Reusing Proofs of Program Correctness in ENCOMPASS

Robert B. Terwilliger

CU-CS-437-89   May 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Reusing Proofs of Program Correctness in ENCOMPASS

Robert B. Terwilliger

Department of Computer Science,
University of Colorado,
Boulder, CO 80309-0430

ABSTRACT

Many techniques can enhance the production of software. For example, mathematical verification techniques may help improve software quality, and reusability may greatly reduce the cost of software production. If a program's proof of correctness can be reused, higher quality may be achieved with reasonable cost. Unfortunately, reusing proofs of program correctness is difficult. In this paper we explore the approach being taken towards this problem in the ENCOMPASS project. Specifically, we present examples of three types of proof reuse: instantiating (reusing) a parameterized component and it's proof, reusing a development step with it's proof, and finally reusing a provably correct program schema. We believe that while program verification will in general remain expensive, the reuse of verified components may become practical through the use of such methods.

1. Introduction

Traditional methods do not ensure the production of correct software. It is unlikely that any one language, method or tool will completely solve this problem; however, many techniques may enhance the process [4]. The software correctness problem can be divided into *validation*, i.e. determining that the customer's desires have been correctly specified, and *verification*, i.e. certifying that the system satisfies its specification.

It has been suggested that *rapid prototyping* and the use of *executable specification languages* can aid in the validation process [1,24]. Prototypes can be used to enhance communication between customers and developers, in experiments performed to guide the design process, or possibly even be installed for use on a trial basis.

It has also been suggested that methods combining *stepwise refinment* with *formal proof* can help solve the verification problem [5,8,11,12]. In these methods, components are first described using using a mathematical notation; these specifications are then incrementally refined into implementations. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final implementations satisfy the original specifications. Since each refinement step is small, design and implementation errors can be detected and corrected sooner and at lower cost.

Such methods have been used in industrial environments to enhance the development process [3,13]. In these environments, the methods are typically not applied in all their formality. Formal specifications serve mostly as a tool for precise communication, and the major impact on methodology is that more time is spent on specification and design. However, the methods do prove useful in practice. Formal techniques could prove even more useful if they were applied more rigorously and supported by automated tools. Many feel the cost is justified, and environments to support such methods are being constructed [2,7,23].

ENCOMPASS [16,18,20] is an environment to support the incremental development of software using a combination of stepwise refinement and formal proof techniques. ENCOMPASS extends these methods with the use of executable specifications and testing-based verification. It provides tools to automate and support these techniques and integrates them as smoothly as possible into the traditional life-cycle.

Many feel that the *reuse* of components can greatly reduce the cost of software production and maintenance [6,10,25]. By reusing complex components in different contexts, the work required to produce them need only be performed once. Many different types of components can be reused including machine code, source code, specifications, and even system designs. The more complex a component is, the more effort is potentially saved by its reuse. The more dependent a component is on other objects, the more effort is potentially required for its reuse.

A program's proof of correctness is an extremely complex object which is highly dependent on the program itself; therefore, both the savings and costs of

This research was supported by grants from AT&T and USWest, as well as NSF Grant CCR-8809418

proof reuse are potentially great. If a program's proof of correctness can be reused, higher quality may be achieved at a reasonable cost; however, reusing proofs is a difficult problem. Very small changes to the text of a program can render the original proof invalid. In the worst case, both the original program and its proof must be examined and understood in their entirety to update the proof to the modified program. Fortunately, this worst case need not always occur; in may cases large parts of, or even entire, proofs of correctness can be reused without examination.

In this paper, we explore the approach being taken in the ENCOMPASS project towards reusing proofs of program correctness and present examples of three different types of proof reuse. In section two we give a brief overview of ENCOMPASS. In section three we describe the instantiation (reuse) of a parameterized component along with its proof, the primary technique for reuse in ENCOMPASS. In section four we show how a development step and its proof can be reused, and in section five, we discuss the use of a provably correct program schema. In section six, we summarize and draw some conclusions from our current experience.

2. ENCOMPASS

ENCOMPASS [16,18,20] is an environment to support the incremental development of software using a combination of executable specifications and stepwise refinement with formal proof techniques.

In ENCOMPASS, software is specified using a combination of natural language and the PLEASE [15,19] family of wide-spectrum, executable specification languages. The basic idea behind PLEASE is to execute pre- and post-conditions using logic programming techniques. For example, [15,16] describe an Ada based version of PLEASE in which pre- and post-conditions are translated into pure Prolog which is executed by a standard interpreter. [19] describes a C++ based version of PLEASE which uses a different execution mechanism. [15,16,21] give examples of PLEASE specifications.

In ENCOMPASS, PLEASE specifications are refined into implementations in conventional programming languages. PLEASE specifications are both formal and executable; therefore, refinements can be verified using any combination of peer review, formal proof or testing-based methods.

In ENCOMPASS, *verification conditions* are generated during the formal proof of a refinement step. Most of these VCs are certified using a number of simple (and inexpensive) proof tactics. Those not proved in this manner may be submitted to a more powerful (and expensive) general purpose theorem prover, be certified by a peer review process, or be saved for examination at a later time.

ENCOMPASS is an environment for the *rigorous* development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Proof techniques may be used that range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

In ENCOMPASS, a *knowledge-based assistant* [17] can also use the simple proof methods to perform deductive synthesis on specifications: automatically constructing simple fragments in the target programming language.

The ENCOMPASS environment has been under development since 1984. A prototype implementation became operational in 1986; it is described most succinctly in [16,18,20]. It contains a number of significant tools including ISLET [22], a prototype program/proof editor. This ENCOMPASS prototype has been used to develop about twenty programs, including specification, prototyping, and mechanical verification. At present, all the programs developed have been less than one hundred lines in length, but some have included more than one module, allowing demonstrations of the ENCOMPASS configuration control and project management systems.

ENCOMPASS has just completed the "proof of concept" stage. Even at this point, we feel we have shown that logic programming and conventional languages can be combined into executable specifications and that automated environments can provide significant support for formal development methods. Detailed conclusions on our technical approach can be found in [18]. We believe that the use of future environments similar to ENCOMPASS will greatly enhance the specification, design and development of software.

Our current experience has led us to believe that, in general, program verification will remain expensive for the foreseeable future; however, we are hopeful that the use of verified software can be made practical through the reuse of verified modules. The principal mechanism we are investigating towards this end is the use of parameterized components.

3. Parameterized Components

A *parameterized* component is like a template which must be instantiated with a number of arguments to produce an actual software structure. In the simplest case, instantiating a parameterized component is like expanding a macro. Many software structures can be usefully parameterized including *packages* [6], a language construct which hides some structures and makes others visible to the rest of the program.

For example, Figure 1 shows the specification of a parameterized sort package. An instantiation of *sort_pkg* takes three parameters: the type, *T*, of the elements to be considered; the type, *ST*, of the sequences to be sorted; and the relation, ≤, on which they are to be ordered. There are a number of constraints on the parameters to *sort_pkg*. The type *ST* must be structurally equivalent to *seq(T)*, and the boolean function ≤ must satisfy the conditions necessary for a binary relation to be a reflexive total order: it must be transitive, antisymmetric, total and reflexive. These constraints must be checked whenever the package is instantiated.

The package provides three structures to the rest of the program: a *sort* procedure as well as the predicates *perm* and *sorted*. All three operate on variables of type *ST*, or sequence of *T*. In our notation, the empty sequence is denoted by *()*, and the elements of a

```
pack sort_pkg(

    type    T ;
    type    ST = seq(T) ;
    func    ≤(T,T) : boolean ;
            where  (∀x,y,z:T.
                        (x≤y ∧ y≤z => x≤z) ∧
                        (x≤y ∧ y≤x => x=y) ∧
                        (x≤y ∨ y≤x) ∧
                         x≤x ) ;
) :

    pred sorted( var x : ST ) :
        (∀j,k:int.0≤j<k<|x|.x[j]≤x[k]) ;

    pred perm( var x,y : ST ) :
        x=y=()  ∨
        (∃f,b:ST.
            y=f||x[0]||b ∧
            perm(x[1..],f||b)) ;

    proc sort( B : ST ; var b : ST ) :
        pre : true ;
        post: perm(b,B) ∧ sorted(b) ;
```

Figure 1. Specification of parameterized *sort* package

sequence *b* are denoted by *b[0]*, *b[1]* ... *b[n-1]*, where *n* is the length of the sequence. The *k*th element of a sequence is denoted by *b[k]*, the *k*th through final elements are represented by *b[k..]*, and the *j*th through *k*th elements are represented by *b[j:k]*. The concatenation of sequences *b* and *c* is denoted by *b*||*c*, and the length of a sequence *b* is denoted by |*b*|.

The specification is written in a notation which emphasizes specification and design concepts rather than programming language syntax. In ENCOMPASS, our present implementation efforts center on C++, and we have done considerable work in Ada; however, our methods are in general independent of particular programming languages. We therefore present the specifications and programs in this paper using a guarded command style notation [5].

The *sort* procedure takes a possibly unordered sequence (*B*) as input and produces a permutation which is sorted as output (*b*). The procedure is specified using a *pre-condition* that states the properties required of valid inputs and a *post-condition* that describes the relationship of inputs to outputs. *Pre* is simply *true*: the parameter declarations specify all the properties for valid input. *Post* states that the final value of *b* must be a *perm*utation of *B* and also be *sorted*. Since *B* is not a *var* parameter, its value can not be changed in the procedure. *Pre* and *post* use the predicates *perm* and *sorted*, which are also defined by *sort_pkg*.

In ENCOMPASS, a *predicate* is similar to a boolean function in that it returns a truth value; however, it differs in that it can modify its arguments while functions may not. Predicates are defined using logical expressions which can be translated into definite clauses and executed using logic programming techniques. This allows the construction of prototypes from pre- and post-condition pairs.

Although a predicate may modify its arguments, it need not always do so. In ENCOMPASS, a procedure or predicate invocation must indicate which of the actual parameters may be modified. The set of arguments which can actually be modified by an invocation is the intersection of the sets defined as *var* by the definition and call. If a formal parameter is marked as *var* in the definition, but the corresponding argument in a call is not, then the system makes a copy of the argument for use in the call. A predicate invocation with none of the parameters marked as *var* is identical to a boolean function invocation and may appear, for example, in the condition of an *if* statement.

In ENCOMPASS, a predicate definition syntactically resembles a procedure or function and may contain local type, constant, variable or predicate definitions. For example, the predicate *sorted* states that a sequence *x* is sorted if the relation ≤ holds for every pair of elements *b[j]*, *b[k]* in the sequence such that *j<k*. Similarly, the predicate *perm* states that two sequences *x* and *y* are permutations of each other if they are both empty, or if the first element in *x* appears somewhere in *y* and the remainders of *x* and *y* are permutations of each other.

Sort_pkg can be instantiated to create a *sort* procedure for sequences of any element type. For example, the following instantiation binds the formal parameters *T*, *ST* and ≤ to the actual types *int* and *Sint* and the ≤ relation on integers respectively.

```
type Sint = seq(int) ;
<sort_pkg>(int,Sint,int::≤(int)) ;
```

This instantiation produces a *sort* procedure as well as *perm* and *sorted* predicates for the type *Sint*.

Sort_pkg may have a number of different implementations. The source code for each of theses implementations can be reused for each instantiation of the package. The proofs of these implementations can use no information about the type *T* except the fact that ≤ is a reflexive total order. Since this is specified as a constraint, and is checked when the package is instantiated, each implementation's proof of correctness can also be reused for each instantiation.

While this level of proof resue is significant, higher levels are possible as it is also possible to share "proof parts" between the different implementations. This is possible because the construction of different implementations may involve some of the same development steps.

4. Development Steps

Assume we are given the task of developing a program from the specification given in Figure 1. First, we notice that any sequence of length one or less is both *sort*ed and a *perm*utation of itself; therefore, when called with such a sequence as input the *sort* procedure can simply return it as output. This insight allows us to refine the original pre- and post-condition into the incomplete procedure body shown in Figure 2.

This is an example of a *development step*: we have gone from a pre- and post-condition pair to a provably correct program fragment. The fragment may not be a complete program; it may contain assertion pairs without interveining code. When these "blanks" are filled in with fragments which satisfy the corresponding pre- and post-conditions, a program which satisfies the original specification is produced.

The procedure body in Figure 2 contains a single *if* command, the semantics of which differ from the *if* statement in C, Pascal, or Ada [5,8]. In a guarded command notion, an *if* can have any number of

```
proc body sort :

{Q: true}
if |B|≤1 →   b:=B ;
 ▯ |B|>1 →   {Q₁: |B|>1}
             <S₁>
             {R₁: perm(b,B) ∧ sorted(b)}
fi
{R: perm(b,B) ∧ sorted(b)}
```

Figure 2. First step of sort development

alternatives, each consisting of a command and its associated guard. If none of the guards is true when the *if* is executed then the the program aborts. If at least one guard is valid then a non-deterministic choice is made from the valid ones and the command associated with the guard is executed.

For example, the *if* in Figure 2 conists of two alternatives: the command *b:=B*, with guard *|B|≤1*; and the as yet unknown program fragment represented by $<S_1>$, with guard *|B|>1*. When the *if* is executed either *B* has no more than one element and its value is assigned to *b*, or *B* has more than one element and $<S_1>$ is executed.

To prove the development step correct we must show that if execution of the procedure begins in a state which satisfies the pre-condition, then the *if* statement will terminate normally in a state which satisfies the post-condition. This proof consists of two parts. First, we must show that at least one of the guards will always be true, so that the *if* will not abort. Second, we must show that if either guard is true then execution of the corresponding command will result in a state which satisfies the post-condition. The proof is independent of any properties of the type *T*; it relies only on the properties of sequences and integers, plus the assumption that S_1 satisfies its pre- and post-conditions.

We now must develop a fragment which satisfies the pre- and post-conditions Q_1 and R_1 respectively. Assume we decide to implement an *insertion sort*. We will maintain *b* as a sorted permutation of a prefix of *B*. We will take an element at a time from the part of *B* not included in the prefix and insert it into the proper place in *b*. The result of this development step is shown in Figure 3.

The program fragment in Figure 3 consists of a *do* loop and its initialization. As with the *if*, the

```
{Q₁: |B|>1}
const n:int := |B| ;
k,b:=0,() ;
do
    {inv P₁: 0<n ∧ 0≤k≤n ∧
             perm(b,B[0:k-1]) ∧ sorted(b)}
    {bnd t₁: n-k}
    k≠n → {Q₂: k≠n ∧ P₁}
          <S₂>
          {R₂: (P₁)^k_{k+1}}
          k:=k+1
od
{R₁: perm(b,B) ∧ sorted(b)}
```

Figure 3. Second step of sort development

semantics of the *do* differ from the *while* in other programming languages. Like the *if*, a *do* loop can have any number of alternatives, each consisting of a command and its associated guard. When program execution reaches the *do*, a non-deterministic choice is made from the valid guards and the associated command is executed. This process is repeated until all the guards are false.

For example, the *do* in Figure 3 conists of a single alternative with guard *k≠n*. The command associated with this guard is the unknown fragment $<S_2>$ followed by the assignment *k:=k+1*. When the loop is executed, $<S_2>$ and *k:=k+1* are repeatedly performed until *k* becomes equal to *n*.

The loop has both an *invariant* (P_1), which states the properties that must be maintained by each execution of the loop body; and a *bound* function (t_1), which puts an upper limit on the number of iterations remaining for the loop. Using this information, the proof of the second development step consists of five parts [8].

First, we must show that the invariant is true at the beginning of loop execution. Second, we must show that the invariant is maintained by each execution of the loop. Third, we must show that termination of the loop with the invariant true will result in a state which satisfies the post-condition. Fourth, we must show that the bound is positive if the loop is running, and finally we must show that each iteration of the loop decreases the bound.

We must now develop a fragment which satisfies the pre- and post-conditions Q_2 and R_2, respectively. One solution is to first place the new element at the beginning of *b* and then "bubble" it into place by

```
{Q₂:    0<n ∧ 0≤k<n ∧
        perm(b,B[0:k-1]) ∧ sorted(b)}
j,b:=0,B[k] ‖ b ;
do
    {inv P₂: 0<n ∧ 0≤j≤k<n ∧
             perm(b,B[0:k]) ∧
             sorted(b[0:j-1] ‖ b[j+1..]) ∧
             b[0:j-1]≤b[j]}
    {bnd t₂: k-j}
    j≠k cand b[j+1]≤b[j] →
             {Q₃: P₂ ∧ j≠k ∧ b[j+1]≤b[j]}
             b[j],b[j+1]:=b[j+1],b[j] ;
             {R₃: (P₂)^j_{j+1}}
             j:=j+1 ;
od
{R₂:    0<n ∧ 0≤k+1≤n ∧
        perm(b,B[0:k]) ∧ sorted(b)}
```

Figure 4. Fourth step - completed inner loop

swapping; Figure 4 shows such an implementation. Two steps are required to produce Figure 4 from Figure 3: the first produces a loop with a partially unknown body, and the second implements the unknown with an assignment statement.

The proof of the loop is similar to that discussed previously. The proof of the assignment is simplified because functions have no side effects, and all the expressions on the right hand side are evaluated before any of the values are stored. Therefore, the assignment *b[j],b[j+1]:=b[j+1],b[j]* swaps the *j*th and *j+1*st elements in the sequence *b*. To prove an assignment correct we simply show that the pre-condition implies the post-condition with the right hand side of the assignment substituted for the left hand side [8].

The proofs of these development steps are independent of the type *T*, but are highly dependent on the properties of the relation ≤. Care must be taken to produce a proof which uses only the properties of ≤ stated in the specification. For example, the loop condition in Figure 4 might just as well be written as *b[j]≥b[j+1]* or *b[j]>b[j+1]* rather than *b[j+1]≤b[j]*. However, from the specification alone we do not know the meaning of these expressions. Our intuition tells us that ≤ is the negation of >, but the specification only states that ≤ is a reflexive partial order.

Although the steps in the preceding development follow one another in sequence, their proofs are independent. The result, and proof, of a step can be used to derive many alternative implementations. For

```
{Q2:    0<n ∧ 0≤k<n ∧
        perm(b,B[0:k-1]) ∧ sorted(b)}
j:=0;
do
    {inv P2: 0<n ∧ 0≤j≤k<n ∧
             perm(b,B[0:k-1]) ∧ sorted(b) ∧
             b[0:j-1]≤B[k]}
    {bnd t2: k-j}
    j≠k cand b[j]≤B[k] → j:=j+1 ;
od
{P2 ∧ (j=k cor ¬(b[j]≤B[k])}
b:=b[0:j-1] || B[k] || b[j..];
{R2:    0<n ∧ 0≤k+1≤n ∧
        perm(b,B[0:k]) ∧ sorted(b)}
```

Figure 5. Alternative implementation

example, another implementation of the inner loop might scan *b* for the proper location, and then perform the insertion with a single assignment. Figure 5 shows such an implementation, which can be produced in two development steps beginning with Figure 3.

The implementations in Figure 4 and Figure 5 share the first two steps in their developments, but also have two unique steps of their own. The proofs of these shared steps can be completely reused without examination. Both programs implement an insertion sort. This is an example of proof reuse within *program families*. While this is significant, even greater degrees of reuse can be achieved by combining a number of design steps into a program schema.

5. Program Schemas

For example, Figure 6 shows a *divide_and_conquer* schema. In a divide and conquer algorithm, the problem to be solved is first divided into a number of sub-parts. These sub-problems are then solved to produce a number of sub-solutions that are combined to produce a solution to the original problem.

The schema in Figure 6 is written as a parameterized fragment. In ENCOMPASS, a *fragment* is a language level construct similar to a non-recursive procedure. However, unlike a procedure a fragment can have no global references: all variables used in the fragment must be declared within it or passed as arguments. Also, while a procedure is invoked, a fragment is instantiated: the arguments are first substituted for the formal parameters and then the resulting code is substituted in-line for the "call".

```
frag divide_and_conquer(

    type   T1, T2,
           ST1 = seq(T1), ST2 = seq(T2) ;

    var    input:T1,  output:T2,
           ip   :ST1, op    :ST2 ;

    pred   valid    ( T1 ),
           divided  ( T1, ST1),
           conquered( T1, T2 ),
           combined ( ST2,T2 ),
           solved   ( T1, T2 ) ;
           where (∀x:T1,xp:ST1,y:T2,yp:ST2.
                  divided(x,xp) ∧
                  (∀k:int.0≤k<|xp|.
                  conquered(xp[k],yp[k]))
                  ∧ combined(yp,y) =>
                            solved(x,y)) ;

    frag   divide ( x:T1, var xp:ST1 ) :
              pre : valid(x) ;
              post: divided(x,xp) ;
    frag   conquer( x:T1, var y:T2 ) :
              pre : true ;
              post: conquered(x,y) ;
    frag   combine( xp:ST2, var x:T2 ) :
              pre : true ;
              post: combined(xp,x) ;

) :

    pre :  valid(input) ;
    post:  solved(input,output) ;

    <divide>(input,ip) ;
    var k:int := 0 ;
    do
        {inv P:    0≤k≤|ip| ∧
                   divided(input,ip) ∧
                   (∀j:int.0≤j<k.
                   conquered(ip[j],op[j]))}
        {bnd t:    |ip|-k}
        k≠|ip| →  <conquer>(ip[k],op[k]) ;
                   k:=k+1 ;
    od
    <combine>(op,output) ;
```

Figure 6. *divide_and_conquer* schema

In ENCOMPASS, a schema is not a language level construct; rather, a schema is a fragment that abstracts some important feature common to a class of programs. The distinction between a schema and a fragment is more one of intent than substance; for example, the fragments in Figure 2 and Figure 3 can be viewed as schemas. However, we do not feel that Figure 2 captures the essence of any important class of programs. Figure 3 can be viewed as an insertion sort schema.

The *divide_and_conquer* schema takes a number of parameters. T_1 and T_2 are the types of the input and output respectively, while *input* and *output* are the variables which hold these quantities. ST_1 and ST_2 are the types of the sub-problems and sub-solutions respectively, while *ip* and *op* are the variables used to store them. The predicate *valid* defines the allowable inputs, while the predicate *divided* describes the proper division of the input into sub-parts. The predicate *conquered* specifies the correct solution of a sub-problem; the predicate *combined* describes how the sub-solutions are assembled to produce a global solution; and the predicate *solved* defines the correct solutions to the entire problem.

Divide_and_conquer also takes three program fragments as parameters. The fragment *divide* partitions the input into sub-parts. *Divided* takes two parameters, the input sequence and the variable which holds the sequence of sub-parts into which the input is divided. The pre-condition for *divide* states that the input is *valid*, while the post-condition states that the input is correctly *divided*. The fragment *conquer* solves a single sub-problem and stores the sub-solution, while the fragment *combine* takes the sub-solutions and assembles them into a solution to the entire problem.

The body of *divide_and_conquer* consists of single *do* loop with both an initial and finalization. When an instantiation of *divide_and_conquer* is invoked, the fragment *divide* is first executed to divide *input* into sub-parts that are stored in *ip*. The *do* loop is then executed. This causes the fragment *conquered* to be executed on each sub-part of the input, storing the computed sub-solution in the appropriate element of *op*. Finally, the fragment *combine* is executed to assemble the sub-solutions in *op* to produce *output*.

An instantiation of *divide_and_conquer* is correct with respect to its pre- and post-conditions as long as the constraints on the parameters to the schema are met: the types of all arguments must match and satisfy the structural constraints; the actual fragments *divide*, *conquer* and *combine* must satisfy the pre- and post-conditions given in the schema; and the predicates *divided*, *conquered*, *combined* and *solved* must satisfy the stated constraint. The schema's proof also makes use of language level information such as the value or *var* declarations of the fragment parameters.

The constraint on the predicates *divided*, *conquered*, *combined* and *solved*:

```
(∀x:T₁,xp:ST₁,y:T₂,yp:ST₂.
       divided(x,xp) ∧
       (∀k:int.0≤k<|xp|.
           conquered(xp[k],yp[k]))
      ∧ combined(yp,y) =>
                 solved(x,y)) ;
```

is at the heart of the schema's proof of correctness. The antecedent states the properties we know are true following execution of the fragment *combine*, while the consequent is that the entire problem has been correctly solved.

The proof of the schema is complicated by the fact that it involves *invariants* for each fragment instantiation. The proof rule for fragment instantiation used in ENCOMPASS is similar to the rule for non-recursive procedure calls [16]. To use the rule, one needs to define an *invariant* for each instantiation. The invariant states properties that are necessary for the proof of the rest of the program, but are not effected by execution of the fragment.

For example, the invariant for the *conquer* fragment is the same as the invariant for the loop which encloses it (labeled *P* in Figure 6). Intuitively, at the beginning of a loop iteration we know a number of sub-problems have been solved and that *conquer* will solve another sub-problem when it is executed. For the loop to be correct, we also must know that *conquer* will not tamper with any of the sub-solutions previously computed. The proof of this relies on language level information such as the fact that passing a sequence element by reference does not give access to the entire sequence, and that predicates can contain no non-local references.

Continuing our example, assume we want to implement a recursive quicksort algorithm. In quicksort, an input element is selected and the rest of the input is divided into two sub-sequences such that all the items in one are less than or equal to the element, and all the items in the other are greater than or equal to the element. *Sort* is then recursively called with these sub-sequences and the results concatenated to form the output. The quicksort algorithm can be seen as an instantiation of *divide_and_conquer* with selection and partitioning as *divide*, the recursive calls to *sort* as *conquer*, and the concatenation of the sorted sub-sequences as *combine*

Beginning with the incomplete procedure body shown in Figure 2, we can implement quicksort by replacing $<S_1>$ with the instantiation of *divide_and_conquer* shown in Figure 7. The instantiation defines *SSint*, or sequence of *Sint*, as the type of the sub-problems and sub-solutions. In our notation, a boolean function applied to a sequence is a syntactic abbreviation for element-wise invocation. For

```
type   SSint  = seq(Sint) ;
var    Bp,bp  : SSint     ;

pred   valid     ( var x:Sint ) : |x|>1 ;
pred   is_part   ( var in:Sint,
                   var ip:SSint ) :
               perm(in,ip[0] || ip[1] || ip[2])
               ∧ ip[0]≤ip[1]≤ip[2] ;
pred   combined  ( var op:SSint,
                   var out:Sint ) :
                 out=op[0] || op[1] || op[2];
pred   is_sorted( var x,y:Sint ) :
                 perm(x,y) ∧ sorted(y) ;

frag partition( in:Sint, var ip:SSint ) :
       pre : valid(in) ;
       post: is_part(in,ip) ;

       var  j:int := 1 ;
       ip[0],ip[1],ip[2]:=(),(in[0]),() ;
       do
           {inv:  0≤j≤|in| ∧
                  is_part(in[0:j-1],ip)}
           {bnd:  |in|-j}
           j≠|in|  →
                  if in[j]≤ip[1] →
                      ip[0]:=in[j] || ip[0];
                   □ ip[1]≤in[j] →
                      ip[2]:=in[j] || ip[2];
                  fi ;
                  j:=j+1 ;
       od

frag sortf( in:Sint, var out:Sint ) :
       pre : true ;
       post: is_sorted(in,out) ;
       sort(in, var out) ;

frag combine( op:SSint, var out:Sint ) :
       pre :  true ;
       post:  combined(op,out) ;
       out := op[0] || op[1] || op[2] ;

<divide_and_conquer>(
    Sint,Sint,SSint,SSint,B,b,Bp,bp,
    valid,is_part,is_sorted,combined,
    is_sorted,partition,sortf,combine ) ;
```

Figure 7. Instantiation of *divide_and_conquer*

example, the expression *in[j]≤ip[1]* is an abbreviation for *(∀k:int.0≤k<|ip[1]|.in[j]≤ip[1][k])*.

The instantiation declares *Bp* and *bp* as the variables used to hold the sub-problems and sub-solutions respectively. It also defines the predicates *valid*, *is_part*, *combined*, and *is_sorted*, as well as the fragments *partition*, *sortf*, and *combine*.

The fragment *partition* and predicate *is_part* are central to the quicksort algorithm. *Partition* divides the input into three parts. The first element of the input is taken as the *pivot* and is stored in *ip[1]*. The remainder of the input is then divided using this pivot. *Is_part* describes the result of a successful partition: all the elements in *ip[0]* are less than or equal to the pivot, all the elements in *ip[2]* are greater than or equal to it, and the concatenation of *ip[0:2]* forms a permutation of the input.

Figure 8 shows the implementation of quicksort produced by the instantiation in Figure 7. In our notation, declarations can appear at any point in a program and are viable from the point they appear until the end of the structure in which they are contained. A declaration can be over-ridden by one for an object of the same name and type appearing later in the scope.

Most of the predicate and fragment declarations in Figure 7 are only for the purpose of schema instantiation. Since they are not necessary to the final program, they do not appear in Figure 8. We have not resolved how these temporary declarations can best be handled in language-oriented tools that support such development methods. One approach would be to have a multi-window interface with a separate, temporary window for each instantiation.

One problem remains. We have constructed a recursive quicksort procedure; however, the proof of *divide_and_conquer* does not take the possibility of a recursive invocation into account. Therefore, we must separately prove that the recursion will terminate. This can be accomplished in a manner similar to that used to prove termination of a loop. We define a bound function of the procedure parameters and show that it is greater than zero for any procedure invocation and that each recursive invocation decreases the bound. In Figure 8, this reduces to showing that each of the sub-parts into which the input is divided is smaller than the original sequence.

This completes the development and proof of quicksort; the proof of *divide_and_conquer* has been reused in its entirety. While the proof of the instantiation is complex, it is considerably simpler than proving the final program correct from scratch. We feel that program schemas are a very powerful technique for reuse. The instantiation of a schema can reuse a very complicated structure in a context much different from that in which it was created.

At this point, we may wonder how general this technique is. For example, would it be possible to

```
proc sort( B : ST ; var b : ST ) :

{Q: true}
if |B|≤1 →   b:=B ;
 ▯ |B|>1 →
    type SSint ;
    var  Bp,bp : SSint ;
    pred is_part( var in:Sint,
                  var ip:SSint ) :
              perm(in,ip[0] || ip[1] || ip[2])
              ∧ ip[0]≤ip[1]≤ip[2] ;
    var  j:int := 1 ;
    Bp[0],Bp[1],Bp[2] := (),(B[0]),() ;
    do
        {inv:  0≤j≤|B| ∧
               is_part(B[0:j-1],Bp)}
        {bnd:  |B|-j}
        j≠|B|  →   if B[j]≤Bp[1] →
                       Bp[0]:=B[j] ||Bp[0];
                    ▯ Bp[1]≤B[j] →
                       Bp[2]:=B[j] ||Bp[2];
                   fi ;
                   j:=j+1 ;
    od
    pred is_sorted( var x,y:Sint ) :
                   perm(x,y) ∧ sorted(y) ;
    var k:int := 0 ;
    do
        {inv P:    0≤k≤|Bp| ∧
                   is_part(B,Bp) ∧
                   (∀j:int.0≤j<k.
                   is_sorted(Bp[j],bp[j]))}
        {bnd t:    |Bp|-k}
        k≠|Bp|  →  sort(Bp[k],var bp[k]) ;
                   k:=k+1 ;
    od
    b:=bp[0] ||bp[1] ||bp[2] ;
fi
{R: perm(b,B) ∧ sorted(b)}
```

Figure 8. Implementation of quicksort algorithm

reuse the proof of the *divide_and_conquer* schema, or the quicksort implementation already developed, in the development of a procedure that performed an in-place quicksort (in other words, using the same sequence as both input and output)? While we have only limited experience in this area, we feel a key to achieving such reuse is program coordinate transformation.

A *program coordinate transformation* [5,9] modifies the underlying state space on which a program operates; for example, a coordinate transformation could change both the names and types of the variables in a program. This allows a schema written in terms of abstract data elements to be transformed into many different programs that solve concrete problems. While the coordinate transforms themselves have to be proved correct, the schema's proof can be reused in its entirety.

For example, *greedy algorithms* can be used to solve many problems in combinatorics. Any problem which can be optimally solved using a greedy algorithm can be written in terms of a matroid [14], a subset system which satisfies certain axioms. We are now working on transformational developments of a number of greedy algorithms from a schema written in terms of a matroid.

6. Summary and Conclusions

Mathematical verification techniques may help improve software quality [5,8,11,12]; however, the cost of such methods is high. One promising approach to reducing the cost of software production and maintenance is the reuse of components [6,10,25]. We believe that while program verification will in general remain expensive, the use of verified software may become practical through the reuse of verified modules.

ENCOMPASS [16,18,20] is an environment to support the incremental development of software using a combination of stepwise refinement and formal proof techniques. ENCOMPASS extends these methods with the use of executable specifications and testing-based verification. It provides tools to automate and support these techniques and integrates them as smoothly as possible into the traditional life-cycle.

In ENCOMPASS, we are addressing the problem of proof reuse through the mechanism of parameterized components. A parameterized component is like a template that must be instantiated with a number of arguments to produce an actual software structure. There are constraints on the parameters that define the conditions necessary for reuse of the component's proof of correctness.

A component may simply specify a structure; for example, a procedure or function. Such a component may have many different implementations which form a program family. The developments of different programs within the family may share intermediate steps, the proofs of which can be reused. If the original component is parameterized, then each program in the family (and its proof) can also be reused for each instantiation.

While these types of proof reuse are significant, we feel the greatest potential lies in the use, and reuse, or program schemas. A program schema is a fragment which abstracts an important feature common to a class of programs. For example, a schema might describe the essential features of divide and conquer or greedy algorithms. Instantiation of program schemas allows the reuse of considerable structure in a large number of different contexts.

A program schema typically takes fragments as parameters; therefore, the proof of a schema instantiation can be complex. However, such proofs should be considerably simpler than proving the resulting program correct from scratch. Program coordinate transformations [5,9] may further increase the utility of such techniques.

In ENCOMPASS, we have implemented tools to support the reuse of parameterized components and development steps along with their proofs [16,22]. We have also implemented somewhat limited support for the reuse of program schemas and their proofs of correctness [17]. Although these tools are still research prototypes, we feel that their long term potential is great. We believe that eventually the reuse of verified components can greatly enhance the development of high quality software.

7. References

1. Balzer, R., T. E. Cheatham and C. Green, "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer 16, 11* (November 1983), 39-45.
2. Bjorner, D., T. Denvir, E. Meiling and J. S. Pedersen, "The RAISE Project - Fundamental Issues and Requirements", RAISE/DDC/EM/1, Dansk Datamatik Center, 1985.
3. Bloomfield, R. E. and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 988-993.
4. Brooks, F. P., "No Silver Bullet", *IEEE Computer 20, 4* (April 1987), 10-19.
5. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.
6. Goguen, J. A., "Reusing and Interconnecting Software Components", *Computer 19, 2* (February 1986), 16-28.
7. Goguen, J. and M. Moriconi, "Formalization in Programming Environments", *IEEE Computer 20*, 11 (November 1987), 55-64.
8. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
9. Gries, D., *The Transform -- Working Notes*, Dept. of Computer Science, Cornell University, March 1989.
10. Horowitz, E. and J. B. Munson, "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering SE-10, 5* (September 1984), 477-487.
11. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
12. Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.
13. Oest, O. N., "VDM From Research to Practice", *Information Processing*, 1986, 527-533.
14. Papadimitriou, C. H. and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
15. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the *Journal of Systems and Software*), September 1986.
16. Terwilliger, R. B., "ENCOMPASS: an Environment for Incremental Software Development using Executable, Logic-Based Specifications", Report No. UIUCDCS-R-87-1356 (Ph.D. Dissertation), Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.
17. Terwilliger, R. B., "An Example of Knowledge-Based Development in ENCOMPASS", *Proceedings of the Third Annual Conference on Artificial Intelligence & Ada*, George Mason University, October 1987, 40-55.
18. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 344-354.
19. Terwilliger, R. B. and P. A. Kirslis, "PK/C++: an Object-Oriented, Logic-Based, Executable Specification Language", *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989, 407-416.
20. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software", *Journal of Systems and Software* (also Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign) *10*, 1 (July 1989).
21. Terwilliger, R. B., M. J. Maybee and L. J. Osterweil, "An Example of Formal Specification as an Aid to Design and Development", *Proceedings of the 5th International Workshop on Software Specification and Design*, May 1989.
22. Terwilliger, R. B., "ISLET: a Program/Proof Editor to Support the Vienna Development Method", *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989, 68-77.
23. Yuasa, T. and R. Nakajima, "IOTA: A Modular Programming System", *IEEE Transactions on Software Engineering SE-11, 2* (February 1985), 179-187.
24. "Theme Issue: Rapid Prototyping", *IEEE Computer 22*, 5 (May 1989).
25. "Theme Issue: Tools, Making Reuse a Reality", *IEEE Software 4*, 4 (July 1987).