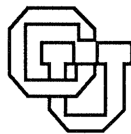
A Formal Specification and Verified Design for Kemmerer's Library Problem

Robert B. Terwilliger

CU-CS-562-91 December 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Formal Specification and Verified Design for Kemmerer's Library Problem

Robert B. Terwilliger

CU-CS-562-91 December 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax

A Formal Specification and Verified Design for Kemmerer's Library Problem

Robert B. Terwilliger

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
email: 'terwilli@cs.colorado.edu'

ABSTRACT

In this paper, we present a formal specification and verified design for a solution to Kemmerer's Library Problem. The problem is concerned with a small database that provides a number of transactions to library users and staff members. The architectural design for our solution consists of a single module that encapsulates the database and provides entry routines for each transaction. The state of the module is modeled abstractly using high-level data types, and the entry routine for each transaction is specified using pre- and post-conditions written in first-order predicate logic. The algorithms that implement each entry routine are described using a guarded command notion, and we verify these designs relative to their specifications using standard proof rules.

1. Introduction

Despite years of effort by both researchers and practitioners the production of software remains both difficult and expensive [7]; software developments remain notoriously difficult to manage and are routinely completed late and/or over budget [10]. There are many opinions as to the essential (or accidental) causes of this problem, and many suggestions for its solution [2]. One proposal that has received considerable attention is the use of *formal methods* [1,9]. In this paper, we present a formal specification and verified design for a solution to Kemmerer's Library Problem [12,20,21].

The basic idea behind formal methods is the use of mathematical abstractions to describe software. Many different languages and systems have been developed that elaborate on this fundamental theme. For example, the Vienna Development Method uses stepwise refinement to transform formal specifications into verified programs [3,11], Z combines a small number of high-level constructs with mathematical data types to provide an implementation independent specification medium [4,18,19], and ANNA extends Ada to support formal specification and run-time assertion checking [14,15].

There are a few key ideas in formal specification. Many software components can be specified by combining pre- and post-conditions with invariants. A *pre-condition* specifies the conditions that must hold before a unit begins execution; for example, the requirements a procedure's input must satisfy. A *post-condition* states the properties that must hold when execution has completed; for example, the relationship between a procedure's input and output. An *invariant* states properties that must always be satisfied by some object, or within some scope; for example, a type declaration might have an associated invariant that specifies the allowable values for the type.

The correctness of a program can be rigorously verified with respect to a formal specification [8,13]. To accomplish this, the semantics of the data types and programming constructs used in the program must first be mathematically defined. The formal verification process decomposes the overall correctness into a number of smaller problems that can be solved using the rules of the definitional system. Programs can be be shown to be either *partially correct* (if they terminate they will be in the correct state), or *totally correct* (they will terminate and be in the correct state).

Kemmerer's Library problem has received considerable attention in the software engineering literature and has been formally specified a number of times [12,20,21]. The problem is concerned with a small library database that provides both query and update transactions to library staff and users. The problem also states some properties that must be maintained during the library's operation. In this paper, we present a formal specification for a solution to the Library Problem, describe a design derived from this specification, and then discuss verifying the total correctness of the design with respect to its specification.

The architectural design for our solution consists of a single module that encapsulates the database and provides entry routines for each transaction. The state

This research was supported by a grant from AT&T, as well as NSF Grant CCR-8809418

of the module is modeled abstractly using high-level data types, and the entry routine for each transaction is specified using pre- and post-conditions written in first-order predicate logic. The algorithms that implement each entry routine are described using a guarded command notion [5,6], and we verify that these designs are totally correct relative to their specifications using standard proof rules [8,13].

In the remainder of this paper we present our treatment of the Library Problem in more detail. In section two we list the original requirements, and in section three we present a formal specification of a solution to the problem. In section four we present a design derived from the specification given in section three, and in section five we discuss it's verification. Finally, in section six we summarize and draw some conclusions from our experience.

2. Requirements

Figure 1 shows the statement of the Library Problem used in [21]. The text is reasonably self explanatory, so we will not belabor the point; however, we should note that the requirements are both ambiguous and incomplete. Fortunately, these problems have been thoroughly analyzed by others. [20] lists the five major ambiguities as follows.

What is a library?
: what is the boundary between the library database (for which a program is to be built) and its environment?

What is a user?
: are staff members a subset of users? how is access to transactions restricted?

What is a book?
: is it a physical object, or an ISBN? can there be more than one copy of a book?

What is "available"?
: can a book be neither checked out or available? if so, how?

What is "last checked out"?
: currently checked out? most recently checked out?

We resolve these issues in the following ways.

A library is a library
: The interface between the library and the rest of the world is not especially well specified. Problems include check out/check in transactions with neither book or user physically present and assignment of unique identifiers.

A staff member is a user
: staff members are a subset of users.

A book is a copy of a book
: different copies are distinct objects.

"Available" is
: not checked out.

"Last checked out" is
: currently checked out.

[20] also describes the six major sources of incompleteness.

Initialization
: how does the library system get loaded with data?

Consider a small library database with the following transactions:

1 - Check out a copy of a book / Return a copy of a book;
2 - Add a copy of a book to / Remove a copy of a book from the library;
3 - Get the list of books by a particular author or in a particular subject area;
4 - Find out the list of books currently checked out by a particular borrower;
5 - Find out what borrower last checked out a particular copy of a book.

There are two types of users; Staff users and ordinary borrowers. Transactions 1,2,4 and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves.

The data base must also satisfy the following constraints:

1 - All copies in the library must be available for checkout or be checked out.
2 - No copy of a book may be both available and checked out at the same time.
3 - A borrower may not have more than a predefined number of books checked out at one time.

Figure 1. The Library Problem

Missing Operations
how do we add or remove users?

Error Handling
what errors should be handled and how?

Missing Constraints
can a book be checked out to more than one person at once?

Change of state
what is and is not changed by each transaction?

Nonfunctional
what does the user interface look like? how fast does it run?

We handle these these issues as follows.

Initializations
are explicit.

Transactions are added:
add_user and remove_user.

Error handling
is ignored.

Additional constraints
are specified.

State changes
are precisely specified.

Nonfunctional requirements
are ignored.

The Library Problem provides a reasonable starting point for the software development process. With the ambiguities and incompleteness is the requirements taken care of, we can move on to the formal specification.

```
package library is

    [       state        ]
    [     invariant      ]
    [ support functions  ]
    [update transactions]
    [query transactions  ]

end library ;
```

Figure 2. Library Database Architecture

3. Specification

The architecture of our solution to the Library Problem is shown in Figure 2. There is a single package that encapsulates the database and provides an entry routine for each transaction. The package state is

```
type book   =  record
               title   : string  ;
               author  : string  ;
               subject: string  ;
               copy    : integer ;
               end book ;
type user   =  record
               name    : string  ;
               staff   : boolean ;
               end user ;

var  books :   set(book) := {} ;
var  users :   set(user)
               := {("super",T)} ;

type vbook =   book
               where b:vbook=>b∈books;
type vuser =   user
               where u:vuser=>u∈users;

type corec =   record
               name : vuser ;
               item : vbook ;
               end corec ;

var checks :   set(corec) := {} where
               c1,c2:corec∈checks =>
                   ( c1.book=c2.book
                       <=> c1=c2 ) ;

invariant    (∀b∈books:
               available(b) ∨
                   is_out(b)) ∧
             ¬(∃b∈books:
               available(b) ∧
                   is_out(b)) ∧
             (∀u∈users:nout(u)≤limit) ;
```

Figure 3. Library State and Invariant

abstractly modeled using high level data types, and there is a package invariant stating properties that must hold both before and after execution of any transaction. The entry routines are specified in terms of the package state using pre- and post-conditions.

Figure 3 shows the specification the library state; it consists of three sets: "books", "users", and "checks". "Books" holds a record for every book in the library. Each "book" record contains the title, author, subject, and copy number of the physical book it represents. Similarly, "users" holds a record for every user in the library system. Each "user" record holds the user's name and a boolean variable signifying whether they are a staff member or regular user. "Books" is initialized to the empty set, while "users" initially contains a single staff user called "super".

The set "checks" holds a record for each book currently on loan from the library. A "corec" records the book checked out as well as the user it is out to. The invariant on "checks" states that no book may be checked out to more than one user at a time. To simplify the rest of the specification, the types "vbook" and "vuser" are defined. A "vbook" (valid book) is simply a book that is currently in the library database (that is an element of "books"), while a "vuser" is a user that is an element of "users".

```
constant    limit : integer := 8 ;

function    nout(u:vuser) : integer ;
   pre      true ;
   post     nout = (N c∈checks:c.name=u);

function    checked_out(   u:vuser;
                           b:vbook
                       ) : boolean ;
   pre      true ;
   post     checked_out =
               (corec(u,b)∈checks) ;

function    is_out(b:vbook) : boolean ;
   pre      true ;
   post     is_out =
            (∃u∈users:checked_out(u,b)));

function    available(b:vbook) : boolean;
   pre      true ;
   post     available = ¬is_out(b) ;

function    active(u:vuser) : boolean ;
   pre      true ;
   post     active =
            (∃b∈books:checked_out(u,b)));
```

Figure 4. Library Support Functions

Figure 3 also shows the invariant specified on the library state. It states that each book is either available or checked out, that no book may be simultaneously available and checked out, and that each user has no more than his limit of books checked out at any time. This invariant must be established by package initialization, and must be maintained by all the entry routines. It uses a number of support functions.

Figure 4 shows the support functions defined on the library state (note that functions are not allowed to modify their arguments). "Limit" is the maximum number of books a user may have checked out, while "nout" is the number he currently has checked out (the notation (N...) denotes the "number of" quantifier [8]). "Checked_out(u,b)" is true if book "b" is checked out to user "u", while "is_out(b)" is true if book "b" is checked out to any user. "Available(b)" is true if book "b" is not checked out, and "active(u)" is true if user "u" has no books currently checked out.

Figure 5 shows the specification of the update transactions on the database. Note that parameter modes are specified as either "in", "out", or "inout". "In" parameters must have a value when the routine is invoked and cannot be modified during execution. "Out" parameters are set by the called procedure. "Inout" parameters must have a value on invocation and may be modified by the routine (it is assumed that they are passed using a copy-restore strategy). The "modify" clause describes the global variables modified by the routine.

The procedures "add_book" and "remove_book" add and remove a book from the library respectively, while "add_user" and "remove_user" perform the same functions for users. The "check_out" and "check_in" transactions modify the database to reflect the borrowing and return of of a book respectively. Access to the update transactions is restricted to library staff members. The first parameter of each routine is the user performing the transaction, and the pre-condition requires that this be a staff member.

For example, the "check_out" routine has three "in" parameters. The first is the person performing the transaction, the second is the borrower, and the third is the book in question. The pre-condition for "check_out" states that the book must be available, that the borrower must be under their check out limit, and that the initial value of "checks" is called "CHECKS". This allows the post-condition to state that the final value of "checks" is equal to the initial value with the

```
procedure  add_book(  s:in vuser;
                      b:in book) ;
    pre    s.staff ∧
           b∉books ∧ books=BOOKS ;
    post   books=BOOKS+b ;
    modify books ;

procedure  remove_book(   s:in vuser;
                          b:in vbook) ;
    pre    s.staff ∧
           available(b) ∧ books=BOOKS ;
    post   books=BOOKS-b ;
    modify books ;

procedure  add_user(  s:in vuser;
                      u:in user) ;
    pre    s.staff ∧
           u∉users ∧ users=USERS ;
    post   users=USERS+u ;
    modify users ;

procedure  remove_user(   s:in vuser;
                          u:in vuser) ;
    pre    s.staff ∧
           ¬active(u) ∧ users=USERS ;
    post   users=USERS-u ;
    modify users ;

procedure  check_out( s,u:in vuser;
                      b  :in vbook) ;
    pre    s.staff ∧
              available(b) ∧
              nout(u)<limit ∧
              checks=CHECKS ;
    post   checks=CHECKS+corec(u,b) ;
    modify checks ;

procedure  check_in(  s,u:in vuser;
                      b  :in vbook) ;
    pre    s.staff ∧
              checked_out(u,b) ∧
              checks=CHECKS ;
    post   checks=CHECKS-corec(u,b) ;
    modify checks ;
```

Figure 5. Update Transactions

addition of a "corec" with "u" and "b" in the "name" and "item" fields respectively.

Figure 6 shows the specification of the query transactions on the database. The "by_author" and "on_subject" functions may be initiated by any user.

```
function by_author( a:string
                  ) : set(vbook) ;
    pre  true ;
    post by_author =
           {b∈books:b.author=a} ;

function on_subject( sb:string
                   ) : set(vbook) ;
    pre  true ;
    post on_subject =
           {b∈books:b.subject=sb} ;

function what_out( s,u:vuser
                 ) : set(vbook) ;
    pre  s.staff ∨ s=u ;
    post what_out =
           {b∈books:corec(u,b)∈checks};

function who_has( s:vuser ; b:vbook
                ) : set(vuser);
    pre  s.staff ;
    post who_has =
           {u∈users:corec(u,b)∈checks};
```

Figure 6. Query Transactions

They return the set of all books with a specific author or on a specific subject respectively. The "what_out" function is limited to staff members, except that ordinary users may invoke it with themselves as the second argument. It returns the list of books currently checked out to a particular user. "Who_has" may only be invoked by staff members and returns the set of all users who currently have a particular book checked out.

The specification is now complete. Although it is simple, it provides a suitable base for development of a design. In fact, using the the methods described in [8] the design can be derived from the specifications.

4. Design

For the purpose of this paper, the design of our solution will consist of algorithms that implement each routine specified in section three. These algorithms are written using guarded commands [5, 6, 8], use the same data types as their specifications, and are correct with respect to them. All of these algorithms are fairly simple and fall into two categories. The *simple routines* can be implemented using a single assignment statement, while the *complex routines* require a loop.

```
function    checked_out(   u:vuser;
                           b:vbook
                         ) : boolean is
    {Q:true}
    checked_out := (corec(u,b)∈checks);
    {R:checked_out=(corec(u,b)∈checks)}

function available(b:vbook): boolean is
    {Q:true}
    available := ¬is_out(b);
    {R:available = ¬is_out(b)}

procedure   add_book(   s:in vuser;
                        b:in book) is
    {Q:s.staff ∧ b∉books ∧ books=BOOKS}
    books := books+b;
    {R:books=BOOKS+b}

procedure   remove_book(   s:in vuser;
                           b:in vbook) is
    {Q:s.staff ∧
       available(b) ∧ books=BOOKS}
    books := books-b;
    {R:books=BOOKS-b}

procedure   add_user(   s:in vuser;
                        u:in user) is
    {Q:s.staff ∧ u∉users ∧ users=USERS}
    users := users+u;
    {R:users=USERS+u}

procedure   remove_user(   s:in vuser;
                           u:in vuser) is
    {Q:s.staff ∧ ¬active(u) ∧ users=USERS}
    users := users-u;
    {R:users=USERS-u}

procedure   check_out( s,u:in vuser;
                       b  :in vbook) is
    {Q:s.staff ∧ available(b) ∧
       nout(u)<limit ∧ checks=CHECKS}
    checks := checks+corec(u,b);
    {R:checks=CHECKS+corec(u,b)}

procedure   check_in(  s,u:in vuser;
                       b  :in vbook) is
    {Q:s.staff ∧
       checked_out(u,b) ∧ checks=CHECKS}
    checks := checks-corec(u,b);
    {R:checks=CHECKS-corec(u,b)}
```

Figure 7. Simple Routine Designs

4.1. Simple Routines

Figure 7 shows the design of the simple routines; each consists of a single assignment statement. For example, "checked_out" should return true if and only if a record showing that "u" has "b" checked out is an element of "checks". The function's body consists of a single assignment that sets the return value of the function (which has the same name as the function itself) to the result of the appropriate membership test on "checks". The design of "available" is even simpler, as the return value is just the negation of the value returned by a call to "is_out".

The body of "add_book" simply sets "books" to its initial value plus "b", while "remove_book" sets "books" to the initial value with "b" removed. The design of "add_user" simply states that "users" is set to its value upon entry to the routine plus the user to be added, while the body of "remove_user" assigns to "users" its original value with "u" removed. The design of "check_out" states that a new "corec" with "u" and "b" in the "name" and "item" fields respectively is added to the set "checks". The body of "check_in" will set "checks" to its value on entry to the routine, minus any records stating that "b" is checked out to "u".

4.2. Complex Routines

Not all the routines can be implemented so simply. The designs of "nout", "is_out", "active", "by_author", "on_subject", "what_out", and "who_has" require the use of a loop with an embedded conditional. All of the complex routines are functions. In each case, computation of the desired result involves processing each element of a set in turn. A local set variable holds all the items still to be processed, while a local scalar holds the item currently under examination. The result variable is initialized to the identity element before the loop begins, and each iteration modifies the result depending on whether the item under examination satisfies a certain property.

For example, Figure 8 shows the design of three complex routines. The function "nout" computes the number of books currently checked out to the user "u". The body of "nout" consists of a single loop with an embedded conditional. Two local variables are declared. "Chks" is a set containing all the records still to be considered, while "chk" is the record currently being processed; therefore, "chks" is initialized to "checks" and the result to zero. The loop interates over all the records in "checks". If the record under question has "u" in the "name" field then the count is incremented, otherwise nothing is done. When all records have been considered, the correct count has been calculated.

The functions "is_out" and "active" are similar, but differ in that an existential quantifier defines the

```
function nout(u:vuser) : integer is
    {Q:true}
    var chk:corec; chks:set(corec);
    chks,nout:=checks,0 ;
    {inv P:chks⊆checks ∧
         nout=(Nc∈checks-chks:c.name=u)}
    {bnd t: |chks|}
    do chks≠{} →
        choose(chks,chk); chks:=chks-chk;
        if chk.name=u → nout:=nout+1
         [] chk.name≠u → skip ;
        fi
    od
    {R:nout=(Nc∈checks:c.name=u)}

function is_out(b:vbook) : boolean is
    {Q:true}
    var usr:user; usrs:set(user);
    is_out,usrs:=false,users ;
    {inv P:usrs⊆users ∧ is_out =
         (∃u∈users-usrs:checked_out(u,b))}
    {bnd t: |usrs|}
    do usrs≠{} →
        choose(usrs,usr); usrs:=usrs-usr;
        if  checked_out(usr,b) →
            is_out:=true ;
         [] ¬checked_out(usr,b) → skip;
        fi
    od
    {R:is_out=(∃u∈users:checked_out(u,b))}

function active(u:vuser) : boolean is
    {Q:true}
    var bk:book; bks:set(book);
    active,bks:=false,books ;
    {inv P:bks⊆books ∧ active =
         (∃b∈books-bks:checked_out(u,b))}
    {bnd t:|bks|}
    do bks≠{} →
        choose(bks,bk); bks:=bks-bk;
        if  checked_out(u,bk) →
            active:=true ;
         [] ¬checked_out(u,bk) → skip;
        fi
    od
    {R:active=(∃b∈books:checked_out(u,b))}
```

Figure 8. Complex Routine Designs(1)

```
function by_author( a:string
                  ) : set(vbook) is
    {Q:true}
    var bk:book; bks:set(book);
    by_author,bks:={},books ;
    {inv P:bks⊆books ∧ by_author =
         {b∈books-bks:b.author=a}}
    {bnd t:|bks|}
    do bks≠{} →
        choose(bks,bk); bks:=bks-bk;
        if bk.author=a →
            by_author:=by_author+bk;
         [] bk.author≠a → skip;
        fi
    od
    {R:by_author={b∈books:b.author=a}}

function on_subject( sb:string
                   ) : set(vbook) is
    {Q:true}
    var bk:book; bks:set(book);
    on_subject,bks:={},books ;
    {inv P:bks⊆books ∧ on_subject =
         {b∈books-bks:b.subject=sb}}
    {bnd t:|bks|}
    do bks≠{} →
        choose(bks,bk); bks:=bks-bk;
        if bk.subject=sb →
            on_subject:=on_subject+bk;
         [] bk.subject≠sb → skip;
        fi
    od
    {R:on_subject={b∈books:b.subject=sb}}
```

Figure 9. Complex Routine Designs(2)

correct result. For example, "is_out" returns true if and only if there is some user who has "b" checked out; in other words, if for some "u" in "users" "checked_out(u,b)" is true. The loop iterates over the set "users", and the result is initialized to false (the identity for $\exists$). If a user with the correct property is found, the result is set to true, otherwise no action is taken. As before, the correct result is calculated.

Figure 9 Shows the design of more complex routines. "By_author" and "on_subject" are similar to the routines previously discussed, but differ in that they compute subsets based on a property. For example, "by_author" returns the set of all books that were written by a particular author; in other words, the set of all "book" records in "books" that have "b.author=a". The loop iterates over the set "books", and the result is

initialized to the empty set (the identity for the "set_of_all" operator). If a book is found with the desired author, it is added to the result, otherwise nothing is done. As always, the correct result is calculated. Figure 10 shows the designs for more complex query routines; They are similar to those in Figure 9, and we will not discuss them further.

The design of our solution is now complete. Although it still uses the high-level data types found in the specification, it represents a considerable shift in abstraction level. We will now turn to a discussion of its correctness.

```
function what_out( s,u:vuser
                  ) : set(vbook) is
    {Q: s.staff ∨ s=u}
    var bk:book; bks:set(book);
    what_out,bks:={},books ;
    {inv P: bks⊆books ∧ what_out =
          {b∈books-bks:corec(u,b)∈checks}}
    {bnd t: |bks|}
    do bks≠{} →
        choose(bks,bk); bks:=bks-bk;
        if corec(u,bk)∈checks →
            what_out:=what_out+bk ;
         [] corec(u,bk)∉checks → skip;
        fi
    od
    {R: what_out =
        {b∈books:corec(u,b)∈checks}}

function who_has( s:vuser ; b:vbook
                ) : set(vuser) is
    {Q: s.staff}
    var usr:user; usrs:set(user);
    who_has,usrs:={},users ;
    {inv P:usrs⊆users ∧
          who_has = {u∈users-usrs:
                         corec(u,b)∈checks}}
    {bnd t: |usrs|}
    do usrs≠{} →
        choose(usrs,usr); usrs:=usrs-usr;
        if corec(usr,b)∈checks →
            who_has:=who_has+usr ;
         [] corec(usr,b)∉checks → skip;
        fi
    od
    {R: who_has =
        {u∈users:corec(u,b)∈checks}}
```

Figure 10. Complex Routine Designs(3)

5. Verification

We contend that the design presented in section four is totally correct with respect to the specification presented in section three. The proof consists of three main parts. First, we show that the truth of the package invariant is implied by the routine specifications; therefore, we can ignore it during the rest of the verification. Second, we demonstrate that each simple routine satisfies its specification, and third we prove that each complex routine does the same. We present the proof of correctness in detail for one member of each class of routines and leave it to the reader to modify these proofs for the other members.

5.1. Package Invariant

The package invariant must hold both before and after execution of each entry routine for the module. It is given in Figure 3.

```
(∀b∈books:available(b) ∨ is_out(b)) ∧
¬(∃b∈books:available(b) ∧ is_out(b)) ∧
(∀u∈users:nout(u)≤limit)
```

The invariant states that all books must be either available or checked out, no book may be both available and checked out, and that all users must have no more than limit books checked out at any time.

The first two of these properties follow from the definitions of available and is_out (available(b) = ¬is_out(b)). The third property is initialized correctly (no one has anything checked out), and can only be invalidated by check_out (it is the only routine that adds check out records). Since the pre-condition of check_out requires that the user have less than limit books checked out, it guarantees that the invariant is maintained. Since the truth of the invariant can be derived from the rest of the specification, we will not consider it further.

5.2. Proof Rules

Figure 11 shows proof rules for the total correctness of the constructs used in our designs [8,13]. Rules one and two are concerned with simple statements. Rule one states that the statement skip terminates in a state where R is true if it was started in a state where R is already true. Rule two says that an assignment x:=e is totally correct with respect to pre-condition Q and post-condition R if Q implies R with e substituted for x.

Rules three and four are concerned with structuring statements into larger constructs. Rule three states that the statement sequence $S_1;S_2$ is totally correct with

1) $\{R\}$ skip $\{R\}$

2) if $Q \Rightarrow R^x_e$
then $\{Q\}\ x:=e\ \{R\}$

3) if 1) $\{Q\}\ S_1\ \{R'\}$
2) $\{R'\}\ S_2\ \{R\}$
then $\{Q\}\ S_1;S_2\ \{R\}$

4) if 1) $\{Q'\}\ S\ \{R'\}$
2) $Q \Rightarrow Q'$
3) $R' \Rightarrow R$
then $\{Q\}\ S\ \{R\}$

5) let IF = if $B_1 \rightarrow S_1$
$[]\ B_2 \rightarrow S_2$
•
•
$[]\ B_n \rightarrow S_n$
fi
BB = $(\exists k{:}1 \le k \le n{:}B_k)$

if 1) $Q \Rightarrow BB$
2) $\{Q \wedge B_k\}\ S_k\ \{R\}$, $1 \le k \le n$
then $\{Q\}$ IF $\{R\}$

6) let DO = {invariant P}
{bound t}
do $B_1 \rightarrow S_1$
$[]\ B_2 \rightarrow S_2$
•
•
$[]\ B_n \rightarrow S_n$
od
BB = $(\exists k{:}1 \le k \le n{:}B_k)$

if 1) $Q \Rightarrow P$
2) $\{P \wedge B_k\}\ S_k\ \{P\}$, $1 \le k \le n$
3) $P \wedge \neg BB \Rightarrow R$
4) $P \Rightarrow (t \ge 0)$
5) $\{P \wedge B_k\}\ t1{:=}t;\ S_k\ \{t < t1\}$, $1 \le k \le n$
then $\{Q\}$ DO $\{R\}$

Figure 11. Proof Rules

respect to pre-condition Q and post-condition R if there exists a formula R' such that S_1 and S_2 are correct with respect to Q,R' and R',R respectively. Rule four is sometimes called the consequence rule. It says that if we know that a program is correct with respect to pre- and post-conditions Q' and R', that Q implies Q', and that R' implies R, then the program is correct with respect to Q and R.

Rule five lets us demonstrate the correctness of if statements. IF is used as an abbreviation for the entire alternative command and BB represents the disjunction of all the guards. The rule states that if the pre-condition implies that at least one of the guards is true, and if each guarded command (S_k) will produce a state that satisfies the post-condition when started in a state that satisfies both the pre-condition and the appropriate guard ($Q \wedge B_k$), then then entire statement is correct with respect to the pre- and post-conditions.

Rule six shows us how to prove the correctness of a loop. The operation of the loop is specified using an invariant P and bound function t. The invariant describes the properties that are true both before and after each execution of the loop, while the bound function is an upper limit on the number of iterations remaining. The rule has five conditions. The first three are concerned with the invariant, while the last two address the bound.

The first requirement is that the the pre-condition implies the invariant; in other words, the invariant is true before the loop begins execution. The second condition is that the body of the loop maintains the invariant. To be more specific, each guarded command in the loop must leave a state that satisfies the invariant when started in a state that satisfies both the invariant and the corresponding guard. The third requirement is that if the loop terminates with the invariant true, then the state produced will satisfy the post-condition.

The fourth condition is that the bound function is at least zero while the loop is running. This simply shows that the function has a lower bound and so cannot decrease forever. The fifth condition is that each execution of the loop body must decrease the bound function. For the purpose of the proof, a temporary variable t1 is used to hold the value of the bound when execution of the body begins. The bound function must be less than this value when execution is complete.

We can use these rules to verify the designs of both the simple and complex routines.

5.3. Simple Routines

Each simple routine is implemented using a single assignment. For example, the design of checked_out is as follows.

```
function    checked_out(   u:vuser;
                           b:vbook
                         ) : boolean is
    {Q:true}
    checked_out := (corec(u,b)∈checks);
    {R:checked_out=(corec(u,b)∈checks)}
```

Rule two in Figure 11 tells us that {Q} x:=e {R} is true if Q => R^x_e. In the case of checked_out, the design satisfies its specification if

true => (corec(u,b)∈checks)=(corec(u,b)∈checks).

This reduces to true, so the design is correct. All the simple routines have similar, simple proofs.

5.4. Complex Routines

The proofs of the complex routines are considerably larger and more involved. For example, Figure 12 shows a fully annotated version of the design for the who_has function. The top level structure of the proof is as follows.

{Q'} I {Q} DO {R} {R'}

To prove the design of the function is correct, we must prove that the initialization sets up the loop in the proper manner ({Q'} I {Q}), that the loop is correct ({Q} DO {R}), and that correct termination of the loop ensures the post-condition for the routine is satisfied (R => R').

The proof of the initialization uses the assignment rule in Figure 11 and is similar to the proof of a simple routine.

Q' => $Q^{\text{who_has,usrs}}_{\{\},\text{users}}$
Q' => { },users={ },users

We can see that R implies R' by simply expanding their definitions.

R => R'
P ∧ usrs={ } => R'
who_has={u∈users-usrs:corec(u,b)∈checks} ∧
  usrs={ } =>
    who_has={u∈users:corec(u,b)∈checks}

The proof of the loop is reasonably straight forward, but a bit more complicated. It uses three lemmas that we will prove before proceeding.

```
function who_has( s:vuser ; b:vbook
                ) : set(vuser) is
  {Q': s.staff}
  var usr:user; usrs:set(user);
  who_has,usrs:={},users ;
  {Q: who_has={} ∧ usrs=users}
  {inv P:usrs⊆users ∧
       who_has = {u∈users-usrs:
                  corec(u,b)∈checks}}
  {bnd t: |usrs|}
  do usrs≠{} →
     {Q1: P ∧ usrs≠{} ∧
          usrs=USRS ∧ who_has=WHO_HAS}
     choose(usrs,usr); usrs:=usrs-usr;
     {Q2: who_has=WHO_HAS ∧
          Q2': USRS⊆users ∧
               WHO_HAS =
                {u∈users-USRS:
                 corec(u,b)∈checks} ∧
               usrs=USRS-usr ∧
               usr∈USRS}
     if corec(usr,b)∈checks →
          who_has:=who_has+usr ;
      [] corec(usr,b)∉checks →
          skip ;
     fi
     {R2: Q2' ∧
          (corec(usr,b)∈checks ∧
             who_has=WHO_HAS+usr ∨
          corec(usr,b)∉checks ∧
             who_has=WHO_HAS)}
  od
  {R: P ∧ usrs={}}
  {R': who_has =
     {u∈users:corec(u,b)∈checks}}
```

Figure 12. Annotated Design of *Who_has*

Lemma 1: {Q1} S1 {Q2}
{Q1} choose(usrs,usr) {Q1 ∧ usr∈usrs}
{Q1 ∧ usr∈usrs} usrs:=usrs-usr {Q2}
Q1 ∧ usr∈usrs
=> $Q2^{\text{usrs}}_{\text{usrs-usr}}$
=> who_has=WHO_HAS ∧
USRS⊆users ∧
WHO_HAS =
{u∈users-USRS:
corec(u,b)∈checks} ∧
usrs-usr=USRS-usr ∧
usr∈USRS

Lemma 2: $\{Q2\}$ IF $\{R2\}$

1) $Q2 \Rightarrow BB$
   $Q2 \Rightarrow corec(usr,b) \in checks \lor$
   $\quad corec(usr,b) \notin checks$

2.1) $\{Q2 \land corec(usr,b) \in checks\}$
   $who_has := who_has + usr$;
   $\{R2\}$
   $Q2 \land corec(usr,b) \in checks$
   $\quad \Rightarrow R2^{who_has}_{who_has+usr}$
   $\quad \Rightarrow Q2' \land$
   $\quad\quad (corec(usr,b) \in checks \land$
   $\quad\quad\quad who_has+usr =$
   $\quad\quad\quad\quad WHO_HAS+usr \lor$
   $\quad\quad corec(usr,b) \notin checks \land$
   $\quad\quad\quad who_has+usr = WHO_HAS)$

2.2) $\{Q2 \land corec(usr,b) \notin checks\}$ skip $\{R2\}$
   $Q2 \land corec(usr,b) \notin checks \Rightarrow R2$
   $\quad \Rightarrow Q2' \land$
   $\quad\quad (corec(usr,b) \in checks \land$
   $\quad\quad\quad who_has =$
   $\quad\quad\quad\quad WHO_HAS+usr \lor$
   $\quad\quad corec(usr,b) \notin checks \land$
   $\quad\quad\quad who_has = WHO_HAS)$

   therefore, $\{Q2\}$ IF $\{R\}$

Lemma 3: $R2 \Rightarrow P$
   $R2 \Rightarrow Q2'$
   $\quad \Rightarrow USRS \subseteq users \land$
   $\quad\quad T1:(usrs = USRS - usr \land usr \in USRS)$
   $\quad \Rightarrow P1: usrs \subseteq users$
   $R2 \Rightarrow Q2'$
   $\quad \Rightarrow T2: WHO_HAS =$
   $\quad\quad\quad \{u \in users - USRS : corec(u,b) \in checks\}$
   $R2 \Rightarrow T3: (corec(usr,b) \in checks \land$
   $\quad\quad who_has = WHO_HAS+usr) \lor$
   $\quad T4: (corec(usr,b) \notin checks \land$
   $\quad\quad who_has = WHO_HAS)$
   $T1 \land T2 \land T3 \Rightarrow$
   $\quad P2: who_has = \{u \in users - usrs : corec(u,b) \in checks\}$
   $T1 \land T2 \land T4 \Rightarrow P2$
   $P1 \land P2 \Rightarrow P$
   $R2 \Rightarrow P2$

   therefore, $R2 \Rightarrow P$

Using these three lemmas, we can prove the correctness of the loop, following the rule in Figure 11.

1) $Q \Rightarrow P$
   $who_has = \{\} \land usrs = users \Rightarrow P^{who_has,usrs}_{\{\},users}$
   $Q \Rightarrow users \subseteq users \land$
   $\quad\quad \{\} = \{u \in users - users : corec(u,b) \in checks\}\}$

2) $\{P \land B\}\ S\ \{P\}$
   $\{P \land B\}\ \{Q1\}\ S1\ \{Q2\}$ IF $\{R2\}\ \{P\}$
   $\quad P \land users \neq \{\} \Rightarrow Q1$
   $\quad \{Q1\}\ S1\ \{Q2\}$
   $\quad \{Q2\}$ IF $\{R2\}$
   $\quad R2 \Rightarrow P$

3) $P \land \neg BB \Rightarrow R$
   $P \land \neg(usrs \neq \{\}) \Rightarrow P \land usrs = \{\}$

4) $P \Rightarrow (t \geq 0)$
   $P \Rightarrow |usrs| \geq 0$

5) $\{P \land B\}\ t1 := t;\ S1;$ IF $\{t < t1\}$
   $\{P \land B\}\ t1 := t\ \{t1 = t\}\ S1\ \{t < t1\}$ IF $\{t < t1\}$
   $\quad \{P \land B\}\ t1 := t\ \{t1 = t\}$
   $\quad \{t1 = t\}\ choose(usrs,usr)\ \{t1 = t \land usr \in usrs\}$
   $\quad \{t1 = t \land usr \in usrs\}\ usrs := usrs - usr\ \{t < t1\}$
   $\quad\quad usr \in usrs \Rightarrow |usrs - usr| < |usrs|$
   $\quad \{t < t1\}$ IF $\{t < t1\}$

Therefore $\{Q\}$ DO $\{R\}$

The proof of the loop is now complete, and with it the proof of the "who_has" function. The elaboration is reasonably lengthy, but follows from the basic principles fairly simply. We leave it to the reader to modify this proof for the "nout", "is_out", "active", "by_author", "on_subject", and "what_out" routines.

6. Summary and Conclusions

It has been suggested that the use of *formal methods* may help reduce the cost of software development and maintenance [1,9]. In this paper, we have presented a formal specification and verified design for a solution to Kemmerer's Library Problem [12,20]. The problem describes a small library database that provides both query and update transactions. The database has two classes of users: library staff and ordinary borrowers. Only staff members are allowed access to some transactions. The problem also states some properties that must be maintained during the library's operation.

The formal specification for our solution describes a single module that encapsulates the database and provides an entry routine for each transaction. The state of the module is modeled abstractly using high-level data types, and the entry routine for each transaction is specified using pre- and post-conditions. Our experience in writing the formal specification was very favorable. It was reasonably easy to construct and helped resolve the ambiguity and incompleteness in the problem statement.

Our design consists of an algorithm to implement each routine in the specification. These algorithms are written using guarded commands [5,6] and the data types used in the specification. The design was constructed quite easily using a small number of coding cliches; unfortunately, it is not especially useful. The high-level data types used are not available in most implementation languages; therefore, the design may have little to say about efficient implementation algorithms. It would have been much more effective to choose implementation data structures before performing algorithm design.

Our proof that these algorithms are totally correct with respect to their specifications uses standard rules [8, 13]. Our experience with verifying the design was primarily negative. Constructing the proofs was tedious and time consuming without providing much new insight into the correctness of the algorithms. We feel that verification was the least time-effective aspect of the exercise. It would have been much more efficient to verify the small number of coding cliches we used and then apply them in a rigorous manner.

At the present time we agree with Hall [9] that the most effective use of formal methods is for specification. We feel that the next best use is the generation of designs. The practicality of this process may dramatically increase as the current generation of automatic programming systems [16, 17] move from research into industrial settings. We believe that the least effective use of formal methods is for detailed verification; unfortunately, it is likely to remain extremely expensive for the immediate future.

7. References

1. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
2. Brooks, F. P., "No Silver Bullet", *IEEE Computer 20, 4* (April 1987), 10-19.
3. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.
4. Delisle, N. and D. Garlan, "A Formal Specification of an Oscilloscope", *IEEE Software 7*, 5 (September 1990), 29-36.
5. Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of the ACM 18*, 8 (August 1975), 453-457.
6. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.
7. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
8. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
9. Hall, A., "Seven Myths of Formal Methods", *IEEE Software 7*, 5 (September 1990), 11-19.
10. Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, Reading, Massachusetts, 1989.
11. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
12. Kemmerer, R. A., "Testing Formal Specifications to Detect Design Errors", *IEEE Transactions on Software Engineering SE-11, 1* (January 1985), 32-43.
13. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
14. Luckham, D. C. and F. W. Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software 2, 2* (March 1985), 9-22.
15. Luckham, D. C., F. W. Henke and B. Krieg-Bruckner, *ANNA - A Language for Annotating Ada Programs (Reference Manual)*, Springer-Verlag Lecture Notes in Computer Science, Vol. 260, New York, 1987.
16. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
17. Rich, C. and R. C. Waters, "Automatic Programming: Myths and Prospects", *IEEE Computer 21*, 8 (August 1988), 40-51.
18. Spivey, J. M., *The Z Notation*, Prentice Hall, New York, 1989.
19. Spivey, J. M., "Specifying a Real-Time Kernel", *IEEE Software 7*, 5 (September 1990), 21-28.
20. Wing, J. M., "A Study of 12 Specifications of the Library Problem", *IEEE Software 5*, 4 (July 1988), 66-76.
21. *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION