A Formal Specification
and Verified Design for
Kemmerer's Library Problem
Robert B. Terwilliger
CU-CS-562-91 December 1991
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 ANN 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 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
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 imple-
ment 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 its 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 ambigu-
ous 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 avail-
able? 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.

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 particu-
lar 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 per-
form transaction 4 to find out the list of books current-
ly borrowed by themselves.

The database must also satisfy the following con-
straints:

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

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 incomple-
teness.

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

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

```plaintext
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) ;

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

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

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

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

Figure 2. Library Database Architecture

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 of 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".

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 "CHFCKS". This allows the post-condition to state that the final value of "checks" is equal to the initial value with the
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.

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.
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 iterates 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: user) : integer is
    {Q: true}
    var chk: core; chks: set(core); chks, nout := chks, 0;
    {inv P: chks\epsilon chks \land nout=(\exists \epsilon chks . c.name=u)}
    {bdn t: |chks|}
    do chks\neq{} ->
      choose(chks, chk); chks := chks - chk;
      if chk . name = u -> nout := nout + 1
      [] chk . name\neq u -> skip;
      fi
    od
    {R: nout=(\exists \epsilon chks . c.name=u)}

function is_out(b: vbook) : boolean is
    {Q: true}
    var usr: user; usrs: set(user);
    is_out, users := false, users;
    {inv P: users=\epsilon users \land is_out =
      (\exists \epsilon users . usrs . checked_out(u, b))}
    {bdn t: |usrs|}
    do users\neq{} ->
      choose(users, usr); users := users - usr;
      if checked_out(users, b) ->
        is_out := true;
      [] ~ checked_out(users, b) -> skip;
      fi
    od
    {R: is_out=(\exists \epsilon users . checked_out(u, b))}

function active(b: vbook) : boolean is
    {Q: true}
    var bk: book; bkvs: set(book);
    active, bkvs := false, books;
    {inv P: bkvs\epsilon bkvs \land active =
      (\exists \epsilon bkvs . bk . checked_out(u, b))}
    {bdn t: |bkvs|}
    do bkvs\neq{} ->
      choose(bkvs, bk); bkvs := bkvs - bk;
      if checked_out(bkvs, bk) ->
        active := true;
      [] ~ checked_out(bkvs, bk) -> skip;
      fi
    od
    {R: active=(\exists \epsilon bkvs . checked_out(u, b))}

Figure 8. Complex Routine Designs(1)

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.

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.

\[
\forall b \in \text{books:available}(b) \land \text{is_out}(b) \rightarrow \\
\neg (\exists b \in \text{books:available}(b) \land \text{is_out}(b)) \land \\
\forall u \in \text{users:nout}(u) \leq \text{limit}
\]

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 (\(\text{available}(b) = \neg \text{is_out}(b)\)). 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 \( Q \Rightarrow R' \) 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
   i
   .
   .
   .
   i
   fi
   BB = (\exists k: 1 \leq k \leq n: B_k)
   if 1) Q \Rightarrow BB
   2) \{ Q \land B_k \} S_k \{ R \}, 1 \leq k \leq n
   then \{ Q \} IF \{ R \}

6) let DO = \{ invariant P \}
   \{ bound t \}
   do B_1 \rightarrow S_1
   i
   .
   .
   .
   i
   od
   BB = (\exists k: 1 \leq k \leq n: B_k)
   if
   1) Q \Rightarrow P
   2) \{ P \land B_k \} S_k \{ P \}, 1 \leq k \leq n
   3) P \land \neg BB \Rightarrow R
   4) P \Rightarrow (t \geq 0)
   5) \{ P \land B_k \} t_1 := t; S_k \{ t < t_1 \}, 1 \leq k \leq n
   then \{ Q \} DO \{ R \}

Figure 11. Proof Rules

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 \land B_k), then the 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 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
\[ Q : \text{true} \]
\[ Q \rightarrow (\text{corec}(u,b) \in \text{checks}) \]
\[ R : \text{checked\_out} = (\text{corec}(u,b) \in \text{checks}) \]

Rule two in Figure 11 tells us that \( Q \Rightarrow R \) is true if \( Q \Rightarrow R \). In the case of checked\_out, the design satisfies its specification if

\[ \text{true} \Rightarrow (\text{corec}(u,b) \in \text{checks}) = (\text{corec}(u,b) \in \text{checks}) \]

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 (\( R \Rightarrow R' \)).

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} \}
\[ Q' = \{ \}, \text{users} = \{ \}, \text{users} \]

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

\[ R \Rightarrow R' \]
\[ P \land \text{users} = \{} \Rightarrow R' \]
\[ \text{who\_has} = \{ \text{u:users-users:corec}(u,b) \in \text{checks} \} \land \]
\[ \text{users} = \{ \} \Rightarrow \]
\[ \text{who\_has} = \{ \text{u:users:corec}(u,b) \in \text{checks} \} \]

The proof of the loop is reasonably straightforward, 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.\text{staff} \}
\[ \text{var} \ u:\text{user}; \ u:\text{set}(\{\text{user}\}) \]
\[ \text{who\_has, users} = \{ \} \land \text{users} = \{ \}
\[ \{ Q : \text{who\_has} = \{ \} \land \text{users} = \{ \} \}
\[ \{ \text{inv} \ P : \text{users} = \{ \}
\[ \text{who\_has} = \{ \text{u:users-users:corec}(u,b) \in \text{checks} \} \}
\[ \{ \text{bd t : users} \}
\[ \text{do users\#}() \rightarrow \]
\[ \{ Q1 : P \land \text{users\#}() \land \]
\[ \text{users = USRS} \land \text{who\_has} = \text{WHO\_HAS} \}
\[ \text{choose(users,usr); users = users-usr;}
\[ \{ Q2 : \text{who\_has} = \text{WHO\_HAS} \land \]
\[ \text{Q2' : USRS} \land \text{users = USRS} \land \]
\[ \text{who\_has = USRS \land \text{users}\#} \}
\[ \text{if corec(usr,b) \in checks} \rightarrow \]
\[ \text{who\_has = who\_has +usr ;}
\[ \] \text{end corec(usr,b) \#checks} \rightarrow \]
\[ \text{skip ;}
\[ \text{fi}
\[ \{ R2 : Q2' \land \]
\[ \text{(corec(usr,b) \in checks} \land \]
\[ \text{who\_has = WHO\_HAS +usr} \land \]
\[ \text{corec(usr,b) \#checks} \land \]
\[ \text{who\_has = WHO\_HAS) \}
\[ \text{od}
\[ \{ R : P \land \text{users} = \{ \}
\[ \{ R' : \text{who\_has} = \}
\[ \{ \text{u:users:corec}(u,b) \in \text{checks} \} \}

Figure 12. Annotated Design of Who\_has

Lemma 1: \{ Q1 \} S1 \{ Q2 \}
\[ \{ Q1 \} \text{choose(users,usr) \{} Q1 \land \text{users = usrs} \}
\[ \{ Q1 \land \text{users = usrs}\} \text{usr = users-usr \{} Q2 \}
\[ \text{Q1 \land \text{users = usrs}}
\[ \Rightarrow \text{Q2ussrs} \}
\[ \Rightarrow \text{who\_has = WHO\_HAS } \land \]
\[ \text{USRS = users} \land \]
\[ \text{WHO\_HAS =}
\[ \{ \text{u:users-USRS:corec}(u,b) \in \text{checks} \} \land \]
\[ \text{users = USRS-usr} \land \]
\[ \text{usr = USRS} \]
Lemma 2: \{Q2\} IF \{R2\}
1) Q2 => BB
   Q2 => corec(usrs,b) \epsilon checks | corec(usrs,b) \epsilon checks
2.1) \{Q2 \land corec(usrs,b) \epsilon checks\}
    who_has := who_has + usr;
    \{R2\}
    Q2 \land corec(usrs,b) \epsilon checks
    => R2 who_has + usr
    => Q2' \land
    (corec(usrs,b) \epsilon checks \land
    who_has + usr =
    WHO_HAS + usr \lor
    corec(usrs,b) \epsilon checks \land
    who_has + usr = WHO_HAS)
2.2) \{Q2 \land corec(usrs,b) \epsilon checks\} skip \{R2\}
    Q2 \land corec(usrs,b) \epsilon checks => R2
    => Q2' \land
    (corec(usrs,b) \epsilon checks => R2
    who_has =
    WHO_HAS + usr \lor
    corec(usrs,b) \epsilon checks \land
    who_has = WHO_HAS)

therefore, \{Q2\} DO \{R\}

Lemma 3: R2 => P
R2 => Q2'
   => USERS users \land
   T1:(usr=USRS-user \land usr \epsilon USERS)
   => P1:usr \epsilon users
R2 => Q2'
   => T2:WHO_HAS =
   \{user \epsilon USERS:corec(u,b) \epsilon checks\}
R2 => T3:(corec(usrs,b) \epsilon checks \land
   who_has = WHO_HAS +usr) \lor
   T4:(corec(usrs,b) \epsilon checks \land
   who_has = WHO_HAS)
T1 \land T2 \land T3 =>
   P2:who_has = \{user \epsilon users-users:corec(u,b) \epsilon checks\}
T1 \land T2 \land T4 => P2
P1 \land P2 => P
R2 => P

therefore, R2 => P

Using these three lemmas, we can prove the correctness of the loop, following the rule in Figure 11.
1) Q => P
   who_has = \{\} \land users = users => P who has, users
   Q => users \subseteq users \land
   \{\} = \{user \epsilon users-users:corec(u,b) \epsilon checks\}
2) \{P \land B\} \{S\} \{P\}
   \{P \land B\} \{Q1\} S1 \{Q2\} IF \{R2\} \{P\}
   P \land users \notin \{\} => Q1
   \{Q1\} S1 \{Q2\}
   \{Q2\} IF \{R2\}
   R2 => P

3) P \land \neg BB => R
   P \land \neg (users \notin \{\}) => P \land users = \{\}
4) P => \{t \geq 0\}
   P => |users| \geq 0
5) \{P \land B\} t1 := t; S1; IF \{t < t1\}
   \{P \land B\} t1 := \{t \geq t\} S1 \{t < t1\} IF \{t < t1\}
   \{P \land B\} t1 := \{t \geq t\}
   \{t = t\} choose(using,usr) \{t \land \epsilon users \epsilon users\}
   \{t \land \epsilon users \epsilon users\} users := users -usr \{t < t1\}
   \epsilon users => |users -usr| < |users|
   \{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 Kummer'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


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