REUSABLE PROBLEM DOMAIN THEORIES

Donald I. Good

Technical Report 31 September 1982

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Institute for Computing Science
The University of Texas at Austin
Austin, Texas 78712

Abstract

One of the main reasons why constructing deductive proofs that programs satisfy their specifications can be very expensive in practice is the absence of reusable problem domain theories. These theories contain functions that define relevant concepts in the application area of the program, and they contain properties that are deduced from these definitions. Presently, the cost of proving programs is highly inflated by the fact that we usually have to build a new problem domain theory for each new application. If we can develop reusable problem domain theories, the cost of specifying and proving programs in actual practice can be greatly reduced. The development of these theories also would have significant benefits for other aspects of computing science. This paper discusses the composition of problem domain theories and their relation to program specification and proof.

Acknowledgements

During its eight year existence, well over 50 people have contributed to the Gypsy project, and I will not attempt to name them all. Gypsy, however, has been a true team effort, and the current team consists of Mike Smith, Ann Siebert, Bill Young, Rich Cohen, John McHugh, Larry Akers, Larry Smith, Bill Bevier, Judy Merriam, Ben DiVito, Sandy Olmstead, and Ayshegul Musallam.

The Gypsy project has been sponsored in part by the U. S. Naval Electronic Systems Command through contract N00039-81-C-0074, by Digicomp Research Corporation, and by Digital Equipment Corporation.

1. Introduction

If we can reduce the current level of effort required to specify and prove programs by a factor of 100, then I believe that we should be close to the point where rigorous specification and deductive proof will be economical enough so that they can become a normal part of the program development process. This paper examines the development of reusable problem domain theories as a way of doing a substantial part of that reduction. The development of these theories would require a very considerable effort, but once developed, it appears that they can produce significant reductions in the effort required to specify and prove programs. With a high enough degree of reusability, they could account for a large portion of the factor of 100 that seems needed to make these techniques practical.

Constructing a deductive proof that a program satisfies a specification requires three major steps.

1. Precise statements of both the program and its specification are formulated.
2. These statements are transformed into statements in ordinary logic, called verification conditions, which are sufficient to imply that the program meets its specification.
3. Deductive proofs of the verification conditions are constructed.

The general strategy here is problem transformation. We transform the problem of proving programs into a problem of proving theorems. Much of the program proving research of the past 15 years has concentrated on this transformation step. The state of the art has now progressed to the point where the transformation from programs and specifications into verification conditions can be made automatically for a very useful collection of programming language constructs.

Studying this transformation process can do much to increase our understanding of programming languages and the programming process. But, if we want to use proofs about programs as an effective way of actually constructing reliable programs in practice, we must perform all three steps. When we do that, we usually find that most of our effort goes not into the transformation step, but into formulating the program and its specifications in such a way that we can construct manageable proofs.

We should point out that the transformation strategy is the one that normally is used in proving programs written in languages such as Pascal [Jensen 74], Gypsy [Good 78] and Euclid [Lampson 76]. This transformation step is not necessary, or at least very minimal, in programming languages that closely resemble ordinary logic such as pure Lisp [McCarthy 62]. The computational logic of [Boyer 79], for example, has been very successful in formulating both programs and specifications in Lisp and mechanically proving their consistency.

In any case, though, once one has the logical formulas that need to be proved, the key to manageable proofs is abstraction. We must use abstraction with a sufficiently fine level of granularity in programs, in specifications and in proofs so that each of these can be kept small enough to think about rigorously and soundly. This is our primary mechanism for mastering the complexity of a large program.

As we construct these abstractions, we find what that we really are doing is constructing what we shall call a "problem domain theory." The axioms of this theory are a set of function definitions that are assumed to characterize the relevant concepts of the application area of the program. These functions are used to compose the program specifications. When we transform the program and its specifications into verification conditions, these specification functions also appear in the verification conditions. In order to keep our proofs of the verification conditions at an appropriate level of abstraction, we normally also need to use various properties of the specification functions. Then, to complete the proof, we also need to prove these properties. The proofs of these properties also must be kept at an appropriate level of abstraction, typically by appealing to slightly less powerful properties. We continue in this way to decompose a large proof into manageable pieces. The specification functions, whose definitions we assume, and the properties that are proved to follow from them form a theory about the application domain of the problem that is to be solved by the program. This is the problem domain theory.

Given the ability to construct verification conditions automatically, the proof that a program satisfies its specification can become almost arbitrarily straight forward if we have the appropriate problem domain theory. In the best case, every verification condition is exactly some property in a previously proved problem domain

theory, and therefore, the proof of the program is trivial. In practice, however, we seldom have such a theory. Rather than having a previously proved theory that allows us to deduce the verification conditions in a few steps, we find ourselves having to build such a theory. We continually find ourselves reformulating problem domain functions and their derived properties (and usually the program too) so that we can get a proof that can be segmented into manageable pieces. It is this theory building effort that frequently dominates the effort required to prove a program in actual practice. This is often a difficult and intellectually challenging task.

Presently, proving that a program satisfies its specification typically requires building a new problem domain theory for each new application, and therefore, proving programs in practice can be quite costly. However, doing something once requires less effort than doing it twice, and it can requires much less effort than doing it many times. If we can develop problem domain theories that can be reused in a variety of applications within a similar problem area, then the costs of both specification and proof can be greatly reduced. It may even be possible to reduce these costs to the point where specifying and proving a program can become a conventional part of the process of program development.

In addition to reducing verification costs, the development of reusable problem domain theories can have other important benefits. The development of these theories requires a deep study and understanding of the particular problem domain and how it relates to computer programs. The development of these theories would greatly increase our understanding of the relationship between our programs and the problems they are supposed to solve. Just this increased understanding of the problems we are trying to solve would lead to a significant improvement of the quality of programs we build. All too often, programming bugs are a result of an inaccurate or an incomplete understanding of the problem. Once these theories are developed, they also can be a mechanism for transferring knowledge about the problem domain from one program builder to another. Communication of programs among people is one of the important roles of programming languages, and the development of rigorously formulated problem domain theories could play a similar role in transferring knowledge about an application area. The existence of reusable problem domain theories also would make it possible for more of the program development process to be done objectively rather than subjectively. Therefore, it appears that the development of these theories might well do much to accelerate the maturation of computing as a science.

The remainder of this paper begins by giving a simple example of a problem domain theory. The problem domain of this example is editing papers for a technical journal. This theory is not intended to be a reusable theory; it is intended just to be an example of a simple theory about an application area. Next, we will show how this simple theory relates to the synchronization of abstraction in specification, implementation and proof of the editing program. With these basic principles established, we will then give an example of a reusable theory for the problem domain of searching sequences for a particular object. We will show how this theory can be reused in the specification and proof of several different kinds of search implementations. This theory is not intended to be complete in any sense; its purpose is to illustrate various aspects of reusability.

2. A Simple Problem Domain Theory

We will now illustrate the main ideas of a problem domain theory with a simple example. Suppose that we want to construct a program that will assist the editor of a technical journal in refereeing papers for publication. In order to specify and prove this program, we will need a small theory for the problem domain of editing papers. First, we will formulate such a theory, and then we will show how that theory is used to specify and prove a simple refereeing program.

This theory is not intended to be a reusable theory. It is much too simple minded. A reusable theory is given in the next section. The points that we want to illustrate with this example are

- The specification and proof of the refereeing program does require formulating precise definitions and properties about what it means to referee papers. These are not properties about specific parts of the program implementation. They are properties about the process of editing papers.
- These definitions and properties can be isolated in a simple theory.
- Given the appropriate theory, the specification and proof of the program is very straight forward.

The following presentation gives first the theory and then the specification and proof of the program. This is not how one would normally develop a proved program in a new problem domain. In such a case, the development of the theory, the program and its specifications and proof normally would be highly interleaved.

2.1 Editing Theory

Let us begin the formation of the theory by deciding on the data objects that are relevant to this problem domain. We choose three basic types of objects: files, papers, and words. A file is a sequence of papers, and a paper is a sequence of words. We can state this more precisely as follows:

```
type a_file = sequence (max_papers) of a_paper;
const max_papers:integer = pending;

type a_paper = sequence (max_words) of a_word;
const max_words:integer  = pending;
type a_word_index = integer[1..max_words];

type a_word = pending;
```

A_FILE is composed of a sequence of at most MAX_PAPERS objects each of which is A_PAPER. For generality, the value of MAX_PAPERS is left pending. Similarly, we define A_PAPER to be a bounded sequence of A_WORD. At this point, a more precise definition of A_WORD is not needed, and it can be left pending for generality. A_WORD_INDEX is an integer in the interval 1,...,MAX_WORDS.

Given these basic objects, we now can begin to define precisely what it means to referee papers. Every paper should be refereed with respect to some fixed criteria, and therefore, for now, we extend our theory as follows:

```
type some_criteria = pending;

function acceptable(p:a_paper; c:some_criteria) : boolean =
pending;
```

ACCEPTABLE is some function that determines that acceptability of a paper with respect to the refereeing criteria.

Given these definitions, we can define what it means for a file R to be refereed with respect to a file of submitted papers S and a refereeing criteria C.

```
function papers_refereed(r,s:a_file; c:some_criteria):boolean =
begin
exit [assume result iff all p:a_paper,
      p in r iff p in s & acceptable(p,c)];
end;
```

The RESULT produced by PAPERS_REFEREED is a boolean value, TRUE or FALSE, and the RESULT is TRUE if and only if file R contains exactly the acceptable papers in S.

From just this information, we can prove several properties of PAPERS_REFEREED.

```
lemma empty_files(r,s:a_file; c:some_criteria) =
  papers_refereed(null(a_file),null(a_file),c);

lemma accept_paper(r,s:a_file; p:a_paper; c:some_criteria) =
  papers_refereed(r,s,c) & acceptable(p,c)
  -> papers_refereed(r <: p, s <: p, c);

lemma reject_paper(r,s:a_file; p:a_paper; c:some_criteria) =
  papers_refereed(r,s,c) & not acceptable(p,c)
```

```
    -> papers_refereed(r, s <: p, c);
```

In the preceding, NULL(A_FILE) denotes the empty file, and R <: P denotes the sequence formed by adjoining the single element P onto the end of sequence R.

Lemma EMPTY_FILES says that the empty files are refereed with respect to any criteria C. Lemma ACCEPT_PAPER states how to extend the files R and S with an acceptable paper, and REJECT_PAPER states how to extend them with an unacceptable paper. Each of these lemmas can be proved without further refinement of ACCEPTABLE, and therefore, their proofs are independent of the definition of acceptability.

These definitions and their properties define a fairly general theory of refereeing papers. Now, let us expand the theory by making it considerably more specific. We will do this by refining the definitions of SOME_CRITERIA and ACCEPTABLE. There are many ways we could refine these definitions, but let us choose a particularly simple one. Let us suppose that the refereeing criteria is just a list of keywords, and that a paper is acceptable if and only if it contains none of the keywords. More precisely,

```
type some_criteria = a_keyword_seq;
type a_keyword_seq = sequence (max_keys) of a_word;
const max_keys:integer  = pending;
type a_keyword_index = integer[1..max_keys];

function acceptable(p:a_paper; c:some_criteria) : boolean =
begin
exit [assume result iff no_keywords(p,c)];
end;

function no_keywords(p:a_paper; c:some_criteria) : boolean =
begin
exit [assume result iff all w:a_word,
       w in p -> not in_keywords(w,c)];
end;

function in_keywords(w:a_word; c:some_criteria) : boolean =
begin
  exit [assume result iff isin(w,c,1,size(c))];
end;
```

This defines ACCEPTABLE in terms of NO_KEYWORDS which is defined in terms of IN_KEYWORDS which is defined in terms of ISIN. ISIN(X,S,I,J) is a function from a theory of sequences that is TRUE if and only if element X in one of the sequence elements S[I],...,S[J].

From these additional definitions, we can prove the following additional properties:

```
lemma empty_paper(c:some_criteria) =
   no_keywords(null(a_paper),c);

lemma accept_word(p:a_paper; w:a_word; c:some_criteria)=
  no_keywords(p,c)
  & not in_keywords(w,c)
  -> no_keywords(p <: w,c);

lemma reject_word(w:a_word; p:a_paper; c:some_criteria) =
  w in p & in_keywords(w,c) -> not no_keywords(p,c);
```

Lemma EMPTY_PAPER says that the empty paper has no key words. Lemma ACCEPT_WORD says how to extend the examination of a paper to include a new word that is not a key word. Lemma REJECT_WORD says that if we find any key word, the whole paper should be rejected.

We now have in hand a set of function definitions and properties that comprise a small theory of editing.

This theory is sufficient to specify and prove some simple editing programs.

2.2 Editing Program

Now let us construct a program called REFEREE that computes the refereed file from a file of submitted papers. Given the preceding theory, both the specification and proof of the program are very straight forward.

```
procedure referee(var r:a_file; s:a_file; c:some_criteria) =
begin
exit papers_refereed(r,s,c);
  var i:integer;
  i := 0;
  r := null(a_file);
  loop
    assert papers_refereed(r,s[1..i],c);
    if i = size(s) then leave
    end;
    i := i + 1;
    if do_acceptable(s[i],c) then r := r <: s[i];
           end;
  end;
end;

function do_acceptable(p:a_paper; c:some_criteria) : boolean =
begin
  exit result iff acceptable(p,c);
  pending;
end;
```

REFEREE is specified in terms of the function PAPERS_REFEREED which was defined in the editing theory. This function is used in stating both the EXIT specification of REFEREE and in stating its loop invariant. In the loop invariant, S[1..I] denotes S[1],...,S[I].

REFEREE is implemented as a simple loop that invokes the function DO_ACCEPTABLE. DO_ACCEPTABLE is specified to be equivalent to the specification function ACCEPTABLE defined in the editing theory. (There is nothing wrong with implementing the problem domain function ACCEPTABLE and using it directly to implement REFEREE. We have used two functions with different names so that we can clearly identify those functions that are implementations and those that define problem domain concepts. ACCEPTABLE defines the problem domain concept, and DO_ACCEPTABLE implements it.)

Now let us examine the proof of REFEREE. There are four verification conditions that need to be proved. The first of these covers the case when REFEREE enters its loop the first time,

```
papers_refereed (null (a_file), null (a_file), c).
```

This is exactly the EMPTY_FILES lemma from the editing theory.

The second vc covers the looping case when DO_ACCEPTABLE is TRUE,

```
papers_refereed (r, s[1..i], c)
& i ne size (s)
& do_acceptable (s[i + 1], c)
-> papers_refereed (r <: s[i + 1], s[1..i + 1], c).
```

The first step in this proof is to apply the specification of DO_ACCEPTABLE, and we get

```
papers_refereed (r, s[1..i], c)
& i ne size (s)
& acceptable (s[i + 1], c)
```

```
-> papers_refereed (r <: s[i + 1], s[1..i + 1], c).
```

Notice that this simple step maps the vc from the domain of programs and implementations into the problem domain. We now use the ACCEPT_PAPER lemma from the editing theory and the proof is complete.

The third vc covers the looping case when DO_ACCEPTABLE is FALSE,

```
papers_refereed (r, s[1..i], c)
& i ne size (s)
& not do_acceptable (s[i + 1], c)
-> papers_refereed (r, s[1..i + 1], c)
```

We apply the specification of DO_ACCEPTABLE and get

```
papers_refereed (r, s[1..i], c)
& i ne size (s)
& not acceptable (s[i + 1], c)
-> papers_refereed (r, s[1..i + 1], c).
```

We now use the REJECT_PAPER lemma from the editing theory and the proof is complete.

The fourth vc covers the case when the loop terminates,

```
papers_refereed(r, s[1..i], c)
& i = size(s)
-> papers_refereed(r,s,c).
```

It is interesting to note that this vc can be proved without using any information from the editing theory. We simple note that S[1..SIZE(S)] = S for any sequence S. This vc, and therefore this part of the program, deals not with any problem domain issue, but just with some of the implementation details.

In a similar manner, we also use the editing theory to specify and prove DO_ACCEPTABLE.

```
function do_acceptable(p:a_paper; c:some_criteria) : boolean =
begin
  exit result iff acceptable(p,c);
  var i:integer;
  i  := 0;
  loop
    assert no_keywords(p[1..i],c)
         & i in [1..size(p)];
    if i = size(p) then
      result := true;  leave;
    end;
    i := i + 1;
    if do_in_keywords(p[i],c) then
      result := false; leave;
    end;
  end;
end;

function do_in_keywords(w:a_word; keys:a_keyword_seq):boolean =
begin
exit [result iff in_keywords(w,keys)];
  result := do_isin(w,keys,1,size(keys));
end;
```

DO_ACCEPTABLE is specified in terms of the specification functions ACCEPTABLE and NO_KEYWORDS. The loop invariant specifies that no key words occur in the first I words of the paper. DO_ACCEPTABLE is implemented in terms of DO_IN_KEYWORDS which is specified in terms of IN_KEYWORDS and implemented in terms of DO_ISIN. DO_ISIN is a function which implements ISIN,

```
function do_isin (x : an_object;  s : a_sequence;
                  p, q : an_index)     : boolean =
begin
  exit result iff isin(x,s,p,q);
  pending;
end.
```

The first vc for DO_ACCEPTABLE covers the loop entry case,

```
no_keywords (null (a_paper), c).
```

This is exactly the EMPTY_PAPER lemma from the editing theory.

The second vc covers the looping case when DO_IN_KEYWORDS is TRUE and the loop terminates,

```
no_keywords (p[1..i], c)
& i in [0..size (p)]
& i ne size (p)
& do_in_keywords (p[i + 1], c)
-> not acceptable (p, c).
```

The proof consists of applying the definitions of DO_IN_KEYWORDS and ACCEPTABLE and then using the REJECT_WORD lemma.

The third vc covers the looping case when DO_IN_KEYWORDS is FALSE,

```
no_keywords (p[1..i], c)
& i in [0..size (p)]
& i ne size (p)
& not do_in_keywords (p[i + 1], c)
-> no_keywords (p[1..i + 1], c)
   & i + 1 in [0..size (p)].
```

The proof consists of applying the definition of DO_IN_KEYWORDS and using the ACCEPT_WORD lemma.

The fourth vc covers the case when the loop terminates because I=SIZE(P),

```
no_keywords (p[1..i], c)
& i in [0..size (p)]
& i = size (p)
-> acceptable (p, c).
```

This is proved by applying the definition of ACCEPTABLE.

This leaves only the proof of DO_IN_KEYWORDS. It has only one vc which is

```
do_isin (w, keys, 1, size (keys)) iff in_keywords (w, keys).
```

It is proved by applying the definitions of DO_ISIN and IN_KEYWORDS.

3. Abstraction

It is important to notice how abstraction has been used in the preceding example. We have used abstract programs, implementations and proofs whose levels of abstraction are closely synchronized. The relations among the various abstractions are illustrated in Figure 1. The higher levels of abstraction are higher in the figure.

The most abstract program is REFEREE. It is specified in terms of an equally abstract specification function, PAPERS_REFEREED, and proved in terms of the specification of DO_ACCEPTABLE and the

Figure 1: Abstraction Relations

Program	Specification	Proof
referee	papers_refereed	empty_files
		accept_paper
		reject_paper
do_acceptable	acceptable	empty_paper
		accept_word
		reject_word
	no_keywords	
do_in_keywords	in_keywords	
do_isin	isin	

lemmas, EMPTY_FILES, ACCEPT_PAPER and REJECT_PAPER. These lemmas are stated at a similar level of abstraction. Similarly, the other programs are specified and proved in terms of abstractions appropriate to their abstraction levels. Because of the synchronized levels of abstraction, it was never necessary for any single proof to span significantly different levels of abstraction.

It is instructive to observe what happens if we try to specify and prove REFEREE without these abstractions. For example, if we retain exactly the same program structure that we had above, but do not use abstraction in the specifications, we get the programs shown below. There are several things to notice. First, the specifications begin to dominate the program text. Second, the specifications are much more difficult to understand because they are larger and visually more complex. For example, the similarity between the EXIT and the ASSERT specification of XREFEREE is more difficult to see just because of the increased textual complexity. Third, the specifications also are more difficult to understand because the problem domain concepts are not explicit. We have to study the specifications in close detail and recognize that they are characterizing the problem domain concepts of PAPERS_REFEREED, ACCEPTABLE and the others. The specifications are easier to comprehend if these concepts are explicit as in our previous version.

```
procedure xreferee(var r:a_file; s:a_file; c:some_criteria) =
begin
exit all p:a_paper,
       p in r iff p in s
                & [all w:a_word, w in p
                    -> not isin(w,c,1,size(c))];
  var i:integer;
  i := 0;
  r := null(a_file);
  loop
    assert all p:a_paper,
              p in r iff p in s[1..i]
                       & [all w:a_word, w in p
                           -> not isin(w,c,1,size(c))];
    if i = size(s) then leave
    end;
    i := i + 1;
    if xdo_acceptable(s[i],c) then r := r <: s[i];
    end;
  end;
end;

function xdo_acceptable(p:a_paper; c:some_criteria) : boolean =
```

```
begin
  exit result iff [all w:a_word, w in p
                   -> not isin(w,c,1,size(c))];
  var i:a_word_index;
  i  := 0;
  loop
    assert all w:a_word, w in p[1..i]
             -> not isin(w,c,1,size(c));
    if i = size(p) then
      result := true;  leave;
    end;
    i := i + 1;
    if xdo_in_keywords(p[i],c) then
      result := false; leave;
    end;
  end;
end;

function xdo_in_keywords(w: a_word; keys : a_keyword_seq)
                          : boolean =
begin
exit [result iff isin(w,keys,1,size(keys))];
  result := do_isin(w,keys,1,size(keys));
end;
```

The other important thing to notice here is what happens to the verification conditions and proofs. For example, the verification condition for XREFEREE for the looping case when XDO_ACCEPTABLE is TRUE becomes

```
all p : a_paper,
  [(all w : a_word, w in p -> not isin (w, c, 1, size (c)))
  & p in s[1..i]]
  iff p in r
& i ne size (s)
& xdo_acceptable (s[i + 1], c)
-> all p : a_paper,
    [(all w : a_word, w in p -> not isin (w, c, 1, size (c)))
    & p in s[1..i + 1]]
    iff s[i + 1] = p or p in r
```

Just the textual appearance of this verification condition is intimidating, and, to make matters worse, we have to prove it without the explicit aid of the problem domain lemmas. What has happened here is that the program is at one level of abstraction, and the specifications are at a lower level. The proof must show the consistency between the two, and therefore, it of necessity must span the gap between the levels. The tractability of the proof obviously depend on the size of the abstraction gap.

4. A Reusable Theory

The foundation of a problem domain theory is a set of function definitions of basic problem domain concepts. Then, from these definitions, we deduce lemmas that are increasingly powerful and applicable to the specific program implementations we have in mind. Often, the key step in implementing the program is the successful exploitation of some problem domain property.

In general, both the functions and their derived lemmas provide opportunities for reusability. A problem domain function can be reused in each of several ways. It may be used in defining other, more complex, problem domain functions. It may be used in stating the problem domain lemmas. It may be used in stating the specifications of the program. A problem domain lemma also can be reused in each of several ways. It may be

used in the proof of additional problem domain lemmas. It may be used in the proof of verification conditions.

The economy of all of this is very simple. Each time we reuse a problem domain function, we save the effort of redefining it, and each time we reuse a previously proved lemma, we save the effort of reproving it. In order to gain maximal benefit from a reusable theory, we need parts of it to be reusable not only in one particular application but also in other applications in the same general problem domain.

The following examples illustrate various aspects of this kind of reusability in a very simple problem domain. We will continue the development of the refereeing program by considering the implemention of the search function DO_ISIN. We will give a simple theory for searching sequences which defines the problem domain function ISIN and which is adequate for the proof of a linear search implementation of DO_ISIN. The proof of the linear search reuses several parts of the theory. The next example is a different implementation of a linear search, and it also can be proved from the same simple theory. The final example is an ordered search. It also reuses some of the simple searching theory, as well as requiring some additional definitions and properties about ordered sequences.

The theory that supports these examples is not intended to be powerful enough to prove all programs that search sequences. Its purpose is to illustrate various kinds of reusability.

4.1 A Linear Search

The basic objects of the theory are bounded sequences of objects of some pending type.

```
type a_sequence = sequence (max_size) of an_object;
const max_size : integer = pending;
type an_index = integer[1..max_size];


type an_object = pending;
```

The theory focuses on finding an object within a specific segment of a sequence. The foundation of the theory is the ISIN function which defines the presence of an object X in the sequence segment S[M],...S[N].

```
function isin (x : an_object; s : a_sequence; m, n : integer)
                 : boolean =
begin
  exit [assume result iff some k:integer,
        k in [1..size(s)] & k in [m..n] & x=s[k]];
end;
```

It also is useful to have a separate function to define absence of X in the segment.

```
function notin (x : an_object; s : a_sequence; m, n : integer)
                  : boolean =
begin
  exit [assume result iff not isin(x,s,m,n)];
end;
```

Searching a sequence for X can be viewed as eliminating segments of the sequence from further consideration, and NOTIN is used to make assertions about these segments. NOTIN also helps to simplify some of the subsequent lemmas.

We now can use these problem domain functions to specify a simple linear search function, DO_LINEAR_ISIN.

```
function do_linear_isin (x : an_object; s : a_sequence;
                         p, q : an_index) : boolean =
begin
  exit result iff isin (x, s, p, q);
```

```
    var m : a_left_index;
    var top : an_index;
    m := max (p, 1);
    top := min (size (s), q);
    loop
      assert if m > top then notin (x, s, p, q)
             else notin (x, s, p, m - 1) fi
           & max (p, 1) le m
           & top = min (size (s), q);
      if m > top then result := false; leave;
      elif x = s[m] then result := true; leave;
      else m := m + 1;
      end;
    end;
  end;
```

This program can be proved from the problem domain lemmas given below.

The INTERSECTION lemma states how MAX and MIN can be used to compute the intersection of the segments [1..SIZE(S)] and [M..N] which are used in the definition of ISIN.

```
  lemma intersection(x : an_object; s : a_sequence;
                     m, k,  n : integer)=
    [k in [1..size(s)] & k in [m..n]]
    iff k in [max(m,1)..min(size(s),n)];
```

The definition of ISIN requires that X be some sequence element in the intersection of these two intervals.

The NOTIN_EMPTY_INTERSECTION lemma says the X can not be in an empty intersection of [1..SIZE(S)] and [M..N].

```
  lemma notin_empty_intersection (x : an_object;
                                  s : a_sequence;
                                m, n : integer) =
    min (size (s), n) < max (m, 1)
    -> notin (x, s, m, n);
```

The LEFT_BOUND and RIGHT_BOUND lemmas say that we can compute NOTIN for the interval [M..N] by considering just the interval [MAX(M,1)..MIN(SIZE(S),N)].

```
  lemma left_bound (x : an_object; s : a_sequence;
                m, n : integer) =
    notin (x, s, max (m, 1), n)
    iff notin (x, s, m, n);

  lemma right_bound (x : an_object; s : a_sequence;
                 m, n : integer) =
    notin (x, s, m, min (size (s), n))
    iff notin (x, s, m, n)
```

These can be proved from the definitions of NOTIN and ISIN and from the INTERSECTION lemma. Thus, INTERSECTION is one example of a lemma that is reusable within the theory itself.

The FOUND_IT lemma says that if K is in the search interval and K is a legitimate sequence element, then S[K] is in the search interval.

```
  lemma found_it (s : a_sequence;  p, k, q : integer) =
    k in [p..q] & k in [1..size(s)] -> isin (s[k], s, p, q);
```

This lemma is used to prove the case when DO_LINEAR_ISIN finds X.

The PUSH_NOTIN_ONE_UP lemma is one that is very specific for a linear search that is scanning the sequence upward in the order 1, 2, ...

```
lemma push_notin_one_up (x : an_object; s : a_sequence;
                         p, k : integer) =
  k in [1..size(s)] & notin (x, s, p, k - 1) & s[k] ne x
  -> notin (x, s, p, k);
```

This is the problem domain property that is basis for this particular linear search implementation. Interestingly enough, though, this lemma is used twice in the proof of DO_LINEAR_ISIN. It is used first in the proof of the loop iteration. It also is used in proving the EXIT specification from the ASSERT. This proof requires two cases, the first for M > TOP and the second for M le TOP. PUSH_NOTIN_ONE_UP is used in the second case to deduce NOTIN(X,S,P,M) from NOTIN(X,S,P,M-1). In this case, we also have that M=TOP and using the RIGHT_BOUND lemma completes the proof.

4.2 Another Linear Search

Now, let us consider a different implementation of a linear search, DO_LINEAR_ISIN2.

```
function do_linear_isin2 (x : an_object;
                          s : a_sequence;
                       p, q : an_index) : boolean =
begin
  exit result iff isin (x, s, p, q);
  var m : a_left_index;
  var top : an_index;
  m := max (p, 1);
  top := min (size (s), q);
  result := false;
  loop
    if m > top or result then leave
    end;
    assert notin (x, s, p, m - 1)
         & m in [max (p, 1)..top]
         & top = min (size (s), q);
    result := x = s[m];
    m := m + 1;
  end;
end;
```

This implementation can be proved from the same theory as the previous implementation.

It also is not hard to imagine other programs that can be specified and proved from this same theory. For example, if we modify these functions slightly so that they find the index of the element being searched for rather than just determining the existence of the element, these new routines also could be proved from the same theory.

It also is easy to imagine linear search programs for which this theory is not adequate. For example, if the loops were implemented to count down instead of up, the PUSH_NOTIN_ONE_UP lemma would not be appropriate. What we would need instead would be a PUSH_NOTIN_ONE_DOWN,

```
lemma push_notin_one_down (x : an_object; s : a_sequence;
                           k, q : integer) =
  k in [1..size(s)] & s[k] ne x & notin (x, s, k+1, q)
  -> notin (x, s, k, q);
```

The other lemmas, such as FOUND_IT, however, could still be used.

4.3 An Ordered Search

Now let us see how much of our theory that supported the specifications and proofs of the linear search also will support the specification and proof of an ordered search. Let us consider the following DO_ORDERED_SEARCH function which is a simple generalization of a binary search.

```
function do_ordered_isin (x : an_object; s : a_sequence;
                          p, q : an_index) : boolean =
begin
  entry inorder (s, p, q);
  exit result iff isin (x, s, p, q);
  var m, k, n : an_index;
  m := max (p, 1) - 1;
  n := min (size (s), q) + 1;
  loop
    assert notin (x, s, p, m)
         & notin (x, s, n, q)
         & max (p, 1) - 1 le m
         & n le min (size (s), q) + 1
         & inorder (s, p, q);
    if m ge n - 1 then result := false; leave
    else set_probe (k, m + 1, n - 1)
    end;
    if s[k] = x then result := true; leave;
    elif less_than (x, s[k]) then n := k
    else m := k;
    end;
  end;
end;

procedure set_probe (var k : an_index; p, q : an_index) =
begin
entry p le q;
exit k in [p..q];
  pending;
end;
```

To state the specifications for DO_ORDERED_SEARCH, the first thing we need to do is define what it means for the sequence to be ordered. We do this by adding the function, INORDER, to our sequence theory.

```
function inorder (s : a_sequence; m ,n : integer) : boolean =
begin
  exit [assume result iff all i, j : integer,
            i in [max(m,1)..min(size(s),n)]
            & j in [max(m,1)..min(size(s),n)]
            & i le j
            -> less_or_eq (s[i], s[j])];
end;
```

INORDER is defined in terms of LESS_OR_EQ which is a function from a theory of ordered objects. We will not include the details of this theory, but the essential property is that there be a total ordering defined on the objects.

The proof of DO_ORDERED_ISIN also requires the use of two additional lemmas which make use of the ordering of the sequence.

```
lemma push_notin_up (x : an_object;  s : a_sequence;
                p, k, q : integer) =
  inorder (s, p, q) & less_than (s[k], x)
```

```
  & k in [p..q]
  & k in [1..size(s)]
  -> notin (x, s, p, k);

lemma push_notin_down (x : an_object;  s : a_sequence;
                p, k, q : integer) =
  inorder (s, p, q) & less_than (x, s[k])
  & k in [p..q]
  & k in [1..size(s)]
  -> notin (x, s, k, q);
```

The lemmas are used in proving the respective cases of the loop iteration.

We also need a new lemma to prove the case when the loop terminates.

```
lemma merge_notin (x : an_object;    s : a_sequence;
          m, p, q, n : integer) =
  notin (x, s, m, p) & notin (x, s, q, n)
  & p + 1 ge q
  -> notin (x, s, m, n);
```

This lemma is a fairly general one that says that if an object is not in the segment from M to P, and it is not in the segment from Q to N and these two segments overlap, then it is not in the segment M to N.

Here it is interesting to note that the PUSH_NOTIN_ONE_UP and PUSH_NOTIN_ONE_DOWN lemmas, which were used in proving the linear search programs, actually are special cases of MERGE_NOTIN. Both of these are easily proved from MERGE_NOTIN and the following

```
lemma start_notin (x : an_object; s : a_sequence;
                   k : integer) =
  k in [1..size(s)] -> [notin(x,s,k,k) iff s[k] ne x];
```

MERGE_NOTIN, therefore, is an example of a lemma that can support the proof of two substantially different kinds of search implementations.

The best example of a reusable lemma in the proofs of both implementations, however, is the FOUND_IT lemma. It is used in both implementations when the object being searched for is found. One actually would expect that this lemma could be used in that case for any search implementation. In addition to FOUND_IT, the lemmas RIGHT_BOUND, LEFT_BOUND and NOTIN_EMPTY_INTERSECTION also are used in the proofs of both the ordered and linear searches. It also is easy to envision these lemmas being used to support proofs of other implementations.

5. Conclusion

In the preceding examples we have illustrated the basic idea of what a problem domain theory is and illustrated various kinds of reusability. Whether or not we can develop problem domain theories with a high enough degree of reusability to reduce program specification and proof costs significantly remains an open question. The main claim of this paper is that reusable problem domain theories do exist and that they are worthy research subjects.

The notion of reusable theories certainly is not limited to simple examples. In fact, the larger the verification application, the more necessary it is to have an effective, reusable theory. It actually was our Gypsy specification and verification work on much larger examples [Good 82] that brought many of these ideas into focus. In applications such as this, in which we are specifying and proving in excess of 4000 lines of implementation, the synchronized abstraction techniques become absolutely essential -- much more so than in a small example. Once these abstractions are developed, the problem domain theory just appears. These larger Gypsy applications contain many examples of reusability. The related work of [DiVito 82] also contains some significant examples of reusability in the area of specification and verification of communications protocols.

In general, some semblance of reusable theories seems to be developing in association with several of the mechanical verification projects. For example, the Boyer-Moore [Boyer 79] theory contains a rather substantial number of theorems that have been mechanically proved to follow from their axioms, and these have been used in proving a variety of programs. A library of proved properties about abstract data types also has developed in conjunction the Affirm system [Gerhart 80]. The reason for this association is that mechanical support makes it possible to do enough verifications and proofs so that one can find useful theories. In fact, these mechanical systems may be the essential tools that we need to develop these theories.

References

[Boyer 79] R. S. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[DiVito 82] B. L. DiVito.
Verification of Communications Protocols and Abstract Process Models.
PhD thesis, University of Texas at Austin, 1982.
Technical Report 25, Institute for Computing Science.

[Gerhart 80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile.
An Overview of AFFIRM: A Specification and Verification System.
In *Information Processing 80, S. H. Lavington (Ed.)*, pages 343-348. October, 1980.
North Holland Publishing Company.

[Good 78] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

[Good 82] Donald I. Good.
The Proof of a Distributed System in Gypsy.
In *Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar - M. J. Elphick (Ed.)*. September, 1982.
Also Technical Report #30, Institute for Computing Science, The University of Texas at Austin.

[Jensen 74] K. Jensen, N. Wirth.
Pascal User Manual and Report.
Springer Verlag, 1974.

[Lampson 76] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.J. Popek.
Report on the Programming Language Euclid.
Technical Report, Xerox Research Center, August, 1976.

[McCarthy 62] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.
LISP 1.5 Programmer's Manual
MIT, 1962.

REUSABLE PROBLEM DOMAIN THEORIES

Table of Contents

List of Figures