C-Patrol: Design and Usage

Hwei Yin and James M. Bieman

Technical Report CS-93-111

August 20, 1993

C-Patrol: Design and Usage

Hwei Yin James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
yin@cs.colostate.edu, bieman@cs.colostate.edu

Technical Report CS–93–111

July 28, 1993

Abstract

The C-patrol system is a simple but powerful CASE tool for C software systems. The heart of the proposed prototype is the *labeled code system*, a procedure-like mechanism that invokes blocks of code through a database-like associative system rather than through explicit procedure names. The C-patrol design resolves several difficult issues in enforcing object-oriented invariants in a language that offers little to no support for object-oriented programming. Applications of C-patrol include object-oriented programming, executable specifications and oracles, debugging, and test script generation. The highly general nature of of this utility makes compatibility with other tools and languages likely, implying that a wide variety of new applications may be uncovered during prototype testing.

1 Introduction

C-patrol is a CASE tool design with the virtues of both simplicity and power. C-patrol has potential applications in a wide range of areas, including executable oracles, specifications, testing, and debugging; however, the design was originally meant to aid object-oriented programming methods in C. C-patrol design concepts are unusually flexible in both implementation and application, and we anticipate a wide spectrum of uses well beyond what is currently envisioned. C-patrol design concepts are independent of C itself; thus, major portions of the implementation are likely to be portable to other languages. This independence also makes compatibility with other software tools likely. Like its inspiration, Anna [LvH85], C-patrol consists of a system of comments that can be

Address correspondence to Dr. James M. Bieman, Department of Computer Science, Colorado State University, Fort Collins, CO 80523. Voice: (303)491-7096, Fax: (303) 491-6639, Email: bieman@cs.colostate.edu

Research partially supported by the Colorado Advanced Software Institute (CASI) and Storage Technology 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.

textually converted into C code by a special pre-processor. C-patrol, however, is a much simpler system that derives its expressive power from a new procedure-like mechanism, the *labeled code system*, rather than from a large inventory of high-level constructs. This emphasis on simplicity is a result of a highly pragmatic approach designed to reduce implementation and user training time. A prototype is currently being 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.

We begin by describing the C-patrol prototype and its major feature, the labeled code system. After surveying potential uses for the system, we discuss the critical design decisions and issues that were instrumental in shaping the current system. We conclude by surveying related work and discuss new enhancements that may be implemented in future versions.

2 C-patrol Description

The following description is only intended to cover the main concepts behind C-patrol – a complete reference manual is beyond the scope of this document. In the interest of clarity, we will defer detailed discussion about the reasons behind design decisions until later sections.

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 special pre-processor that translates virtual code into regular C and inserts it based on a set of instructions called *directives.* This augmented program may then be compiled and run as normal C.

In the current design, virtual code is nothing more than regular C: there is no special meta-language that has to be learned, and there are no restrictions placed on the virtual code written. It is the user's responsibility to ensure that virtual code does not produce any undesired side-effects. C-patrol comments thus consist mainly of normal C interspersed with C-patrol directives.

At this point it is important to mention that *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. 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.

Finally, it is important to remember that the C-patroller is a pre-processor designed only to textually insert code, not compile it. This becomes a factor when we discuss procedure-like features such as *labeled code* and *templates.*

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

Much of the power of C-patrol comes from its system of *labeled code*. Much like the more familiar macro, labeled code consists of a block of virtual code that is inserted when invoked by a `%%call` directive. Unlike more traditional 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");
?*/
```

Note that, unlike procedure or macro names, label identifiers are not unique to blocks of code. Also note that 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
```

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

To clarify the actions of labeled code directives, it is useful to view the system 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 greater flexibility than labeled code through the use of parameters. Their declaration is almost identical to the declaration of a traditional procedure or macro:

```
/*? %%template printme($a, $b);
       printf("%i $a", $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 strings `"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("%i 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 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 target function are. 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 it then consists entirely of standard C code.

3 Uses for C-Patrol

One of the most important aspects of C-patrol is its flexibility. The ability to hide code within comments can be used to insert debugging code without interfering with system performance. "Forbidden path" checks, for instance, can be written in C-patrol to handle contingencies that shouldn't occur in normal program operation. However, it is in the labeled code system that C-patrol really shows potential for new applications.

Although the labeled code system was designed specifically to implement object-oriented invariants, 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. In the following demonstration, note how test states can be organized with labeled code:

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

We realize, of course, that 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 controlled by the testing user to bring up these various states. The first two cases are direct calls to suites `A` and `B`. Note how 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. Finally, we note that this example does not even take advantage of the organizational power in the sub-field system.

The power of C-patrol undoubtably leads to many applications; however, C-patrol was designed as a system for implementing automated oracles and specifications, so it is important to consider it in that context.

An oracle is a method of determining whether a program has performed according to specification [RAO92]. For most systems, the oracle is human: users determine from system and debug output whether the program worked correctly. C-patrol is a tool that can help automate aspects of this process. We can use the C `assert` primitive to check conditions on the state of the program:

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

The `pre`, `post`, and `insert` directives give us power is specifying where in control flow the condition is to be checked.

Note that 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. A full discussion of the advantages and disadvantages of using C as a virtual language appears in a later section.

Checking code can also be used as a method of specification. Efforts such as VDM [Jon86] and Z [Hay87] use pre- and post-condition checks to specify the actions of functions. C-patrol goes a step farther by providing invariants that can be applied to data structures. 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 used to invoke this invariant.

We can also link types to template blocks:

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

/*? %%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:

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

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

Note how 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.

The current limitation to specifications in C-patrol is that that virtual blocks are not sufficiently parameterized to allow for any abstraction: 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.

Prototype testing will be a crucial factor in determining what the chief uses of C-patrol will be. Our software engineering background is also a bias, blinding us to potential applications in other areas. We will rely on the spontaneous ingenuity of the user fulfilling his or her immediate needs as a guide towards understanding the full power of this system.

4 Design Decisions in C-Patrol

The work leading to C-patrol was primarily centered around invariant enforcement for functional languages; however, the system developed appears to have strong potential for applications not anticipated by the initial design. Examination of industrial code provided by our client revealed a heavily object-oriented programming style, which, due to the reliance on side effects, was not appropriate for purely pre- and post-condition analysis. The formidable task of enforcing object-oriented invariants in a language that has no support for such methods led to the current system of labeled code and templates. Thus, it is important to remember that although the C-patrol system is not restricted to a particular application, its design was oriented toward specification and enforcement problems for object-oriented 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 currently being implemented is quite basic: more ambitious features will not be included unless a clear indication is received from users during 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**
  An approach that occurs frequently 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**
  Another important approach is to 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.

The merits and disadvantages of each of these approaches will be discussed 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 [Jon86] or Z [Hay87], in virtual code. The clear syntax and high level primitives of such languages allow the user to express complex requirements in a clear and abstract manner. Furthermore, a language can be designed that inherently protects the underlying program from the actions of virtual code. The intent of such code thus becomes more apparent to the user as primitives come closer to a documentation level of abstraction.

One important reason for using C instead of meta-language virtual code was user training overhead. 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 will be 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. In addition to these benefits, there is a gain in practicality: pre-compilers that implement high level primitives can be quite complex and cannot compete in the efficiency or reliability of proven C compilers.

Another problem lies in determining the type of high order primitives to be included. Most high level primitives are specialized toward particular applications: the types of high level operations needed vary with paradigms of use. When used outside of its intended application, the language becomes awkward to use. Furthermore, a computational argument posed by Hayes and Jones [HJ89] shows that there are classes of high level primitives that are impractical or impossible to implement. We chose to open C-patrol to as many applications and uses as possible rather than attempt to anticipate the primitives that will be of service to the user. It is possible that 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

One approach considered to protect the user from harmful virtual code was to restrict the code to a subset of C that guarantees that a certain level of safety. The chief C subset we considered was *read-only code*: code that is guaranteed to not modify state information. This condition would be enforced in one of two ways: eliminate certain C constructs (such as assignment statements) from virtual code, or scan the code for destructive operations and warn the user.

There are several difficulties with using restricted C in the prototype. 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 and which uses simply read, especially in the presence of variable aliasing or renaming.

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

As described above, there are disadvantages to using un-modified C as virtual code. In general, we have accepted these dangers in exchange for simplicity and application independence. Essentially, the C-patroller is unaware of the content of user insertions: virtual code is treated simply as a block of text that is to 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 will be portable to other imperative languages, such as Fortran or Pascal.

4.2 Insertion Methods and Stability

One strong objection against using labeled code in enforcing object-oriented invariants is that all labeled code must be invoked explicitly by the user. *Automatic* execution 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. A limited form of automatic insertion is currently being considered (see **Future Work**); however, problems relating to *stability* [Mey92] were the predominant force in shaping the current system of labeled code.

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's invariant is temporarily violated. These 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, between the two C statements, there is a temporary instability where `X.a` is less than `X.b`. Before an operation is complete, it is possible that 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 sub-fields 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: the merits and difficulties of such an approach was discussed previously.

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. *Key-jerk* code is a potential statement level insertion system: it relies on constant access levels to ensure the safety of insertions. Key-jerk code is further discussed in **Future Work**.

4.2.2 Function Boundary Insertion

Internal stability problems result from interrupting an object's operation with a check before the procedure is complete. 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 [Mey92] and A++ [CL90]: any function that operates on the object has the object's invariants enforced as an additional pre- and post-condition constraint, with special pre-condition exceptions made for initialization routines.

The main factor facilitating this approach for Eiffel and A++ is that the underlying languages support object-oriented programming. With imperative languages like C, determining which functions are operations on an 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 be complete in locating all functions that modify an object. This seems reasonable since many industrial standards require that a list of objects modified be supplied in function 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 w/ 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 object-oriented 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, producing warnings if any conditions have been 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 the **Uses for C-Patrol** section.

Initial efforts toward providing parameters attempted to incorporate them directly into the labeled code system; however, it proved difficult to ensure that all parameters were bound by the time the virtual code was inserted into the program. This is because there is no one-to-one relationship between label identifiers and the virtual code they are attached to. This point is made more clear when we note that each code block contains a unique set of parameters to be bound; however, call directives do not know which of these blocks are being invoked when a particular identifier is used, so there is no way of knowing 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. Fortunately, we were able to 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 impeding the implementation of extended types is the stability at the points of code insertion. If the type is defined relative to another value, then the resulting code is either internal or external in access level, which can lead to invocation in unstable states. The simplest solution to this problem is to restrict such code to constant access levels; as demonstrated earlier, this is facilitated by the `%%ex-call` feature.

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 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 system, called *key-jerk code*, is described in the **Future Works** system.

5 Related Work

C-patrol research covers a wide spectrum of areas: software specifications, testing, CASE tools, C tools, and object oriented programming to name a few. Thus, it is difficult to identify sources for related work. We briefly discuss work that we are aware of and work that influenced C-patrol design.

Our original objective was to bring abstract concepts from an earlier project, Prosper [Yin91] [BY92] [LB89], into the highly pragmatic world of C. Prosper is an experimental pre- and post-condition enforcement language designed for a purely functional language. With functional languages, side effects are not a factor; however, the industrial C code we analyzed showed a heavily object-oriented style of programming, prompting us to create the labeled code system.

One inspiration for C-patrol work comes from Annotated Ada, or Anna [LvH85]. It is from Anna that the method of using comments to hide insertions was derived. C-patrol extends Anna work by adding the labeled code system; however, it does not provide the Anna primitives that make expressing constraints more intuitive (see **Design Decisions in C-Patrol**). Executing checking code can be expensive, and one Anna implementation [SM93] relies on concurrency to offload checking overhead. 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 [Mey92]. 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++ [CL90], which seeks to do for C++ what Anna does for Ada. 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.

Anna also inspired another cousin, APP, or the Annotation Pre-Processor for C [Ros92]. APP, like all Anna cousins, closely echos C-patrol its 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 primitives that function much like C-patrol insertion directives; however, APP also provides more advanced features, such as the ability to recall old variable values for comparison and an advanced assertion reporting facility, that are not yet provided by C-patrol. C-patrol flexibility makes is quite possible that the features of tools such as APP can be organized and inserted into code by the C-patrol labeled code system, resulting in a combined power beyond that of either tool.

In his PhD thesis, Rubinfeld [Rub90] demonstrated a form of parallel programming called *self-checking code*. 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 Future Work

There are many refinements and features envisioned for C-patrol that will not be provided in the prototype. The following section discusses items in consideration for future development and research.

6.1 Multiple File Organization

The prototype version of C-patrol is not yet capable of handling complex, multi-file systems with interlocking C-patrol comments appearing within different files. In the current system, all C-patrol comments must appear within the same file. Multi-file system capability will be essential to any industrial application of C-patrol.

6.2 Selective Activation of Insertions

The C-patroller activates *all* insertions when invoked. Users may wish to block some insertions from activation without altering the program. Currently, C-patrol does not provide any system for doing this without altering program text.

An approach under consideration labels insertions with the same system used for labeled code. When the pre-compiler is invoked, a call directive can be issued at the operating systems level to select which insertions will be pre-compiled and which will be ignored. The labeled code system provides the power of hierarchical organization and multiple identifiers without creating excessive complexity; moreover, using the same system twice reduces training time for users.

6.3 C-patrol Specifications

One of the main goals of the C-patrol project is to express specifications that can exist and execute throughout the software development cycle. The choice of C as a virtual language, however, creates the problem of implementation-dependent specifications. One approach to better disguising the underlying implementation is to provide parameterized virtual code that allows all implementation-dependent structures to be aliased by an abstract identifier. A system for mapping such aliases to C constructs has yet to be defined.

6.4 Advanced Primitives

C-patrol does not provide any features to make the virtual C code more abstract or easier to use. All desired functions of the virtual code must be coded explicitly by the user. Tools such as A++ [CL90] and APP [Ros92] demonstrate that such features may be necessary to make C-patrol practical for industrial use; however, the flexibility of C-patrol makes determining the sort of primitives required difficult because of the unpredicatably wide spectrum of future applications. Furthermore, C already provides a rich body of library functions that perform advanced services, such as the `assert` function for checking code. We will therefore not provide such features until clear needs and applications develop in prototype testing.

6.5 Scoping of Label Identifiers

In the C-patrol prototype, it is the user's responsibility to ensure that label identifiers are unique. This may be difficult to ensure in a complex multi-file and multi-user system, where identifiers from other files may coincidentally conflict with local identifiers. This may result in the inclusion of labeled code not relevant to the insertion.

A structured programming approach to this problem is to introduce some sort of scoping feature to distinguish local identifiers from identifiers from other areas of code. The sub-field system allows the user to explicitly arrange label identifiers into a hierarchy, but it is a strong possibility that a more implicit system will eventually be required.

6.6 Automated Insertion

C-patrol code insertion techniques rely on explicit invocation and little to no automation. Although this allows the user to better control and protect his or her code from spurious insertions, it also increases the user workload per insertion. A new insertion technique, tentatively named *key-jerk insertion*, provides a more implicit form of virtual code insertion.

Instead of a procedure name, key-jerk virtual blocks are associated with a regular expression or *key*. Whenever the target regular expression appears within the program text, it is treated as a call to the virtual code. The block is then inserted at the next statement boundary. The key may be bound to a parameter, as in the following:

```
/*? %%key-jerk $X = my_array[*]:
        assert($X < 20);   ?*/
```

We assume that the asterisk in the key is a wild card; thus, this code fragment would be invoked by the appearance of a token such as `my_array[512]`. At the next statement boundary, C-patrol would insert the expression:

```
   assert(my_array[512] < 20);
```

Note that the key-jerk definition allows for only one parameter. This will encourage the user to limit assertions to the constant access-level, making stable insertions a great deal more likely. For other problems, the key-jerk method relies on its intuitive nature to guide the user in correct usage.

There are many implementation details a problems associated with key-jerk code that have yet to be resolved. Clearly, there will be a need for some sort of scoping mechanism so that the target area for these insertions can be limited to a manageable portion of the program. However, this method, with further development, appears to have strong potential for increasing the utility of C-patrol.

7 Conclusions and Summary

C-patrol is a code insertion system that offers powerful features without sacrificing practicality. This is achieved by a design emphasis on simplicity. A simple system is a system that is easily comprehended by the user, which in turn allows a design that can rely on the user for self-regulation rather than implement complex constraints and monitors. This design philosophy results in a prototype that should be relatively simple to implement.

The central feature of the C-patrol prototype is the labeled code system. This feature allows the specification and enforcement of object-oriented invariants while effectively dealing with problems in efficiency, implementation, and state stability. The labeled code system is a procedure-like invocation system that calls code blocks based on their association to a series of label identifiers rather than explicit names. This powerful feature is application independent, and has a variety of applications beyond its object oriented objective. The concept is also language independent. C-patrol features can just as easily be implemented in other imperative languages, such as Pascal or Fortran. This broad potential makes it difficult to predict what sort of additional capability will be required, resulting in a "wait-and-see" approach to further development.

Future work will be heavily dictated by prototype test results; however, the powerful features and refinements planned promise to strongly enhance the capabilities of C-patrol. "Key-jerk" code is one of the most intriguing new ideas: it is a method of invoking code blocks by association with regular expressions.

The C-patrol design thus 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

[BY92] James M. Bieman and Hwei Yin. Designing for software testability using automated oracles. *International Test Conference*, pages 900–907, 1992.

[CL90] Marshall P. Cline and Doug Lea. The behavior of C++ classes. *Proc. Symp. on Object Oriented Programming Emphasizing Practical Applications, Marist College*, 1990.

[Hay87] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International, Cambridge, London, 1987.

[HJ89] Ian Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[Jon86] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, Cambridge, London, 1986.

[LB89] Jacek Leszczylowski and James M. Bieman. PROSPER, a language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[LvH85] Donald C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, pages 9–22, March 1985.

[Mey92] Bertrand Meyer. *Eiffel the Language*. Prentice Hall Intl., 1992.

[RAO92] Debra J. Richardson, Stephanie L. Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. *14th Intl. Conf. on Software Engineering, Melbourne, Australia*, May 1992.

[Ros92] David S. Rosenblum. Toward a method of programming with assertions. *Proc. 14th Intl. Conf. on Software Engineering, Melbourne, Australia*, pages 92–104, May 1992.

[Rub90] Ronitt Rubinfeld. A mathematical theory of self-checking, self-testing and self-correcting programs. Intl. Computer Science Inst., October 1990. Technical Report TR-90-054.

[SM93] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, pages 32–41, March 1993.

[Yin91] H. Yin. Automatic enforcement of invariants: The implementation of prosper. Colorado State University, 1991.