be in the catalogue, i.e. $\text{dom.has} \subseteq \text{books}$. The relationship between the borrowings module and the users module is simply that the persons to whom books are lent must be registered users, i.e. $\text{rng.has} \subseteq \text{users}$. In summary, the extra invariant is

$$\text{dom.has} \subseteq \text{books} \wedge \text{rng.has} \subseteq \text{users}$$

Now we just routinely go through the three pieces and ensure that the new invariant is maintained, suitably modifying any action that would violate it.

First we introduce a new convention: we allow more than one operation in a module to have the same name. When a module contains more than one operation with the same name \mathbf{n} we need a rule to determine which \mathbf{n} is being referred to whenever \mathbf{n} is "invoked". We distinguish two cases: a reference to \mathbf{n} in an operation whose name is \mathbf{n} , and all other instances. In the definition of the operations named \mathbf{n} a reference to \mathbf{n} is a reference to the \mathbf{n} most recently defined in the preceding text. Otherwise, any reference to \mathbf{n} refers to the \mathbf{n} that occurs last in the text. This renaming convention can save us the trouble of inventing new names; for example, in Figure 6 the successive definitions

$$\begin{aligned} \mathbf{return0} = \\ & b: \text{BOOK}. \neg \text{available} \rightarrow \text{has:ds} \{b\} \bullet \end{aligned}$$

$$\begin{aligned} \mathbf{return} = \\ & \sigma \text{return0}. \text{return0} [] \text{notout} \bullet \end{aligned}$$

could equally well have been written as

$$\begin{aligned} \mathbf{return} = \\ & b: \text{book}. \neg \text{available} \rightarrow \text{has:ds} \{b\} \bullet \end{aligned}$$

$$\begin{aligned} \mathbf{return} = \\ & \sigma \text{return}. \text{return} [] \text{notout} \bullet \end{aligned}$$

Here is a more algorithmic formulation of the rule to resolve references to a name \mathbf{n} which is defined more than once: attach subscripts $0, 1, 2, \dots$ to each defining occurrence of \mathbf{n} in order of appearance in the text; in the body of each \mathbf{n}_i attach subscript $i-1$ to each \mathbf{n} therein; and finally drop the subscript from the last defining occurrence of \mathbf{n} . Applying this algorithm to the successive definitions of \mathbf{return} above yields

$$\begin{aligned} \mathbf{return}_0 = \\ & b: \text{BOOK}. \neg \text{available} \rightarrow \text{has:ds} \{b\} \bullet \end{aligned}$$

$$\begin{aligned} \mathbf{return} = \\ & \sigma \text{return}_0. \text{return}_0 [] \text{notout} \bullet \end{aligned}$$

which is pretty much the definitions that appear in Figure 6. The main advantage of this convention is that it gives us a mechanism for changing the definition of an operation without changing its name; if we could not retain the old name then we would have the bother of changing every existing reference to the old name. Some care is needed when reusing a name, because reuse precludes any further reference to the old definition.

We proceed to inspect **library_catalogue** for violations of $\text{dom.has} \subseteq \text{books}$; this could only be violated by an action that removes an element from *books*, and that can only occur in **removebook0**, which indeed needs some revision:

$$\begin{aligned} \mathbf{removebook0} = \\ & \sigma \text{removebook0}. \\ & \quad \text{available} \rightarrow \text{removebook0} [] \text{bookout} \bullet \end{aligned}$$

There is no need to expand such definitions, but it may be helpful to see just one:

```

removebook
="definition from Figure 3"
  σ removebook0. removebook0 [] badbook
="definition of removebook0 above"
  σ removebook0. (available → removebook0 [] bookout) [] badbook
="[] associative"
  σ removebook0. available → removebook0 [] bookout [] badbook
="definition of removebook0 from Figure 3"
  b: BOOK.
  available → (catalogued → books:- {b})
  [] bookout [] badbook
="(E) from section 3"
  b: BOOK.
  available ∧ catalogued → books:- {b}
  [] bookout [] badbook
="definitions"
  b: BOOK.
  b ∈ dom.has ∧ b ∈ books → books:- {b}
  [] b ∉ books → message('Unknown book')
  [] b ∈ dom.has → message('Book out on loan')

```

Going through the same procedure for **library_users** we find that we have to enlarge **removeuser0**, and checking through **library_borrowings** we find that action **borrow0** needs modifying.

Finally, it is at this level that we should deal with querying the status of a book; it could not be done sooner as it involves both the catalogue of books and the borrowings file. The combined specification is shown in Figure 7. Note that we have composed the module of Figure 7 with pieces that are not as small as we have employed in earlier modules, just to show an alternative style.

Now we might proceed to design a protected library system in which certain operations are restricted to staff, but we shall leave the library there.

5. Discussion of the specification

One half of a specification methodology is the systematic construction of specifications, and we tried to show in the previous section that the methodology we are employing is capable of going some way towards achieving this. We have avoided many issues however: small issues such as initialising variables and exporting names, and larger issues such as scope of declarations, and parametrisation of modules. The usefulness of parametrisation is hinted at by our use of unspecified types, and further by the evident similarity between the modules of Figures 3 and 4, which we did not exploit. Despite the issues dodged, we feel that the specification style we have been illustrating suits its purpose well, but it would be wrong to claim too much on the evidence of one example. Let us consider why we specified the library as we have, and how we might have done it differently, both for better and for worse.

At first sight it may be irritating to have to read modules composed of so many small pieces: the module of Figure 3, for example, seems rather fragmented in comparison to the functionally equivalent one of Figure 2. But there are good reasons for keeping the pieces small. First of all, that's how we make them. Secondly, it increases the opportunities for reusing components, and indeed the final module makes use of operations **catalogued** and **available** from Figure 3, operations that do not exist in their own right in the catalogue of Figure 2. Thirdly, modules composed of small pieces make it easier to adapt the module to changing requirements. Suppose, for example, that we've just been told that a new feature is to be

```

library_system module [BOOK, TITLE, AUTHOR, SUBJECT,
                        PERSON, NAME, ADDRESS, STATUS] =

  include library_catalogue[BOOK, TITLE, AUTHOR, SUBJECT],
          library_users[PERSON, NAME, ADDRESS, STATUS],
          library_borrowings[BOOK, PERSON];

  | dom.has  $\subseteq$  books  $\wedge$  rng.has  $\subseteq$  users;

  removebook0 =
     $\sigma$  removebook0.
    available  $\rightarrow$  removebook0 [] bookout •

  removeuser0 =
     $\sigma$  removeuser0.
     $\neg$  hasbook  $\rightarrow$  removeuser0 [] bookdue •

  hasbook =
    p: PERSON. p  $\in$  rng.has •

  bookdue =
    hasbook  $\rightarrow$  message('Has a book on loan') •

  borrow0 =
     $\sigma$  borrow0.
    registered  $\wedge$  catalogued  $\rightarrow$  borrow0
    [] baduser [] badbook •

  bookonshelf =
    b: BOOK. catalogued  $\wedge$  available •

end

```

Figure 7: The library system

added to the catalogue module: all operations that change the state are to be "undoable", i.e. an operation **undo** is to be added that enables the user to recover the state that existed immediately prior to the current one. We can quite nicely build an extension on the catalogue of Figure 3 — see Figure 8 — but Figure 2 offers no such possibility: we would have to decompose it to its constituent parts, make the change, and reassemble. Note that although the specification of Figure 8 retains a complete copy of the old state, the semantics do not oblige an implementation to follow suit, and of course we would derive an implementation that just keeps a note of the difference between the present state and its predecessor.

The specifier's slogan is "Be minimalist!". The less said in a specification the better, not only because it increases the options when we come to implementing, but also because once we have set down the framework of a specification, much of the detail follows almost mechanically. Let's examine the library specification to see where we might have been tempted to say more, or could have done with saying less.

Consider the catalogue of Figure 3. Thinking as programmers, ever on the lookout to save space, we might have been tempted to specify the body of **removebook0** as

```
catalogued  $\rightarrow$  books:- {b} || title:ds {b} || authors:ds {b} || subject:ds {b}
```

```

undoable_library_catalogue module [BOOK, TITLE, AUTHOR, SUBJECT] =

  include library_catalogue[BOOK, TITLE, AUTHOR, SUBJECT];

  booksbefore: set of BOOK;
  titlebefore: BOOK  $\rightarrow$  TITLE;
  authorsbefore: BOOK  $\rightarrow$  sequence of AUTHOR;
  subjectbefore: BOOK  $\rightarrow$  SUBJECT;

  save =
    booksbefore:= books || titlebefore:= title ||
    authorsbefore:= authors || subjectbefore:= subject •

  addbook0 =
     $\sigma$  addbook0. addbook0 || save •

  removebook0 =
     $\sigma$  removebook0. removebook0 || save •

  undo =
    books:= booksbefore || title:= titlebefore ||
    authors:= authorsbefore || subject:= subjectbefore || save •

end

```

Fig. 8: Library catalogue with undo command.

There is nothing wrong with the new definition, but it adds pointlessly to the text and so just increases the work to be done in discharging proof obligations. Functionally the specifications are identical and admit precisely the same implementations.

There is only one guard in Figure 3, namely **catalogued** (and its negation). Employing this guard was a design decision in that the specification without it would still have been consistent with the informal requirements, although we would now have a different specification. We employed **catalogued** because we surmised that an attempt to add a book that was already in the library is indicative of something amiss with the world, but we could equally well have taken the view that all was in order, perhaps that the librarian was simply changing the attributes of the book. It is an advantage of formal specifications that they invite us to make decisions early, when the options are still open.

Turning to the borrowings module of Figure 6, consider **borrow0**. Its guard is a conjunction of two conditions, **available** and $\text{within}(\text{limit}-1)$. The first represents a design decision and could well have been omitted — one can easily think of a situation where it is reasonable to borrow a book that is on loan. The second condition, however, is motivated by the need to maintain the invariance of $\forall(\text{within}.\text{limit})$. Yet it turns out, and this may be surprising, that it too is dispensable — the functionality of **borrow0** is not changed a whit by its presence or absence. The reasons for this are a consequence of the formal semantics that we attach to assignments in the presence of module invariants. Of course, we will have to supply the guard at some stage in the derivation of an implementation, but the beauty of the refinement calculus is that the guard can be *calculated*, almost automatically from the statement it is to guard and the invariant that is to be maintained. Had we omitted the condition, we would inevitably meet it anyhow. The module of Figure 7, also contains a good deal of redundancy, and most of it could in fact have been calculated once the invariant was decided upon. (We could significantly shorten the description of modules — while imposing extra calculational work on the implementor — by the use of notational devices that exploit these redundancies, but we shall not do so in the present paper.) Having made the point that

specifications can leave out a great deal of what we might have regarded as crucial, perhaps we should also make the point that redundancies may be advantageous in so far as they may improve clarity.

6. A summary of refinement

The other half of the methodology is the transformation of specifications into programs, which we now address. Suppose we want to make a Pascal program that sorts array b , indexed from 0 to $N-1$, in situ; we may specify the problem as that of implementing " $b := \text{sorted}.b$ " where **sorted** yields the lexicographically least permutation of its argument. We might then proceed to write a first outline thus:

```
i:= 0;
while i<>N do begin
    "k:= index of least value in b[i], b[i+1], ..., b[N-1]";
    "swap b[i], b[k]";
    i:= i+1
end
```

Observe that the texts within quotes are informal specification statements. The above in formal dress is

```
i:= 0;
while i<>N do begin
    k:~ i≤k<N ∧ b[k]=min.b[i..N-1];
    b[i]:= b[k] || b[k]:= b[i];
    i:= i+1
end
```

Next we would proceed to tackle " $k:~ i \leq k < N \wedge b[k] = \text{min}.b[i..N-1]$ " on its own, and then substitute the solution into its place in the body of the loop, and so on. So the specification language also serves as a fancy pseudo-code, that supports the gradual translation of the original specification into a program. This style of programming is what is familiarly known as stepwise refinement; in formal dress it is the application of what we call the "refinement calculus".

A specification t is said to be a "refinement" of specification s , informally speaking, whenever a customer asking for s would be willing to accept t . The two specifications need not be quite functionally equivalent because the refining specification may discard choices offered by the original specification. We will write $s \sqsubseteq t$ to denote " s is refined by t ". For example,

```
b:= sorted.b           ⊆           i:= 0;
                                while i<> N do begin
                                    k:~ i≤k<N ∧ b[k] = min.b[i..N-1];
                                    b[i]:= b[k] || b[k]:= b[i];
                                    i:= i+1
                                end
```

When $s \sqsubseteq t$ and t is a program we say that t "implements" s . By a program we mean a specification that does not contain certain constructs that we deem to be non-algorithmic because they are impossible or too expensive to implement; there is some flexibility here to allow for the ingenuity of the compiler-writer, but we can just think of Pascal-like programs. Note that $[]$ and \rightarrow are part of the programming language, and so is $||$ when its arguments are simple assignments, although we may prefer to refine it away on efficiency grounds. Here are examples of the refinement relation (the variables are of type INTEGER):

- (i) $x:~x>0$ \sqsubseteq $x:= 15$
- (ii) $x:= |x|$ \sqsubseteq $x \leq 0 \rightarrow x := -x \quad [] \quad x \geq 0 \rightarrow \text{skip}$

- (iii) $x:=y \parallel y:=x \sqsubseteq (z: \text{INTEGER}; z:=x; x:=y; y:=z)$
 (iv) $x:=0 \parallel x:=1 \sqsubseteq x:=1$
 (v) $x:=3 \parallel y:=x*x \sqsubseteq y:=x*x; x:=3$

The refinement relation is transitive; this is important because it means that we don't have to implement a specification in one great action, but can proceed from specification to program through a series of small steps and be sure that what we end up with indeed refines the original specification.

If we give a formal semantics to specifications, and we can, then we can formally define refinement and hence derive a collection of useful laws. Among such laws are

- (F) $s \sqsubseteq P \rightarrow s \parallel \neg P \rightarrow s$
 (G) $Q \rightarrow x:\sim Q \sqsubseteq Q \rightarrow \text{skip}$

— (F) allows us to proceed by case analysis, and (G) allows us to replace a non-algorithmic construct with an algorithmic one. The refinement calculus is a collection of such laws. Rather than guess a refinement or implementation and then verify it afterwards, the motivation behind a calculus is that we transform the specification into a program by systematically applying the laws to its components until all the non-algorithmic or unacceptably inefficient constructs have been eliminated. The maker of the laws needs a formal semantics to justify them, but the semantics play no role in their application — for that one just needs to know the laws of logic and the problem domain.

A very desirable property of refinement gives us the ability to refine a specification by refining its components in relative isolation, and then gluing the refinements together with the same structure as the original specification. For example, we should be able to infer from the examples above without further work that

- (vi) $x:\sim x>0; (x:=y \parallel y:=x) \sqsubseteq x:=15; (x:=y \parallel y:=x)$
 (vii) $x:\sim x>0 \parallel x:=y \parallel y:=x \sqsubseteq x:=15 \parallel (z: \text{INTEGER}; z:=x; x:=y; y:=z)$

— refinement (vi) follows from (i) and the fact that semicolon happens to be what we may call "refinement-preserving" in its left argument, and (vii) follows from (i) and (iii) and the fact that choice happens to be refinement-preserving in both its arguments (in fact, semicolon is refinement-preserving in its right argument also). We prefer the shorter word "monotonic" to "refinement-preserving". Formally, an operation $F(x)$ on specifications is monotonic if for all specifications s and t , $F(s) \sqsubseteq F(t)$ whenever $s \sqsubseteq t$. So the statement "semicolon is monotonic in its left argument" means that $s; u \sqsubseteq t; u$ whenever $s \sqsubseteq t$. An implementation can preserve the structure of its specification just to the extent that its combinators are monotonic. We have already argued that structure-preserving implementations are a good thing, so are the combinators we have been using monotonic? The good news is that choice, semicolon, \rightarrow , and parametrisation are monotonic, but parallel composition is not. (The left argument of \rightarrow is an expression, and for expressions refinement is just functional equality.) We will consider the implications of this shortly.

So far we have been talking about what is called "procedural refinement". We will also want to replace some variables with more efficiently implementable ones, and make whatever changes are thereby induced on the statements that act on these variables; this kind of refinement is called "data refinement". For example, in implementing the library catalogue we may want to replace *books* of type set of BOOK with *books1* of type sequence of BOOK, because sets are not part of the programming language. A consequence is that the assignment $\text{books} := \text{books} \cup \{b\}$ must be transformed to, perhaps, the assignment $\text{books1} := \text{books1} + \langle b \rangle$ where $+$ denotes sequence concatenation and $\langle b \rangle$ denotes the sequence containing b only. There is a calculus of data refinement that allows us to make these transformations constructively, by calculation. The calculus is based on a relation \llcorner_I on specifications that may operate on different variables, I being some predicate — called the "abstraction invariant", that relates the two sets of variables. $s \llcorner_I t$ is defined to hold, roughly speaking, whenever the effect of s operating on one set of

variables is mimicked by t operating on the other, where I states the nature of the mimicking. For example, we can derive

$$\text{books} := \text{books} \cup \{b\} \quad \ll_{I_0} \quad \text{books1} := \text{books1} + \langle b \rangle$$

where I_0 is the relation " $\text{books} = \text{the set of values in sequence } \text{books1}$ ". There will usually be several candidates for an abstraction invariant. For example, when replacing books with books1 we might have chosen the stronger predicate I_1 as follows: " $\text{books} = \text{the set of values in sequence } \text{books1}$, and the values in books1 are distinct". This would lead to a somewhat different translation of the statements that refer to books , for example

$$\text{books} := \text{books} \cup \{b\} \quad \ll_{I_1} \quad \begin{array}{l} b \text{ in } \text{books1} \rightarrow \mathbf{skip} \\ [] \neg b \text{ in } \text{books1} \rightarrow \text{books1} := \text{books1} + \langle b \rangle \end{array}$$

where in has the obvious meaning. In any case, once we have decided on the new variables and fixed on the abstraction invariant, the translation of the specification to a similar one operating on the new variables is by a systematic application of the rules of the calculus. The rules are either statements about monotonicity, or they reduce data refinement to procedural refinement; an example of such a rule is

$$(H) \quad s; t \ll_{I_1} u; v \text{ whenever } s \ll_{I_1} u \text{ and } t \ll_{I_1} v$$

— this says we can data refine a composition by data refining its components in isolation.

We can extend the notion of refinement to modules M_1 and M_2 when the "interface" of M_1 is a subset of the interface of M_2 . By the interface of a module we mean the set of pairs $\langle p, TP \rangle$ for each operation name p in the module, where TP denotes the types of its parameters (with an indication of which are output parameters) and result (if any). Refinement among modules does not require that the variables of one module have anything in common with the variables of the other. In fact, there are situations where one wishes to relax the requirement that a refining module share a common interface with its parent, but we will not admit that generality in the present treatment. When modules M_1 and M_2 share a common interface *and* a common set of variables then $M_1 \sqsubseteq M_2$ holds just when each operation of M_1 is refined by the similarly named operation of M_2 ; this is just as we would intuitively expect. More generally, if the interface of M_2 is a superset of the interface of M_1 but it uses a possibly different set of module variables, then $M_1 \sqsubseteq M_2$ holds just when there is a predicate I (the abstraction invariant) relating the two sets of variables such that each operation of M_1 is related to the similarly named operation of M_2 by \ll_{I_1} . In practice we do not prove that one module refines another, but refine modules constructively; we decide on the new set of variables and an abstraction invariant, and then make the new module by operating on the old module with the laws of data and procedural refinement.

For a more comprehensive treatment of the refinement calculus see [1-6].

7. Implementing the library

One way to implement the library described by the specification of Figure 7 is to expand textually all the definitions and **include**'s, arriving at a module in the style of Figure 2, and then implement the procedures and functions. But that is not what we want because, for reasons given in the introduction, we would like to give each module an independent existence, as far as we can.

We will give an informal outline of how we might go about implementing each module, beginning with the catalogue of Figure 3. Examining the module constraints, we observe that they confer some freedom on the implementor. For example, we can maintain the constraint on *title* while strengthening it to $\text{books} = \text{dom.title}$. This will lead to a more space-efficient implementation, so we decide to pursue it.

The theory of data refinement supplies all the laws we need to calculate the new specification. (For interest, the abstraction invariant is $\text{title}_{\text{new}} = \text{title } \mathbf{dr} \text{ books}$ where for any partial function f and set s , $f \mathbf{dr} s$ equals f with those pairs whose first element is not in s removed; the operation \mathbf{dr} is known as "domain restriction".) The refinement results in what one would expect; **removebook0**, for example, becomes

$$\begin{aligned} \mathbf{removebook0} = \\ & b: \text{BOOK}. \\ & \text{catalogued} \rightarrow \text{books:- } \{b\} \parallel \text{title:} \mathbf{ds} \{b\} \bullet \end{aligned}$$

We can strengthen the constraints on *authors* and *subject* similarly. After that, we might choose to get rid of the parallel compositions; it so happens — as will be intuitively evident — that those in Figure 3, as well as those introduced by the strengthening of the invariant, can be replaced with a semicolon.

Examining the parameter and result types of the procedures and functions we see that they are all supplied externally — we are assuming in particular that unbounded sequences are implemented, which is not too unreasonable for the small sequences likely to be involved in the present case. We would have to change the interface of the module if "sequence of" is not a type constructor of the programming language. Next we should consider how to implement the variables. Perhaps we will decide on linked lists of records, one record per book recording its identifier, title, authors, and subject. Having chosen the data structures and abstraction invariant, we apply data refinement to generate the new module. After this, the module will be close to an implementation. In any case, all the combinators are now monotonic, so each little piece can be worked on in isolation. Finally we discard the module invariants. We will have arrived at a module that behaves like that of Figure 3, but can be compiled and executed.

A novel aspect of this implementation style is that the final program will contain procedures whose bodies are partial (i.e. miraculous in some states). When called outside their domain of applicability they do not abort or raise some error condition. Rather they "return" carrying some indication of an attempt at miraculous behaviour, and the system will then look for some other way forward.

Implementing the undoable catalogue of Figure 8 has the extra difficulty of a named operation, **save**, being an argument of the non-monotonic \parallel . We would first have to replace \parallel , carried out in the case of **removebook0**, for example, thus:

```

removebook0
="definition"
  σremovebook0. removebook0 || save
="old definition of removebook0"
  σ removebook0. (catalogued → books:- {b}) || save
="(D) in Section 3"
  σ removebook0. catalogued → books:- {b} || save
="|| commutative"
  σ removebook0. catalogued → save || books:- {b}
="|| may be replaced with semicolon when the variables being assigned to in the
  left hand argument do not appear in the right hand argument"
  σ removebook0. catalogued → save; books:- {b}

```

So we can reuse the old implementation, but we have to "open up" its parent specification. (Note also that we now see it would have been advantageous to have made a named operation of $\text{books:- } \{b\}$ in the specification of Figure 3, and similarly for the body of **addbook0**. We could then have left the replacement of $\text{books:- } \{b\}$ in the above with its implementation to the compiler.) The implementation of the undoable catalogue needs further "inside information" in that the implementation of **save** will have to take account of the implementation of *books* in Figure 3.

The modules of Figures 4 and 6 may be implemented similarly to that of Figure 3. The three modules come together in the library system of Figure 7, whose implementation is now easy, because the modules **include**'d can be replaced by their implementations, without any extra work. The only operation of Figure 7 that has to be implemented is **hasbook**; that can be easily done, but we need to exploit the knowledge of how *has* was implemented when data refining the borrowings module of Figure 6. With hindsight, **hasbook** should probably be moved to the borrowings module, as a companion to **available**.

In summary, we have not quite managed to implement each module independently, but we went a good way towards that goal. What we did achieve was the ability to implement modules in isolation provided we do so in an order consistent with that induced in the obvious way by the **include** clauses. It is pleasing that the old implementations were reusable without change, despite the fact that their constituent operations are altered significantly by the modules that **include** them.

We pay a price for the modular implementation, however. When implementing a module in isolation we can only act on knowledge of the actions of that module alone; we elect not to take into account how other modules interact with the present one. For example, if we implemented the catalogue and borrowings modules in parallel we might choose integrated data structures that would almost certainly be more space-efficient over the system as a whole. Although we are probably prepared to pay a considerable price for the benefits of the modular implementation, the price may sometimes be unacceptably high, and we may have to restructure the specification, at least partly, in the process of implementation.

8. Conclusion

We have outlined a specification methodology that tries to do justice to the dual role of specifications in designing and implementing systems. The method is based on the view that a specification language is an enrichment of a programming language; it admits ease of expression although some of its elements may be unimplementable. The attraction of the approach is that specifications and programs inhabit the same semantic world and so we can easily give a precise meaning to a notion of refinement that supports the implementation of programs by mathematical transformation. We gave a brief outline of this theory, and showed that it could even be applied to construct specifications mechanically once enough of their framework has been set down. The implementation can proceed in small steps, retaining the structure of the parent specification, thanks to the monotonicity of the constructors of the language. The language supports the modular construction and implementation of specifications, with some benefits in the area of reusing specifications and their implementations.

Acknowledgements

The library example is based on a similar one presented by Steve King and Ib Sørensen at the 1988 York Refinement Workshop. The extensive use of partial functions, and the philosophy of specifying incrementally, is borrowed from Z. Thanks to Moira Norrie and David Harper for detailed criticisms of an earlier draft, and to Cliff Jones for critical comments.

References

1. R. J. R. Back, *Correctness Preserving Program Refinements: Proof Theory and Applications*, Tract 131, Mathematisch Centrum, Amsterdam, 1980.
2. R. J. R. Back, A calculus of refinement for program derivations, Report Ser. A 54, Department of Computer Science, Swedish University of Åbo (1987).

3. C. C. Morgan, The specification statement, *ACM TOPLAS* **10** (1988) 403-419.
4. C. C. Morgan and P. H. B. Gardiner, Data refinement by calculation, *Acta Informatica*, to appear.
5. J. M. Morris, Programs from specifications, in *Formal Development of Programs and Proofs*, ed. E. W. Dijkstra, Addison-Wesley, Reading (Mass), 1989.
6. J. M. Morris, Laws of data refinement, *Acta Informatica* **26** (1989) 287-308.
7. J. M. Spivey, *Understanding Z*, Cambridge University Press, Cambridge, 1988.