Towards a Method of Programming with Assertions

David S. Rosenblum

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

Embedded assertions have long been recognized as a potentially powerful tool for automatic runtime detection of software faults during debugging, testing and maintenance. Yet despite the richness of the notations and the maturity of the techniques and tools that have been developed for programming with assertions, assertions are a development tool that has seen little widespread use in practice. The main reasons seem to be that (1) previous assertion processing tools did not integrate easily with existing programming environments, and (2) it is not well understood what kinds of assertions are most effective at detecting software faults. This paper describes experience using an assertion processing tool that was built to address the concerns of ease-of-use and effectiveness. The tool is called APP, an Annotation Pre-Processor for C programs developed in UNIX-based development environments. APP has been used to develop a number of software systems over the past three years. Based on this experience, the paper presents a classification of the assertions that were most effective at detecting faults. While the assertions that are described guard against many common kinds of faults and errors, the very commonness of such faults demonstrates the need for an explicit, high-level, automatically checkable specification of required behavior. It is hoped that the classification presented in this paper will prove to be a useful first step in developing a method of programming with assertions.

Keywords: APP, assertions, C, consistency checking, formal specifications, formal methods, programming environments, runtime checking, software faults.

1 Introduction

Assertions are formal constraints on software system behavior that are commonly written as annotations of a source text. The primary goal in writing assertions is to specify *what* a system is supposed to do rather than *how* it is to do it. The idea of using embedded assertions as an aid to software development is not new. Indeed, 25 years ago Floyd demonstrated the need for loop assertions in mechanical verification of programs [Flo67]. In addition to their use in formal verification, assertions have also long been recognized as a potentially powerful tool for automatic runtime detection of software faults during debugging, testing and maintenance. As long ago as the 1975 International Conference on Reliable Software, several authors described systems for deriving runtime consistency checks from simple assertions [SF75, BMU75, YC75]. For instance, in Stucki and Foshee's approach, the assertions were written as annotations of a FORTRAN source text, and a preprocessor was then used to convert the annotations to embedded self-checks that were invoked at appropriate places at runtime.

Today, assertion features are available in many high-level formal specification languages, as well as some newer programming languages (such as Turing [HC88] and Eiffel [Mey88]). Such languages can be used to specify system behavior at the requirements and/or design levels. Many such languages are suitable for use as annotation languages, not only for generating runtime consistency checks, but also for static analysis of semantic consistency as in the *Inscape* environment [Per89]. These uses of high-level formal specifications offer a practical alternative to mechanical proof of correctness.

Much of the recent research on automatic derivation of runtime consistency checks from assertions is exemplified by the work on *Anna* (ANNotated Ada), a high-level specification language for Ada [LvH85]. This work includes (1) a method of generating consistency checks from annotations on types, variables, subprograms and exceptions [SRN85, SR86]; (2) a method that uses in-

cremental theorem proving to check algebraic specifications [San89, San91]; (3) a method of generating consistency checks that run in parallel with respect to the execution of the underlying system [RSL86]; and (4) a method of constructing large software systems based on algebraic specification of system modules [Luc90].

Yet despite the richness of the notations and the maturity of the techniques and tools for programming with assertions, assertions are a development tool that has seen little widespread use in practice. There appear to be two reasons for this state of affairs:

1. The tools that have been developed to support programming with assertions fail to meet the needs of the "average" software developer. They do not work well in conjunction with existing development tools, nor do they allow suitable flexibility in customizing assertion checks and in enabling or disabling checking at runtime.

2. It is not yet well understood what kinds of assertions are most effective at detecting faults. This is due in part to a lack of extensive tutorials and case studies that describe experience using assertions to build real systems. Consequently, most software developers have little idea of what information needs to be specified in assertions.

Effectiveness of assertions is an especially important issue, as demonstrated by the wide variance in the fault detection capabilities of the self-checks that were written by participants in a case study conducted by Leveson et al. [LCKS90].

To address these concerns and begin making programming with assertions a practical software development method, I have been developing and using an assertion processing system called APP, an Annotation PreProcessor for C programs developed in UNIX®-based programming environments. APP has been designed to be easily integrated with other UNIX development tools. In particular, APP was designed as a replacement for the standard preprocessor pass of C compilers, making the process of creating and running self-checking programs as simple as building unannotated C programs. Furthermore, APP provides complete flexibility in specifying how violated assertions are handled at runtime and how much or how little checking is to be performed each time a self-checking program is executed. APP does not require complete specifications for its correct operation, and the assertions one writes for APP typically are not complete specifications in any formal sense.

An initial prototype of APP was completed three years ago. Since then, I have applied APP to the development of several systems, each comprising around 10-20,000 lines of code. The assertion checks that were applied during the development of these systems automatically revealed a number of serious faults, and the diagnostic information they provided was often sufficient to quickly isolate and remove the faults. Based on this experience, it is possible to classify the kinds of assertions that were most effective at detecting faults.

The paper begins with a brief description of APP. The paper then presents a classification of assertions that is based on a retrospective analysis of the systems I developed with APP and the faults that were detected by the generated checks. The paper concludes with an analysis of the effectiveness of these assertions at detecting faults, and a discussion of how programming with assertions can be used as a component of a more comprehensive formal method that applies across the complete development life cycle. It is hoped that the classification presented in this paper will prove to be a useful first step in developing a method of programming with assertions.

2 APP

In Perry and Evangelist's empirical study, it was shown that most software faults are interface faults [PE85, PE87]. Hence, APP was initially designed to process assertions on function interfaces, as well as assertions in function bodies.[1] APP also supports a number of facilities for specifying the response to a failed assertion check and controlling the amount of checking that is performed at runtime.

2.1 Assertion Constructs

APP recognizes assertions that appear as annotations of C source text. In particular, the assertions are written inside comment regions using the extended comment indicators /*@ ... @*/. Informal comments can be written in an assertion region by writing each comment between the delimiter // and the end of the line.[2]

Each APP assertion specifies a constraint that applies to some state of a computation. The constraint is specified using C's expression language, with the C convention that an expression evaluating to zero is false, while a non-zero expression is true. APP recognizes two

®UNIX is a registered trademark of UNIX System Laboratories, Inc.

[1] In keeping with the terminology of C, this paper uses the generic term "function" to refer to subprograms. Whereas other languages distinguish subprograms having return values from those that do not (e.g., "functions" and "procedures", respectively, in Ada), C uses the term "function" to refer to both. A function whose return type is **void** returns no result to its caller, while a non-**void** function does return a result.

[2] The syntax of informal comments in assertion regions is the same as the comment syntax of C++.

```
int square_root(x)
int x;
/*@
    assume x >= 0;
    return y where y >= 0;
    return y where y*y <= x
                &&  x < (y+1)*(y+1);
@*/
{
    ...
}
```

Figure 1: Specification of Function **square_root**.

enhancements to the C expression language within assertion regions. First, the operator **in** can be used to indicate that an expression is to be evaluated in the entry state of the function that encloses the expression. Second, bounded quantifiers can be specified using a syntax that is similar to C's **for**-loop syntax. These enhancements are illustrated below.

APP recognizes four assertion constructs, each indicated by a different keyword:

- **assume**—specifies a precondition on a function;
- **promise**—specifies a postcondition on a function;
- **return**—specifies a constraint on the return value of a function; and
- **assert**—specifies a constraint on an intermediate state of a function body.

The first three kinds of assertions are associated syntactically with function interface declarations, while the last kind is associated syntactically with statements in function bodies. The **assert** construct corresponds to the **assert** macro found in many C implementations, in the sense that it constrains only the state of the program at the place of the **assert**.

To briefly illustrate these four constructs, consider first a function called **square_root** that returns the greatest positive integer less than or equal to the square root of its integer argument. Such a function can be specified as shown in Figure 1. The first assertion is a precondition of **square_root**, as indicated by the keyword **assume**. It states that the implementation of the function assumes it is given a non-negative argument; if this precondition is not satisfied at runtime, nothing can be guaranteed about the behavior of the function. The remaining two assertions are constraints on the return value of **square_root**, as indicated by the keyword **return**. Each return constraint declares a local variable (called **y** in the return constraints of this example) that is used to refer to the return value of the function

```
void swap(x,y)
int* x;
int* y;
/*@
    assume x && y && x != y;
    promise *x == in *y;
    promise *y == in *x;
@*/
{
    *x = *x + *y;
    *y = *x - *y;
    /*@
        assert *y == in *x;
    @*/
    *x = *x - *y;
}
```

Figure 2: Specification of Function **swap**.

within the constraint. The first return constraint states that the function returns positive roots. The second one states the required relationship between the argument and the return value. It is of course possible to conjoin these two **return** constraints into a single one; however, it is often useful to separate constraints not only for the sake of clarity, but especially when using APP's severity level and violation action features (as described below). Note that all of these assertions merely state what the function does, not how it does it.

Consider next a function called **swap** that swaps two integers without using a temporary variable. The function takes as arguments a pointer to each of the two integers, and it performs the swap through the pointers. The function can be specified and implemented as shown in Figure 2. The assumption states the precondition that the pointers **x** and **y** be non-null (and thus evaluate to true) and not equal to each other. The two postconditions, indicated by the keyword **promise**, use the operator **in** to relate the values of the integers upon exit from the function to their values upon entry. In particular, the first promise states that the exit value of the integer pointed to by **x** should equal the value pointed to by **y** upon entry, while the second promise states the reverse. The assertion in the body of **swap**, indicated by the keyword **assert**, states an intermediate constraint on the integers.

2.2 Violation Actions, Predefined Macros and Severity Levels

When APP converts an assertion to a runtime check, it generates code that is executed whenever the assertion is violated at runtime. By default, this code simply prints out a diagnostic message, such as the following

```
promise *x == in *y
   {
      printf("out *x == %d, out *y == %d\n",
             *x, *y);
   }
```

Figure 3: Violation Action for Promise of Function **swap**.

```
promise *x == in *y
   {
      printf("%s invalid:  file %s, ",
             __ANNONAME__, __FILE__);
      printf("line %d, function %s:\n",
             __ANNOLINE__, __FUNCTION__);
      printf("out *x == %d, out *y == %d\n",
             *x, *y);
   }
```

Figure 4: Enhancement of Violation Action of Figure 3.

for the first promise of function **swap**:

```
promise invalid:  file swap.c, line 6,
                  function swap
```

This default response provides a minimal amount of information to help isolate the fault that the failed check reveals. However, the response to a violated assertion can be customized to provide diagnostic information that is unique to the context of the assertion. This customization is accomplished by attaching a *violation action* to the assertion, written in C.

For instance, in order to determine what argument values cause the first promise of **swap** to be violated, the promise can be supplied with a violation action as shown in Figure 3 (using C's library function **printf** for formatted output). Using some preprocessor macros that are predefined by APP, this violation action can be enhanced as shown in Figure 4 to print out the same information printed out by the default violation action. The macro **__ANNONAME__** expands to the keyword of the enclosing assertion. The macro **__FILE__** expands to the name of the source file in which the enclosing assertion is specified. The macro **__ANNOLINE__** expands to the starting line number of the enclosing assertion. The macro **__FUNCTION__** expands to the name of the function in which the assertion is specified.

In addition to violation actions, APP supports the specification of an optional severity level with each assertion, with 1 being the default and indicating the highest severity. A severity level indicates the relative importance of the constraint specified within an assertion and determines whether or not the assertion will

```
1:  assume x >= 0;
2:  return y where y >= 0;
1:  return y where y*y <= x
                 &&  x < (y+1)*(y+1);
```

Figure 5: Severity Levels for Assertions of Function **square_root**.

be checked at runtime. Severity levels can be used to control the amount of assertion checking that is performed at runtime without recompiling the program to add or remove checks. For example, the assertions on **square_root** can be given severity levels as shown in Figure 5. Under level 1 checking at runtime, only the assumption and the second return constraint would be checked. If one of these assertions were violated at runtime, it might then be desirable to re-execute the program under level 2 checking, in order to additionally enable checking of the first return constraint and obtain more information about the cause of the assertion violation. Level 0 checking disables all checking at runtime. Severity levels are useful for implementing the "two-dimensional pinpointing" method of debugging described by Luckham, Sankar and Takahashi [LST91]. The mechanism for controlling the checking level at runtime is described below.

The macro **__DEFAULTACTION__** expands to the default violation action, while the macro **__DEFAULTLEVEL__** expands to the default severity level. Both of these macros can be redefined to alter the default processing of APP.

2.3 Generating and Running Self-Checking Programs

APP has the same command-line interface as *cpp*, the standard preprocessor pass of *cc*, the C compiler.[3] Hence, to compile an annotated C source file, APP is simply invoked through *cc* by using appropriate command-line options that tell *cc* to use APP as its preprocessor pass; such options are a standard feature of nearly every C compiler. Furthermore, standard build tools such as *make* [Fel79] and *nmake* [Fow90] can be used to build executable self-checking programs, with only slight modifications to existing *makefiles* or build scripts. These processing techniques are illustrated in Figure 6. This method of integrating assertion processing with standard C development tools greatly simplifies the generation of self-checking programs and requires almost no change to one's customary use of UNIX and C

[3] In particular, APP accepts the macro definition options **-D** and **-U** and the interface file directory option **-I**, and it performs all of the macro preprocessing of *cpp* in addition to its assertion processing function.

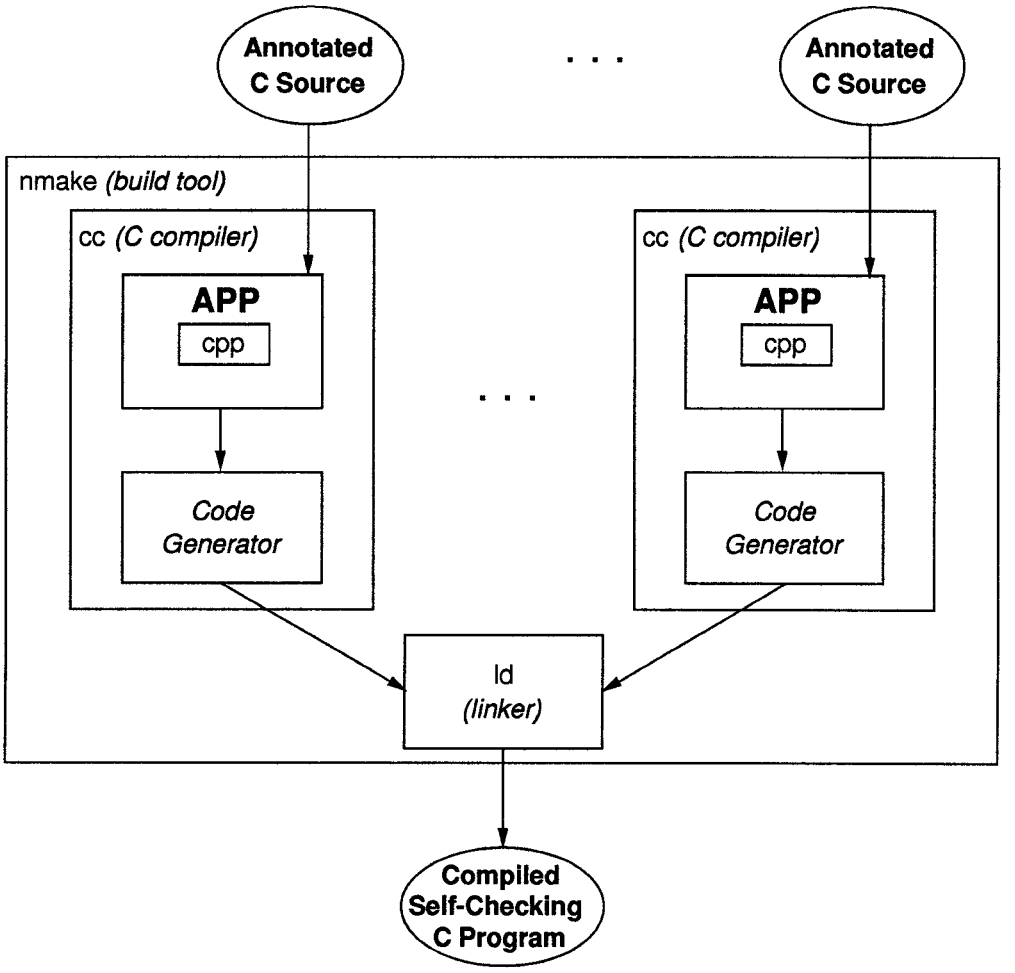

Figure 6: Generating Self-Checking C Programs with APP.

programming environments.

Once a self-checking program has been created it can be executed, with checking performed according to the severity level specified in the UNIX shell environment variable **APP_OPTIONS** (or at the default level if the environment variable is undefined).

3 A Classification of Assertions

I have been using APP for three years in the development of several software development tools, including APP itself. The assertions I have written have proven quite effective at discovering faults. Indeed, the effort of constructing the assertions has repeatedly paid off in quick, automatic detection and isolation of faults that would have otherwise consumed as much as several hours of effort using more primitive debugging tools such as symbolic debuggers and core dumps. Not only have the assertions provided a powerful fault detection capability, but the process of writing the assertions in the first place has resulted in much more careful development of implementation components.

Based on this experience, it would now be fruitful to stop and examine the systems I have built and characterize the kinds of assertions that were most effective in uncovering faults. The categories of assertions that I describe in this section guard against many common kinds of faults and errors. Yet the very commonness of such faults demonstrates the need for an explicit, high-level, automatically checkable specification of required behavior.

The examples provided for each category are abstracted from the actual assertions I have written. A few of the assertions are used to overcome inherent weaknesses in the type system of the C programming

language; in certain cases such assertions would not be needed in programs written in languages that support a strong type model, such as Ada. However, most of the assertions express constraints that are too complex to express in the type or data model of common programming languages. For instance, programming languages rarely, if ever, provide features for explicit specification of data consistency at the level of a function interface.

3.1 Specification of Function Interfaces

The primary goal of specifying a function interface is to ensure that the arguments, return value and global state are valid with respect to the intended behavior of the function. The common characteristic of all function interface constraints is that they are stated independently of any implementation for the function. That is, they describe function behavior at the level of abstraction seen by the callers of the function. The constraints described in this section are special forms of traditional preconditions and postconditions.

3.1.1 Consistency Between Arguments

Function arguments are often interdependent, even though such mutual dependencies cannot be specified directly in the programming language. Assertions can be used to specify mutual consistency constraints. For instance, consider a language processing system that uses a function called **store_token** to store unique copies of the tokens found in an input stream. As shown in Figure 7, the function takes as arguments an enumeration value specifying the kind of token and a pointer to the token string. The assumption checks that the syntax of the token is consistent with the value of argument **kind**: If the token is an identifier, its first character (i.e., the 0th component of the character array pointed to by **token**) should be a lower case letter.[4] If the token is a number, it should begin with a digit. And if the token is a string, it should begin with the double-quote character.

3.1.2 Dependency of Return Value on Arguments

Assertions can be used to state important aspects of the relationship between the return value of a function (or the values of its reference arguments upon exit) and the function's arguments upon entry. The second return constraint of function **square_root** shown in Figure 1 and the promises of function **swap** shown in Figure 2 illustrate this kind of assertion.

[4] C arrays are always indexed starting at 0.

```
enum Token_Kind { identifier, number,
                  string };

void store_token(kind, token)
enum Token_Kind kind;
char* token;
/*@
  assume (kind == identifier
      &&  token[0] >= 'a'
      &&  token[0] <= 'z')
     ||  (kind == number
      &&  token[0] >= '0'
      &&  token[0] <= '9')
     ||  (kind == string
      &&  token[0] == '"');
@*/
{
  ...
}
```

Figure 7: Specifying Consistency Between Function Arguments.

```
void delete_name(name)
char* name;
/*@
    assume hashget(symbols, name);
    promise !hashget(symbols, name);
@*/
{
    ...
}
```

Figure 8: Specifying the Effect on the Global State.

3.1.3 Effect on Global State

Functions in procedural languages often have side effects. Assertions can be used to specify how a function changes the global program state. For instance, consider a language processing system that uses a routine called **delete_name** to remove entries from a global symbol table called **symbols**. The specification of **delete_name** is shown in Figure 8; assume that **symbols** is a hash table that is searched using the routine **hashget**, which returns a non-zero pointer to a table entry if successful and zero if unsuccessful. The assumption states that the argument to **delete_name** should have an entry in **symbols**. In particular, upon entry to **delete_name** a call to **hashget** with the name argument must return a non-zero or "true" result. The promise states that **delete_name** removes the record for its argument from **symbols**. In other words, upon exit from **delete_name** a call to **hashget** with the

```
void print_warning(code, line, file)
int code;
int line;
char* file;
/*@
    assume warnings_on;
@*/
{
    ...
}
```

Figure 9: Specifying the Context in which a Function is Called.

```
promise strcmp(name, in strdup(name)) == 0;
```

Figure 10: Specifying a Frame Constraint for Function **delete_name** of Figure 8.

name argument returns 0, and thus the negation of the **hashget** result (obtained using the negation operator **!**) is true.

3.1.4 The Context in Which a Function Is Called

Sometimes a function should be called only within certain processing contexts, even though the function may behave correctly within all contexts. Assertions can be used to ensure that functions are called in appropriate contexts. For instance, Figure 9 shows a function **print_warning** that is used by a language processor to output detailed warning messages only if a certain command-line option has been given to the processor (as indicated by a non-zero value for the global variable **warnings_on**). The function always generates a correct warning message for any combination of code number, line number and file name. But the assumption is used to check that the function is called only when the appropriate command-line option has been specified.

3.1.5 Frame Specifications

Functions are often required to leave certain data unchanged. Assertions can be used to express such requirements, which are called *frame specifications*. For instance, to specify that the function **delete_name** shown in Figure 8 should not modify its argument (a string passed by reference), the promise shown in Figure 10 can be added to its interface assertions. The promise uses the standard C library function **strcmp** (which returns 0 when its two string arguments are equal) to ensure that the values of the string upon entry to and exit from the function are equal. Notice that it

```
#define BUFFSIZE 80
char buffer[BUFFSIZE];

void fill_and_truncate()
/*@
  promise some (int i=0; i < BUFFSIZE; i=i+1)
                    buffer[i] == '\0';
@*/
{
  ...
}
```

Figure 11: Specifying a Subrange Constraint.

is not sufficient to refer to the entry value of the string with the expression **in name**, since **in name** evaluates to the entry value of the *pointer* **name**, not the entry value of the string pointed to by **name**. The standard C library function **strdup** creates a copy of a string, and thus the expression **in strdup(name)** can be used to provide a pointer to the entry value of the string pointed to by **name**.

3.1.6 Subrange Membership of Data

C does not allow the specification of subrange constraints on numeric types. This weakness can be overcome with simple assertions that specify appropriate bounds on the values of numeric data. However, this weakness becomes particularly troublesome in C's treatment of arrays, which are indexed by integers. C has a rather weak notion of array, which is basically just a region of memory that is referenced through a pointer. Overrunning array bounds in C is thus very common, especially when handling strings (character arrays), which in C require an additional string termination character that is frequently neglected. Assertions can be used to specify constraints that guard against the mishandling of arrays. For example, Figure 11 shows a function **fill_and_truncate** that is used to fill a global string buffer with a line of input text, truncating the line if it exceeds the size of the buffer. The promise states one of the constraints the function must satisfy, namely that according to C programming conventions, it must place the string termination character **'\0'** at the end of the buffered text, but still within the bounds of the global buffer. That is, there must be a subrange (of size one) of the array **buffer** that contains the string termination character.

This constraint is expressed using a bounded existential quantifier, introduced by the keyword **some**, to state that upon exit from the function some element of the buffer must contain the string terminator. An APP quantifier can be thought of as a sequential itera-

```
assume kind >= identifier && kind <= string;
assume kind == identifier || kind == number
   ||  kind == string;
```

Figure 12: Specifying an Enumeration Constraint for Function **store_token** of Figure 7.

tor over a set of values, with the quantified expression evaluated for each element in the set; these individual evaluations are combined in the obvious way for the particular kind of quantifier. An APP quantifier specification contains an existential specifier (**some**) or universal specifier (**all**) followed by a parenthesized sequence of three fields separated by semi-colons. The first field is a declaration of the variable over which quantification is to be performed, including its name, type and the initial value of the set. The second field is a condition that must be true in order for the iteration to continue. The third field is an expression that describes how to compute the next value in the set. Thus, the quantified expression in the above example can be read as follows: There exists some **i** between 0 and **BUFFSIZE**−1 such that the **i**th character of **buffer** is the string termination character.

3.1.7 Enumeration Membership of Data

As is the case with arrays in C, enumeration types in C are also weak, in that they are type compatible with the integers. In particular, enumeration literals are interchangeable with their internal integer values, and any integer can be used where an enumeration literal is required. Assertions can be used to ensure that variables of an enumeration type contain valid values of the type. For instance, the function **store_token** shown in Figure 7 takes an argument whose value belongs to an enumeration type. The assumptions shown in Figure 12 can be added to the function's interface assertions. The two assumptions are equivalent, and they both check that the function is given valid values of the enumeration type **Token_Kind**.[5]

3.1.8 Non-Null Pointers

C programs make very extensive use of pointers, for referencing arrays and strings, for accessing dynamically allocated storage, and for passing arguments to functions by reference. Assertions can be used to specify

[5] Of course, the second form of assertion must be used for enumeration types whose literals are given explicit, non-contiguous internal values, such as

```
enum Token_Kind { identifier=2, number=4, string=6 };
```

```
assume token &&
       ((kind == identifier
      &&  token[0] >= 'a'
      &&  token[0] <= 'z')
      ||  (kind == number
      &&  token[0] >= '0'
      &&  token[0] <= '9')
      ||  (kind == string
      &&  token[0] == '"'));
```

Figure 13: Specifying a Pointer Constraint for Function **store_token** of Figure 7.

when pointers should be non-null. Such assertions are especially useful because the self-checks they generate can provide information prior to the aborted execution and core dump that usually result from invalid pointer manipulation. The assumption "**assume** x && y && x != y" specified on the function **swap** shown in Figure 2 illustrates an assertion that constrains a pointer argument to be non-null.

In addition, it is prudent to first state that a pointer is non-null before specifying constraints on the data to which the pointer is pointing. For instance, the assumption on function **store_token** of Figure 7 should be strengthened as shown in Figure 13 to ensure that the string pointer **token** is non-null (and thus "true") before it is dereferenced in the array subscripting operations.

3.2 Specification of Function Bodies

Function bodies often contain long sequences of complex control statements, which offer many opportunities for introducing faults. Assertions that are stated in terms of a particular function implementation can be used as "enforced comments" to guard against such faults.

3.2.1 Condition of the Else Part of Complex If Statements

The implicit condition of the default branch of an **if** statement (i.e., the final **else** part) is often intended to be stronger than the simple negation of the disjunction of the explicit, non-default conditions. Assertions can be used to specify the intended default condition explicitly. Suppose that the function **store_token** shown in Figure 7, rather than taking an argument indicating the kind of token it is given, instead makes that determination in its implementation. The function might use an **if** statement like the one shown in Figure 14. The final, default **else** branch of this **if** statement will be executed for all values of **token** whose first character is not a digit or lower-case letter. But since the function should only

```
if (token[0] >= 'a' && token[0] <= 'z')
{
   /* Handle identifier */
   ...
}
else if (token[0] >= '0' && token[0] <= '9')
{
   /* Handle number */
   ...
}
else
{
   /* Handle string */
   /*@
       assert token[0] == '"';
   @*/
   ...
}
```

Figure 14: Specifying the Condition of a Default **else** Branch.

be processing string tokens in the default branch, the assertion restricts the execution of the default branch to those situations in which the first character of **token** is the double-quote character.

3.2.2 Condition of the Default Case of a Switch Statement

As is true of **if** statements, **switch** statements often contain a default case that is intended to operate on only a subset of the possible domain of the default case, especially when the switch is performed on a value of an enumeration type. Assertions can be used to describe the limited domain, in a manner similar to the way the default **else** branch was constrained in the **if** statement of Figure 14. Since it is wise to supply default cases for **switch** statements even if they should never be executed, a special form of this kind of assertion is an assertion that always fails, as shown in Figure 15.

3.2.3 Consistency Between Related Data

It is often necessary to process related data in different ways and ensure that the data remain consistent after processing. For instance, consider a function that inserts a new entry into a priority queue before performing other processing on the new entry. The function might first use a loop to find where in the queue the new entry belongs. The function might then use a separate check to determine if the new entry was placed at the end of the queue, in which case the queue's tail pointer would need to be updated. An assertion like the one shown

```
switch (kind)
{
   case identifier:
        ...
        break;

   case number:
        ...
        break;

   case string:
        ...
        break;

   default:
        /* Control should never reach here */
        /*@
            // assert that 0 (i.e., false)
            is true:
            assert 0;
        @*/
        break;
}
```

Figure 15: Specifying the Condition of the Default Branch of a **switch** Statement.

```
/*@
    assert new_entry->next != 0
       || queue.tail == new_entry;
@*/
```

Figure 16: Specifying Consistency Between Related Data.

in Figure 16 can be used to ensure that the two parts of the insertion code treat the tail pointer consistently. The assertion requires the tail pointer to point to the new entry if the new entry contains a null forward link after insertion.

3.2.4 Intermediate Summary of Processing

Assertions can be used to periodically summarize the effect of a complex function at intermediate points in its body. The assertion in the body of function **swap** shown in Figure 2 illustrates this kind of assertion. Following some non-intuitive arithmetic manipulations of the integer arguments, the body assertion states exactly where one of the promises of **swap** must become satisfied.

4 Discussion and Conclusions

This paper has described an assertion processing system for C and UNIX called APP. APP provides a rich collection of features for specifying not only the assertions themselves but also the response to failed runtime assertion checks. APP can process approximately 20,000 lines of C code per CPU-minute on a Sun-4 workstation. The assertion checks generated by APP introduce negligible time and space overhead into the generated self-checking programs.

This paper has also described a classification of assertions that is based on experience using APP. Table 1 summarizes this classification according to the kind of system behavior each class of assertion is intended to capture. In their empirical study, Perry and Evangelist described 15 kinds of interface faults found in the software change request data that they analyzed [PE85, PE87].[6] Most of the kinds of faults they describe can be guarded against by the assertions described in this paper. Table 2 summarizes the kinds of interface faults they identified along with an indication of which assertions from Table 1 appear to be most effective at detecting each kind of fault. In some cases the faults described by Perry and Evangelist call for broader classes of assertions, such as preconditions or postconditions, which are subsumed by the classification of this paper; these broader classes are noted in Table 2 where applicable. Nearly 50 percent of the faults in the Perry and Evangelist study were faults of inadequate error processing, construction or inadequate functionality. As Table 2 shows, the assertions described in this paper can be used to guard against these common classes of faults.

Systems that are specified with assertions need not contain a complete specification of the system, in any sense of the word "complete". Incomplete specifications that capture the essence of the intended behavior are quite sufficient for reliably detecting software faults at runtime. Experience with APP has demonstrated that faults in reasonably well annotated code (with at least every function interface supplied with assertions) often generate multiple assertion violations, in some cases making fault elimination a simple matter of interpreting the diagnostic messages output by the self-checks.

An interesting thing to note about the assertion classification described in this paper is the absence of certain classes of assertions that are important for program verification, especially loop invariants and inductive assertions. I did not rely on such assertions in the various systems I have developed with APP. Inductive assertions can be notoriously difficult to construct and do not always capture one's intuitive understanding of intended system behavior. Thus, it may be that the kinds of assertions that are useful for program verification may not be well suited to runtime fault detection, and vice versa.

The *Yeast*[7] event-action system [RK91] provides an excellent example of a system I have developed with APP. Yeast comprises roughly 12,000 lines of C, *yacc* and *lex* code. I developed Yeast with one other person, who developed roughly half of the source code without assertions and without using APP. My half of the source code contains 116 assertions in 95 assertion regions. Of these 95 assertion regions, 39 are function interface specifications, which contain a total of 61 assertions. The self-checking executables are 12% larger than the non-self-checking executables, and they run with no discernible difference in speed.

Of the 16 faults we have removed since first releasing Yeast to other people within AT&T, 7 were discovered by one or more assertion violations; the assertions that detected these faults belonged to classes **I2**, **I4**, **I7**, **I8** and **B3** of Table 1. Of the 9 faults that were not detected by assertions, 5 could have been caught by assertions that were not written, 3 were detected by a dynamic storage certification routine, and 1 would have been nearly impossible to guard against with an assertion. This last fault was due to an access through an invalid pointer into dynamic storage and could have been guarded against only by an assertion that describes what a valid pointer value is.

Until sophisticated static analysis and verification become practical for large systems comprising many modules, developers of large systems must rely on alternative means of determining the correctness of their systems. Assertion checking is one such alternative—it is powerful, practical, scalable and simple to use. A major goal in the development of APP has been to begin providing developers of large systems written in C with the means of applying assertions to their development efforts. The current implementation of APP allows developers to write their assertions using customary C programming style and to create self-checking programs using existing UNIX and C programming environments. APP will be extended to support other kinds of assertions and higher-level abstraction facilities. New features will include constraints on types and global variables, a richer abstraction of arrays and other abstract data types, and constructs for specifying interactions between program units that are larger than functions. In addition, a version of APP will be developed for C++. Finally, APP can be made available to universities for research use under certain terms and conditions.

Once programming with assertions becomes more

[6] The reader is referred to Perry and Evangelist's paper for a more detailed description of their fault classification.

[7] Yet another Event-Action Specification Tool.

Code	Description
I	*Specification of Function Interfaces*
I1	Consistency Between Arguments
I2	Dependency of Return Value on Arguments
I3	Effect on Global State
I4	Context in Which Function Is Called
I5	Frame Specifications
I6	Subrange Membership of Data
I7	Enumeration Membership of Data
I8	Non-Null Pointers
B	*Specification of Function Bodies*
B1	Condition of **Else** Part of **If** Statement
B2	Condition of Default Branch of **Switch** Statement
B3	Consistency Between Related Data
B4	Intermediate Summary of Processing

Table 1: Summary of Classification of Assertions.

Interface Fault	Effective Assertions
Construction (mismatched separate interface and implementation)	**I1**, **I2**
Inadequate Functionality	**I1**, **I3**, **I4**, preconditions
Disagreements on Functionality	**I2**, **I3**, **I4**, **I5**
Changes in Functionality (requirements changes)	*none*
Added Functionality (requirements changes)	*none*
Misuse of Interface	**I**
Data Structure Alteration	**I1**, **I5**, **I6**, **I7**, **I8**, **B3**
Inadequate Error Processing	**I**, **B**
Additions to Error Processing	**I**, **B**
Inadequate Postprocessing (resource deallocation)	**I3**
Inadequate Interface Support	**I2**, **I3**, postconditions
Initialization/Value Errors	**I1**, **I3**, **I6**, **I7**, **I8**, **B3**
Violation of Data Constraints	**I2**, **I3**, **I5**, **I6**, **I7**, **B3**
Timing/Performance Problems	*none*
Coordination of Changes	**I**

Table 2: Perry and Evangelist's Classification of Interface Faults.

widespread, larger questions will arise as to how programming with assertions relates to other phases and activities of software development. Assertions are nothing more than formal specifications that describe some aspect of system behavior, albeit at a relatively low level of abstraction. Thus, it would be desirable to provide mechanisms for deriving or synthesizing assertions from formal requirements, architecture and high-level design specifications. For instance, many CASE environments provide support for generating code templates from design-level specifications; such a feature could be extended to embed appropriate function interface assertions within the generated templates. In addition, assertions provide information that can be used to help select and/or check the adequacy of test data for both white-box and black-box testing. For instance, it would

be useful to develop assertion-based test data adequacy criteria along the lines of the data flow criteria developed by Weyuker and others [RW85]. By having assertions directly traceable to a requirements specification of a system, and having test data tied explicitly to assertions in the source code of the system, one can conceive of more rigorous methods of execution-based system validation driven directly by system requirements.

Acknowledgments

I am grateful to Alex Wolf, Dewayne Perry, Ed Amoroso and Filip Vokolos, who provided many helpful suggestions and comments on the content of this paper. The comments of the anonymous referees were also helpful in clarifying some of the key points of the paper.

References

[BMU75] B.W. Boehm, R.K. McClean, and D.B. Urfrig. Some experience with automated aids to the design of large-scale reliable software. In *Proceedings of the International Conference on Reliable Software*, pages 105–113. ACM and IEEE Computer Society, April 1975.

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(3):255–265, March 1979.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium in Applied Mathematics, Volume XIX*, pages 19–32. American Mathematical Society, April 1967.

[Fow90] Glenn S. Fowler. A case for make. *Software—Practice & Experience*, 20(S1):35–46, June 1990.

[HC88] Richard C. Holt and James R. Cordy. The Turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[LCKS90] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, SE-16(4):432–443, April 1990.

[LST91] David C. Luckham, Sriram Sankar, and Shuzo Takahashi. Two-dimensional pinpointing: Debugging with formal methods. *IEEE Software*, 8(1):74–84, January 1991.

[Luc90] David C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs.* Springer-Verlag, 1990.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[PE85] Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults. In *Proceedings of the International Symposium on New Directions in Computing*, pages 32–38. IEEE Computer Society, August 1985.

[PE87] Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults—an update. In *Proceedings of the 20th Annual Hawaii International Conference on System Sciences*, Volume II, pages 113–126, January 1987.

[Per89] Dewayne E. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12. IEEE Computer Society, May 1989.

[RK91] David S. Rosenblum and Balachander Krishnamurthy. An event-based model of software configuration management. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 94–97. ACM SIGSOFT, 1991.

[RSL86] David S. Rosenblum, Sriram Sankar, and David C. Luckham. Concurrent runtime checking of Annotated Ada programs. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 10–35. Springer-Verlag (Lecture Notes in Computer Science No. 241), December 1986.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985.

[San89] Sriram Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Department of Computer Science, Stanford University, August 1989. Technical Report 89-1282.

[San91] Sriram Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of TAV4—The 4th Software Testing, Analysis and Verification Symposium*, pages ??–?? ACM SIGSOFT, October 1991.

[SF75] Leon G. Stucki and Gary L. Foshee. New assertion concepts for self-metric software validation. In *Proceedings of the International Conference on Reliable Software*, pages 59–71. ACM and IEEE Computer Society, April 1975.

[SR86] Sriram Sankar and David S. Rosenblum. The complete transformation methodology for sequential runtime checking of an Anna subset. Technical Report 86-301, Computer Systems Laboratory, Stanford University, June 1986. Program Analysis and Verification Group Report 30.

[SRN85] Sriram Sankar, David S. Rosenblum, and Randall B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference*, pages 285–296. Cambridge University Press, May 1985.

[YC75] S.S. Yau and R.C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–457. ACM and IEEE Computer Society, April 1975.