IMPROVING SOFTWARE TESTABILITY WITH ASSERTION INSERTION*

Hwei Yin† James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096, Fax: (303) 491-2466
yin@ssdevo.enet.dec.com, bieman@cs.colostate.edu

Published in *Proc. International Test Conference*, October 1994 (ITC94).

ABSTRACT

Executable assertions can be inserted into a program to find software faults. Unfortunately, the process of designing and embedding these assertions can be expensive and time consuming. We have developed the C-Patrol tool to reduce the overhead of using assertions in C programs. C-Patrol allows a developer to reference a set of previously defined assertions, written in *virtual C*, bind assertion parameters, and direct the placement of the assertions by a pre-processor.

1 INTRODUCTION

Developing reliable software is difficult and requires discipline both in specifying system functionality and in implementing systems correctly. Run time assertion checking is one technique that can help locate defects and insure that programs satisfy specified constraints [10]. Executable assertions have several practical uses:

- They can serve as automated test oracles — automatic means for determining if program output, or a program state, is correct.
- They can serve as diagnostic tools to help locate software faults that cause specific failures. Such diagnostic tools can be useful both during debugging and while code is under maintenance.
- They can serve as dependable program documentation. A testing protocol can assure the consistency between assertion documentation and normal program code.
- They can be used to enforce particular programming styles that are not supported by an implementation language. C-Patrol, for example, provides the user with data abstraction support that is not available in C or many other languages.

In effect, executable assertions are a form of dual programming. Unfortunately, the effort required to design and embed assertions into programs can be significant. Executable assertions will become more useful and more commonly used if the overhead is limited.

In our research, we have carefully evaluated the use of executable assertions in an industrial setting in order to develop useful support tools. We find that executable assertions are used extensively for software testing. They are hand coded and placed at desired locations to indicate when and where a program is in a faulty state. The assertions are written as regular C code; no formal specification language is used. The assertions are often designed to check that data values satisfy specified constraints. Assertions are defined in terms of specific data declarations, and they must be placed where the data values are referenced or modified.

The industrial developers that we observed use executable assertions primarily for checking ad-hoc data requirements that arise during program development. The assertions could be used to enforce constraints

*Research partially supported by the Colorado Advanced Software Institute (CASI), Storage Technology Inc., and Micro-Motion Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

†Hwei Yin is now with Digital Equipment Corp. in Colorado Springs, Colorado.

developed directly from a requirements specification rather than from the program itself. The developers need a mechanism to associate assertions with data objects as well as specific program states. They need constructs to increase the expressiveness of assertions, and support the reuse of assertions. These needs motivate our development of the C-Patrol tool.

In Section 2, we describe the current C-Patrol prototype and its major feature, the labeled code system, and we review potential uses for the system in Section 3. Section 4 contains our analysis of the critical design decisions and the issues that were instrumental in shaping the current system. We survey related work in Section 5, and our conclusions are in Section 6.

2 C-PATROL DESCRIPTION

C-Patrol is a pre-processor that inserts assertions, written in *virtual C*, into specified locations in C programs. C-Patrol design concepts are independent of C itself; thus, major portions of the implementation can be easily modified to support assertion insertion for programs in other languages. C-Patrol is similar to Anna [7] in that assertions are written as comments that can be textually converted into C code by a pre-processor. C-Patrol, however, is much simpler than Anna and derives its expressive power from a procedure-like mechanism, the *labeled code system*, rather than from a large inventory of high-level constructs. A prototype has been implemented; results from the user testing of this prototype will undoubtably uncover new uses for the system and will heavily influence future developments and design goals.

At its simplest level, C-Patrol is simply a code insertion technique. The user places *virtual C code* within special *C-Patrol comments*. These comments are skipped by the C compiler and do not affect the performance of the underlying system. To activate this virtual code, the user invokes the *C-Patroller*, a pre-processor that translates virtual code into regular C and inserts it following instructions called *directives*. This augmented program may then be compiled and run as normal C.

In the current design, virtual code is C code interspersed with directives. No special meta-language needs to be learned, and there are no restrictions placed on the virtual code written. Users are responsible for ensuring that virtual code does not produce any undesired side-effects.

Simplicity is the crucial theme that pervades C-Patrol design. In general, the system will not attempt to control the behavior of the user; it is up to the user to be self-regulating (following the design philosophy of C). To make this task easier, the system is designed to be as intuitive and as easy to understand as possible so that the user can readily comprehend the consequences of his or her actions before they are taken.

2.1 Code Insertion Directives

Insertion directives simply control where code is to be inserted:

```
/*? %%insert:
        printf("hello");
        x = f(y);
        %%call r, s;
        printf("goodbye");   ?*/
```

In this example, the `/*?` and `?*/` tokens delimit the C-Patrol comment, and the `%%insert` directive indicates that the enclosed virtual code is to be inserted exactly where it appears in the surrounding code. The `%%call` directives that appear within the virtual code will be replaced by normal C before insertion (a process to be explained later). Since virtual blocks will consist entirely of standard C, the resulting insertion can be compiled and executed along with the rest of the surrounding C code.

Other insertion directives, such as the `%%pre` and `%%post` directives, specify insertion at the entry and exit points of a target function. These directives provide a clear and simple format for specifying code that is to be executed on function boundaries.

2.2 Labeled Code

Users can control the insertion of previously defined assertions using the C-Patrol system of *labeled code*. Like macros, labeled code consists of a block of virtual code that is inserted when invoked by a `%%call` directive. Unlike procedures or functions, labeled code is not identified by a single name, but by a *label* consisting of a series of tags called *label identifiers*:

```
/*? %%label bill, ted:
        printf("code block one");
        printf("one done");
    %%label ted;
        printf("code block two");
    %%label bill, fred:
        printf("code block three");
?*/
```

Unlike procedure or macro names, label identifiers are not unique to blocks of code. Virtual code in labeled blocks consists only of pure C – no special C-Patrol directives or parameters may appear within. A call directive invokes labeled code by making reference to individual identifiers rather than entire labels. Thus:

```
%%call bill, fred;
```

will cause the first and third blocks to be inserted in their declared order since identifiers in their labels are mentioned by the call. Essentially, `%%call` refers to all labels that contain *any* of the identifiers listed. Thus, the labeled code system is much like a database system, where keywords (label identifiers) access related records (labeled blocks) by association.

2.3 Extensions to the Labeled Code System

The user can subdivide label identifiers through the use of *subfields*:

```
/*? %%label bill.x.v, bill.y:
        assert(bill.x.v < bill.y);
    %%label display, bill;
        printf("bill y:%d, x.v:%d, x.q:%s\n",
               bill.y, bill.x.v, bill.x.q);
?*/
```

In the example, the label identifier `bill` has been divided into subfields `x` and `y`; subfield `x` has been further divided by subfield `v`. The resulting hierarchy of identifiers provides a simple but powerful addition to the labeled code system. Note how we use the labeling system in conjunction with the `assert` and `printf` facilities to enforce and display data-oriented information about object `bill`.

The *exclusive call* is an alternative to the normal call directive that allows greater power in "weeding out" unwanted invocations of labeled blocks:

```
%%ex-call r, s.v
```

No label that contains items outside of those listed in the `%%ex-call` will have its code included.

Labeled code directives can be viewed in terms of mathematical sets:

- Label identifiers: Label identifiers describe individual, disjoint sets.
- Commas: Commas between listed identifiers indicate set union. Thus, the space described by listing several identifiers in a label or call is the union of the individual sets.
- Subfields: Subfields within an identifier describe disjoint strict subsets within the parent set.
- `%%call`: The `%%call` invokes a label if the space described by the call intersects *in any way* with the space of the label; in other words, the intersection between the two sets must be non-null.
- `%%ex-call` The `%%ex-call` invokes a label only if the space described by the label is a subset of the call. Thus, the `%% ex-call` is much more restrictive than the `%%call`, but provides the user with more control over which code is invoked.

2.4 Templates

Template blocks of virtual code provide flexibility through the use of parameters. Template declaration is similar to the declaration of a traditional procedure or macro:

```
/*? %%template printme(a, b):
      printf("%d %d", a.c, b.c);   ?*/
```

The template `printme` has two parameters: `a` and `b`. When given character string bindings, the C-Patroller will look for the `a` and `b` tokens within the virtual block and textually substitute them for the passed strings.

Template blocks can be converted into normal labeled blocks through the use of *binding* directives:

```
/*? %%bind r =
      printme("is the value", my_array[j]);
?*/
```

The `printme` template (defined earlier) is passed the string `"is the value"` and `my_array[j]` for parameters `a` and `b` respectively. The resulting code is then given the label `r`. Thus, the above binding directive is equivalent to the declaration:

```
/*? %%label r:
    printf("%d is the value", my_array[j].c);
?*/
```

Note that identifier `r` may still be used in other labels.

Templates may be invoked directly by `%%call` and `%%ex-call` directives, as long as all parameters are bound.

2.5 Pre-Processing

To make C-Patrol comments a part of the underlying program, the pre-processor, C-Patroller, must first substitute virtual code for all the template and label calls that appear within insertions. After these transformations, insertions will consist entirely of standard C, so the C-Patroller may then insert the new code directly into the host program without further translation. For

`%%insert` directives, the C-Patrol comment is simply replaced by the new code. For `%%pre` directives, the insertion must be made so that the new code is executed just before any statements in the target function. Similarly, `%%post` directive insertions are made so that new code is executed just before any exit from the function. After insertions are complete, the program may be compiled and run as a normal C program, since the program then consists entirely of standard C code.

3 USES FOR C-PATROL

C-Patrol is primarily designed to allow developers to group sets of assertion definitions in a logical place, perhaps with data declarations, and direct the insertion of the assertions as required to perform run time checking. The C-Patrol system is quite flexible, and we expect to find numerous new applications for the tool.

The labeled code system was designed specifically to support data invariants. However, there is nothing application-specific about the concept. Calls to labeled code are simply a way of accessing blocks of code in an associative, database-like method. For example, test states can be organized with labeled code:

```
/*? %%label A:    x = f(3);
    %%label B:    x = f(20);
    %%label A,B:  y = g(8);   ?*/
```

These simple settings of `x` and `y` are only representative of potentially complex manipulations. Now we demonstrate calls that set up and use these test states:

```
/*? %%insert:
   switch(toggle) {
    case '1': %%call A;   break;
    case '2': %%call B;   break;
    case '3': %%ex-call A; y = g(43); break;}
?*/
```

The `toggle` variable can be controlled by the testing user to bring up these various states. The first two cases are direct calls to suites `A` and `B`. Both suites share the settings for variable `y`. In the third case, we use the `%%ex-call` to activate only part of the `A` setting, and then complete the setting explicitly. This example does not take advantage of the organizational power of the sub-field system, which can add further flexibility.

C-Patrol was designed initially as a system for implementing automated oracles and specifications. An oracle is a method of determining whether a program has performed according to specification [9]. Many organizations use human oracles — users determine from system and debug output whether program behavior is correct. C-Patrol can help automate aspects of the oracle process. We can use the C `assert` primitive to check conditions of the state of the program:

```
/*? %%insert:
      assert( <condition> );   ?*/
```

The `pre, post,` and `insert` directives support checking assertions at specific control flow points.

The mission of oracle code is different from the mission of the actual code or an executable specification. Oracle code is designed only to *recognize* correct and incorrect data, not *produce* such data. The choice of C as a virtual language means that the oracle is written in the concrete domain of actual structures used by the program.

C-Patrol provides a means for checking that data structures satisfy invariants. Objects can be linked to label identifiers, thus associating invariants to objects:

```
int X;    /*? %%label X: assert(X < 20); ?*/
```

Calls to identifier `X` can then be used to invoke this invariant. We can also link types to template blocks:

```
/* this is a type definition */
struct the_type { int a; int b; }

/*? %%template the_type(P):
assert(P.a < P.b);  ?*/
```

We can then create an object out of this type by assigning a label identifier to an instantiation of the template:

```
/* this is a variable declaration */
struct the_type the_val;

/*? %%bind the_val = the_type(the_val);
    %%label the_val:
        assert(the_val.a < 10);   ?*/
```

A new condition, `the_val.a < 10`, was added to the existing template condition, `the_val.a < the_val.b`. C-Patrol is thus capable of expressing one level of inheritance — special conditions can be added to an object of a type.

In C-Patrol actual implementation data structures must be accessed when expressing constraints. One of the objectives in future work is to provide a way of better shielding abstract specifications from the details of implementation. In conjunction with traditional methods of specification, such checking code can be used throughout the life cycle of a project to ensure that the original intent of the designer is satisfied.

4 ALTERNATIVES FOR MANAGING ASSERTION INSERTION

Our design was heavily influenced by the commercial sponsors of this research. We learned that our sponsors frequently use an object based programming style even though the implementation language is C. Because of the reliance on data abstraction, our design needed to include features for checking both data invariants and pre- and post-conditions. The task of enforcing data invariants in a language like C, which does not support data abstraction, led to a design using labeled code and templates. The C-Patrol system is not restricted to a particular application. However, the design is oriented toward specification and enforcement problems for data invariants.

Another important factor influencing C-Patrol design is the emphasis on industrial practicality over academic exercise. Restrictions in implementation manpower and user training time led to a design that emphasized simplicity whenever possible. The prototype system is quite basic; more ambitious features will not be included unless a clear need is discovered during prototype testing. A consequence of such simplicity is that little automatic checking is provided to protect users from dangerous operations. C-Patrol relies on a design that is as transparent as possible so that the user can either anticipate problems beforehand or quickly debug the ones that arise.

4.1 Virtual Code

A major issue in the C-Patrol design was the content of virtual code. Three major approaches were considered:

- Meta-Language: A common approach used in specification-oriented systems is to introduce a new meta-language that describes invariants at a more abstract level than is possible in C.

- Restricted C: We could modify or limit the C that can appear as virtual code. If the system is designed to passively check the program without modifying it, such enforcement can protect the program from accidental state modifications caused by insertions.

- Unrestricted C: This is the approach currently adopted by the C-Patrol design.

We discuss the merits and disadvantages of each of these approaches in detail.

4.1.1 Meta-Languages as Virtual Code

One common approach to expressing executable specifications or oracle constraints is to implement powerful, high level constructs, such as those found in VDM [5] or Z [3], in virtual code. The clear syntax and high level primitives of such languages allow the user to express complex requirements in a precise and abstract manner. Furthermore, a language can be designed that inherently protects the underlying program from the actions of virtual code.

User training overhead is one important reason for using C instead of meta-language virtual code. Although meta-languages provide greater expressive power, some transparency may be lost if users do not fully understand the actions of very high level primitives. Furthermore, users are naturally reluctant to devote the time necessary to develop the needed comprehension of such languages. By using the host language, C, the problem of misuse due to incomplete comprehension is minimized. There is also a gain in practicality — pre-compilers that implement high level primitives can be quite complex and cannot compete with the efficiency or reliability of proven C compilers.

Determining the type of high order primitives to be included is another problem. Most high level primitives are directed toward particular applications; the types of high level operations needed vary with paradigms of use. When used outside of its intended application, a language can become awkward to use. Furthermore, a computational argument posed by Hayes and Jones shows that there are classes of high level primitives that are impractical or impossible to implement [4]. We keep C-Patrol open to as many applications as possible rather than anticipate the primitives that will be of service to the user. Feedback from prototype testing may cause the addition of special primitives. However, there are a plethora of C libraries with specialized functions that may provide the high level power needed for most applications. The inclusion of such libraries can be hidden within C-Patrol comments.

4.1.2 Restricted C as Virtual Code

User can be protected from harmful virtual code by restricting the virtual code to a subset of C that guarantees a certain level of safety. The chief C subset that we considered was *read-only code* — code that is guaranteed to not modify state information. We could enforced this restriction in one of two ways. We could eliminate certain C constructs (such as assignment statements) from virtual code, or we could scan the code for destructive operations and warn the user.

There are several difficulties with using restricted C.

Eliminating modifying constructs (such as assignment statements) also eliminates computations that use and modify local computation variables, thus severely limiting the expressive capabilities of virtual code. Attempting to statically separate local variable manipulation from outside variable manipulation will either add a great deal of complexity (and unreliability) to the prototype, or will burden the user with tedious or unclear regulation. Furthermore, it may be difficult to identify which uses of outside variables modify them and which uses simply read them. Variable aliasing or renaming complicates this problem.

Perhaps the most important reason that no restrictions are placed on virtual code is that we are uncertain of how C-Patrol will be used. In some testing applications, for instance, the user may intentionally modify the state to produce certain debugging conditions. By opening virtual code to all possible C, we also open C-Patrol to all possible applications. Results from prototype testing will be critical for determining whether restrictions will be placed on virtual code.

4.1.3 Unrestricted C as Virtual Code

We have accepted the dangers of using unrestricted C as virtual code in exchange for simplicity and application independence. The C-Patroller is not affected by the content of user insertions — virtual code is treated simply as a block of text that is to be inserted into a program. This approach also gives us a certain level of language independence. Since the language in which virtual code is written is unimportant to the C-Patroller, large portions of the C-Patrol implementation should prove to be portable to other imperative languages, such as Fortran or Pascal.

4.2 Insertion Methods and Stability

Labeled code used to enforce data invariants must be invoked explicitly by the user. *Automatic* insertion of such invariants would better approximate the concept of an extended type; enforcement of these additional constraints would be implicit and could be viewed as a kind of rigorous type check. We are considering a limited form of automatic insertion for future versions of C-Patrol. However, problems relating to *stability* are the predominant force in shaping the current system of labeled code [8].

Stability is defined relative to an object, its invariant, and a point in program control flow. An unstable state is a program state where the object invariant is temporarily violated. Unstable states occur frequently during an operation on an object, where the object is being incrementally modified to a new state. In the following example, we assume the invariant `X.a < X.b` holds for object `X`:

```
X.a = X.b + 2;
X.b = X.a + 10;
```

The invariant is maintained in this operation. However, there is a temporary instability between the two C statements where `X.a` is less than `X.b`. Before an operation is complete, relationships between sub-fields of the object may be temporarily violated while fields are being updated. An attempt to check an invariant at such a point can result in a misleading error or, in the case of uninitialized data, aborted execution.

To help identify the problems related to unstable states, we have identified three types of read-only invariants based on their *access levels*:

1. *Constant*: the invariant does not relate the data to other objects or fields, as in `A.x < 10`.

2. *Internal*: the invariant relates fields from the same object to each other, as in `A.x < A.y`.

3. *External*: the invariant relates different objects to each other, as in `A.x < B.x`.

Invariants written at each of these access levels require different levels of stability. State stability and access levels thus have a direct effect on the safety of various insertion techniques.

The C-Patrol labeling system provides the user with a tool for identifying the access level of an invariant. By mapping label identifiers to objects, the user can specify the level of access by listing all objects or subfields referenced in the code's label:

```
/*? %%label A.x:       assert(A < 10);
    %%label A.x, A.y:  assert(A.x < A.y);
    %%label A.x, B.x:  assert(A.x < B.x);
?*/
```

Once identified, the access level desired by the insertion can be specified by the call directive:

```
%%ex-call A.x;   {* constant level only *}
%%ex-call A;     {* constant or internal *}
%%call A.x;      {* all levels *}
```

Another method of controlling access levels is to make such restrictions an inherent property of virtual code. This would involve the use of either restricted C or meta-language virtual code.

4.2.1 Statement Boundary Insertion

One possible automatic insertion method places invariants immediately after all statements that modify an object. Static implementations of this method encounter problems in discriminating statements that modify the object from statements that simply read it (i.e. separating L-value from R-value accesses). Further difficulties may occur due to aliasing; the object may be referenced indirectly through pointers, or local variables may temporarily assume the same name, thus confusing the insertion algorithm. One solution to these problems is to intercept references to the object in the symbol table; however, even this method is subject to problems of stability. Only constant access level invariants are guaranteed to be appropriate for such insertions. Identifying constant level invariants can be facilitated by correct labeling or by using a virtual language that inherently guarantees such a condition.

4.2.2 Function Boundary Insertion

Internal stability problems result from interrupting a procedure with a check. This problem can be circumvented by performing checks only at function boundaries – essentially, adding the checks to the operation's pre- and post-conditions. This is the approach favored by Eiffel [8] and A++ [2] where data invariants are enforced as an additional pre- and post-conditions for operations on objects. Special pre-condition exceptions are made for initialization routines.

The approach used in Eiffel and A++ is feasible because the underlying languages support object-oriented programming. With procedural languages like C, determining which functions are operations on a particular class of object is non-trivial. The problems encountered are similar to those that exist at the statement level; it is difficult to separate read-only usage from modifying usage and unexpected aliasing can either cause excessive or insufficient enforcement. Furthermore, the symbol table access solution is no longer trivial, since the execution of the check must be delayed until the next function boundary is recognized.

Stability can be a problem for function boundary insertions as well. External access level invariants may state relationships between different objects; after an operation on one object is complete, there may be a temporary violation of an inter-object condition until the other object is adjusted accordingly.

4.2.3 Explicit Insertion with Labeled Blocks

By relying on explicit invocation of invariants, C-Patrol defers the problem of identifying object operations and their access level to the user. It is thus the user's responsibility to locate all functions that modify an object. This explicit insertion approach can be aided by coding standards, used in many companies, that require that a list of modified objects be supplied in program documentation. The `%%call` directive can be used to emulate such documentation:

```
/*? %%pre:
       %%call A, B;
       {* just like many doc standards *}
?*/
```

The `%%pre` and `%%post` insertion directives allow convenient function boundary insertion. Invariant invocations can be placed within `%%pre` and `%%post` directives, thus yielding the benefits of function boundary enforcement. The problem of external access can be handled by the `%%ex-call` directive. Assuming the labels on blocks are reliable, the user can deny execution to invariants that contain accesses to objects whose state is uncertain.

```
/*? %%ex-call A;
    {* will not invoke blocks
       with other objects *}
?*/
```

The sub-field system can be used to provide this service at the internal access level.

4.3 Toward Extended Types

One of the objectives of C-Patrol is to provide the user with some form of an extended type. Essentially, the user should be able to annotate a type with additional checking code. Any time an object of that type is modified, the checking code is automatically executed, and warnings are generated if any conditions are violated.

The labeled code system does not have parameters, a crucial feature needed in extended type checking code. Without parameters, checking code must be directed toward a specific object rather that the group of objects of the same type. Template blocks were provided to alleviate this problem, as demonstrated in Section3.

In our initial efforts to provide parameters, we tried to incorporate the parameters directly into the labeled code system. However, we found it difficult to ensure that all parameters were bound before the virtual code is inserted into the program. There is no one-to-one relationship between label identifiers and the virtual code that the labels are attached to. Each code block contains a unique set of parameters to be bound; however, at the point of a call directive, the precise set of blocks that are being invoked cannot be determined when a particular identifier is used. Thus, it is not possible to determine which parameters need to be bound. Our current solution is to create a special structure, the template directive, that ensures a one-to-one mapping between the code block and the identifier that references it. We incorporate bound template blocks into the labeled code system, thus preserving some of the power of the labeled system while adding a system for parameterized code.

Another barrier to the implementation of extended types is the stability at the points of code insertion. If a type is defined relative to another value, then the resulting code has either an internal or external access level, allowing invocation in unstable states. The simplest solution to this problem is to restrict such code to constant access levels.

The final problem with extended types is type inference. In the presence of complex structures, pointers, and aliases, it can be difficult for the pre-compiler to recognize which variables belong to the type being enforced. A solution being considered for future versions of C-Patrol will force the user to syntactically specify which variables are of the type via regular expressions. The pre-processor will scan the program for occurrences of these expressions and insert code at the next statement boundary. The string that matched the expression would be used to instantiate the parameter in the type code. This proposed solution, called *key-jerk code*, still has many problems that must be resolved before it can be implemented.

5 RELATED WORK

We find a wide spectrum of related research areas including software specifications, testing, CASE tools, C tools, and object oriented programming. We provide an overview of related work that influenced the C-Patrol design.

Our original objective was to bring abstract concepts from the Prosper project into the highly pragmatic world of C [13, 1, 6]. Prosper is an experimental pre- and post-condition enforcement language designed for purely functional programs. With functional languages, side effects are not a factor.

One inspiration for C-Patrol work comes from Annotated Ada, or Anna [7]. Anna also uses comments to hide insertions. C-Patrol extends the Anna work by adding the labeled code system. However, C-Patrol does not provide the Anna primitives that make expressing constraints more intuitive. Executing checking code can be expensive, and one Anna implementation relies on concurrency to offload checking overhead [12]. The use of pure C for virtual code makes C-Patrol execution more efficient, reducing the need for such measures.

The labeled code system began as an attempt to imitate the methods of Eiffel [8]. Eiffel object invariants are inserted as additional pre- and post-conditions to all operations on the object. Such methods are difficult to execute in C due to the lack of language support in identifying the operations of an object. Eiffel also has a system for selectively activating insertions, a feature that will eventually have to be implemented in C-Patrol.

The object-oriented nature of C++ also simplified the task of the Annotated C++ project (A++), which seeks to do for C++ what Anna does for Ada [2]. A++ exploits the object-oriented nature of C++ to explicitly provide more advanced object-oriented concepts, such as encapsulation and inheritance. Such features may be the subject of future C-Patrol research.

The Anna project also inspired another cousin, APP, the Annotation Pre-Processor for C [10]. APP, like C-Patrol, was designed using a highly pragmatic philosophy. Unlike C-Patrol, APP has been operational for some time, although testing has been limited to relatively private experiments by the researcher. APP provides four primitives that function much like C-Patrol insertion directives. However, APP also provides specific constructs to reference the input values of variables. C-Patrol provides additional highly flexible capabilities to help users manage and place assertions. C-Patrol is designed so that it is quite possible that APP commands can be organized and inserted into code by the C-Patrol labeled code system. Such a hybrid system would provide utility beyond that of either tool.

Rubinfeld demonstrates a form of dual programming called *self-checking code* [11]. Systems like C-Patrol may be ideal for such applications, allowing users to shield their programs from the effects of checking code by hiding them in comments.

6 CONCLUSIONS

There is a real need for tools to manage the use of executable assertions for finding run time defects in software. Our tool, C-Patrol, inserts assertions, written in *virtual C*, into specified locations in C programs. A prototype C-Patrol system has been implemented, and the prototype is being evaluated at several industrial sites. Users have already reported that C-Patrol has helped to find defects that had previously gone undetected.

In developing C-Patrol, we evaluated alternatives for managing the assertion insertion process. Possible design alternatives concern:

- The choice of language used to express the assertions. Choices include (1) a meta-language such as Z or VDM [3, 5], (2) a restricted form of the implementation language, or (3) an unrestricted form of the implementation language. C-Patrol uses unrestricted C, which maximizes user control and eases training effort.

- The method used to insert the assertions. Assertions can be inserted manually into desired locations, or automatic insertion can be used. Problems related to *invariant stability* — the existence of program states where a data invariant is expected to be temporarily violated — limit the use of automated insertion. We use a flexible labeled code system that allows the definition of assertions at convenient locations, while providing users the ability to direct code insertions via labels. The labeled code system is a procedure-like invocation system that inserts code blocks based on their association to a series of label identifiers rather than through explicit names. Because insertion is under user control, stability problems can be avoided.

- Support for extended types to allow the checking of data invariants. C-Patrol provides generic-like templates to allow the definition of assertions with datatype declarations that can be instantiated with specific variables.

Future work will be heavily dictated by the results of on-going testing of the prototype. We are also planning a number of added features and refinements including support for selective activation of insertions, libraries of pre-defined assertions (in virtual C), support for label identifier scoping, and automated insertion through "Key-jerk" code. "Key-jerk" code is a method of invoking code blocks by association with regular expressions.

The C-Patrol design relies on a network of innovative constructs that balance simplicity, generality, and power. It has shown potential in a spectrum of applications far beyond those originally intended and quite possibly beyond those currently envisioned. Future development of C-Patrol will continue to emphasize the combination of abstract software engineering and industrial pragmatism that has already provided an extremely promising design.

REFERENCES

[1] J. Bieman and H. Yin. Designing for software testability using automated oracles. *International Test Conf.*, pages 900–907, 1992.

[2] M. Cline and D. Lea. The behavior of C++ classes. *Proc. Symp. on Object Oriented Programming Emphasizing Practical Applications*, 1990.

[3] I. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, London 1987.

[4] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pp. 330–338, November 1989.

[5] C. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, London 1986.

[6] J. Leszczylowski and J. Bieman. PROSPER, a language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[7] D. Luckham and F. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, pp 9–22, March 1985.

[8] B. Meyer. *Eiffel the Language.* Prentice Hall International, 1992.

[9] D. Richardson, S. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. *Proc. 14th Int. Conf. Software Engineering*, May 1992.

[10] D. Rosenblum. Toward a method of programming with assertions. *Proc. 14th Int. Conf. Software Engineering*, pp. 92–104, May 1992.

[11] R. Rubinfeld. A mathematical theory of self-checking, self-testing and self-correcting programs. Int. Computer Science Inst., October 1990. Technical Report TR-90-054.

[12] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, pp. 32–41, March 1993.

[13] H. Yin. Automatic enforcement of invariants: The implementation of prosper. MS Thesis, Colorado State University, 1991.