A Practical Approach to Programming With Assertions

David S. Rosenblum, *Member, IEEE*

***Abstract*— Embedded assertions have been recognized as a potentially powerful tool for automatic runtime detection of software faults during debugging, testing, maintenance and even production versions of software systems. 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 PreProcessor for C programs developed in UNIX-based development environments. APP has been used in the development of a variety of software systems over the past five 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.**

***Index Terms*—Anna, APP, assertions, C, consistency checking, formal specifications, formal methods, programming environments, runtime checking, software faults.**

I. 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, more than 25 years ago Floyd demonstrated the need for loop assertions for verification of programs [1]. Luckham *et al.* elaborated the basic principles outlined by Floyd into an algorithm for mechanical program verification that was based on the generation and proof of simple assertions called *verification conditions* [2]; this algorithm was implemented in the Stanford Pascal Verifier [3]. In addition to their use in formal verification, assertions have long been recognized as a potentially powerful tool for automatic runtime detection of software faults during debugging, testing and maintenance. More recently, assertions have been viewed as a permanent defensive programming mechanism for runtime fault detection in production versions of software systems [4].

As long ago as the 1975 International Conference on Reliable Software, several authors described systems for deriving runtime consistency checks from simple assertions [5]–[7]. 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 times during the execution of the program.

Today, assertion features are available as programming language extensions, as programming language features, and in complete high-level formal specification languages. The C programming language [8] has traditionally provided a simple assertion facility in the form of the predefined macro **assert**, which is expanded inline into an **if** statement that aborts the program if the assertion expression evaluates to zero. Extensions have been proposed for other languages such as C++ [9] that originally provided no higher-level assertion capability [10], [11]. Still other programming languages, such as Turing [12] and Eiffel [13], provide assertion features as part of the language definition. Such languages can be used to specify system behavior at the design level. Many such languages are suitable not only for generating runtime consistency checks, but also for static analysis of semantic consistency, as in the Inscape environment [14]. 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 [15]. This work includes (1) a method of generating consistency checks from annotations on types, variables, subprograms and exceptions [16], [17]; (2) a method that uses incremental theorem proving to check algebraic specifications at runtime [18], [19]; (3) a method of generating consistency checks that run in parallel with respect to the execution of the underlying system [20], [21]; and (4) a method of constructing large software systems based on algebraic specification of system modules [22].

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

Manuscript received February, 1993; revised October, 1994.
D. S. Rosenblum is with AT&T Bell Laboratories, Murray Hill, NJ 07974 USA (e-mail: dsr@research.att.com).
IEEE Log Number 9407721.

0098-5589/95$04.00 © 1995 IEEE

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 dearth of case studies that describe experience using assertions to build real systems. Consequently, most software developers have little idea of what information should be specified in assertions.

This second point has left some programmers with the impression that writing assertions is akin to "writing the program twice". Yet an effective assertion does not merely restate something appearing in the program text; unlike the program, it succinctly and unambiguously states an important property of the program in a way that is understandable by anyone who reads it. 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.* [23].

To address these concerns and begin making assertions a natural and practical aid to software development, I have been developing and using an assertion processing system called APP, an Annotation PreProcessor for C programs developed in UNIX[1]-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 five years ago. Since then, I have applied APP to the development of systems 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 that were developed with APP and the faults that were detected by the generated checks. To demonstrate the effectiveness of the proposed classes of assertions, the paper presents an analysis of the fault data from one of the systems that was developed. The paper concludes with a discussion of plans for future enhancements to and experimentation with APP. 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.

[1] UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/OPEN corporation.

II. APP

In Perry and Evangelist's empirical study, it was shown that most software faults are interface faults [24], [25]. Hence, APP was initially designed to process assertions on function interfaces, as well as simple assertions in function bodies.[2] APP also supports a number of facilities for specifying the response to a failed assertion check and for controlling the amount of checking that is performed at runtime.

A. 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.[3]

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 nonzero expression is true. To discourage writing assertion expressions that have side effects, APP disallows the use of C's assignment, increment and decrement operators in assertion expressions. Of course, functions that produce side effects can be invoked within assertion expressions, but such expressions should be avoided except in the rarest of circumstances (such as in the example of Section III-A.5 below), since assertions should simply provide a check on a computation rather than be an active part of it.

APP recognizes two enhancements to the C expression language within assertion regions: quantifiers and the operator **in**. Existential and universal quantification over finite domains can be specified using a syntax that is similar to C's syntax for **for** loops. 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.[4] These extensions are illustrated in the examples below.

[2] In keeping with the terminology of C, this paper uses the generic term "function" to refer to subprograms. A C function whose return type is **void** returns no result to its caller, while a non-**void** function in C always returns a result.

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

[4] The operator **in** of APP is similar to the *hook* in VDM, whereby a hook is placed over a variable in a post-condition of an operation to refer to the value of the variable prior to the execution of the operation [26]. In a similar fashion, in the Z notation a variable can be unprimed or primed within an operation schema to refer, respectively, to the value of the variable in the state before or the state after the execution of the operation [27]. The operator **in** of APP differs from the operator **in** of Anna, in that the former always applies to the entry state of the enclosing function, while the latter applies to the *observable state* of the computation immediately prior to the assertion; in the case of **in**-annotations on function interfaces, this observable state is the entry state of the function. Thus, the operator has the same meaning in both APP and Anna within interface assertions and different meanings within body assertions. See the Anna Reference Manual for details [28].

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 single 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**. As was mentioned above, the choice of what assertion constructs to support in APP was governed primarily by a desire to provide a facility for specifying function interfaces. Having gained sufficient experience with these constructs, other constructs will be supported in future versions of APP, including assertions that apply over smaller regions of function bodies such as loops and nested blocks.

To 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 Fig. 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 nonnegative 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 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 (described below). Note that these assertions merely state what the function does, not how it does it.

```
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);
@*/
{
    ...
}
```

Fig. 1. Specification of function **square_root**.

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 using a series of exclusive-or operations on the integer values. The function can be specified and implemented as shown in Fig. 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 at the point where one of the promises must become satisfied.

```
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;
}
```

Fig. 2. Specification of function **swap**.

As a final example, consider a function **sort** that sorts two arrays of integers. The specification of this function in Fig. 3 describes its required behavior at a level of abstraction that allows the use of any sorting algorithm to implement the function body. In this function, **x** is the unsorted input array, and **size** is the number of elements in the array. The function returns a pointer to the sorted result. The specification of **sort**

```
int* sort(x,size)
int* x;
int size;
/*@
    assume x && size > 0;
    return S where S   // S is non-null
           && all (int i=0; i < in size-1; i=i+1)  S[i] <= S[i+1]   // S is ordered
           && all (int i=0; i < in size; i=i+1)
                  some (int j=0; j < in size; j=j+1)
                       x[i] == S[j];   // S is a permutation of x
@*/
{
    ...
}
```

Fig. 3. Specification of function **sort**.

uses quantifiers to state both the obvious ordering requirement of the result (but disallowing duplicate elements) as well as the requirement that the result must be a permutation of the input array; note that informal comments are used in the figure to show exactly where these requirements are stated.

An APP quantifier can be thought of as a sequential iterator 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. Syntactically, a quantified expression resembles a **for** loop in C. Indeed, as will be described further in Section II-C, APP translates each quantified expression into a **for** loop, with nested quantifiers translated into appropriately nested **for** loops.

As shown in Fig. 3, an APP quantifier specification contains the existential specifier **some** or universal specifier **all**, followed by a parenthesized sequence of three fields separated by semicolons. 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 first universally quantified expression in the return annotation says that each element but the last of the result must be less than its successor element. The second universally quantified expression contains a nested existentially quantified expression to state that for all elements of the input array **x**, there exists an equal element of the result array.

Quantifiers can be used to quantify over *any* finite set of values, not just a range of integers. For instance, a quantifier over the elements of a linked list would specify the head of the list as the initial value, a non-null *next* pointer as the continuation condition, and a dereference of the *next* pointer to retrieve the next value in the set. Since it is considered impractical for runtime checks to quantify over large domains such as the full range of integers, the syntax of quantifiers was designed to encourage careful quantification over reasonably sized domains. Experience has shown that the syntax achieves this aim without reducing the expressive power of traditional quantifiers of first-order predicate logic.

B. Violation Actions, Predefined Macros and Severity Levels

APP converts each assertion to a runtime check, which tests for the violation of the constraint specified in the assertion. If the check fails at runtime, then additional code generated with the check is executed in response to the failure. The default response code generated by APP prints out a simple diagnostic message such as the following, which indicates the violation of the first promise of function **swap**:

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

The default response provides a minimal amount of information needed to 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 Fig. 4 (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 Fig. 5 to print

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

Fig. 4. 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);
   }
```

Fig. 5. Enhancement of the violation action of Fig. 4.

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

Fig. 6. Severity levels for assertions of function **square_root**.

out the same information that is 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 for each assertion, with 1 being the default and indicating the highest severity. A severity level indicates the relative importance of an assertion and determines whether or not the assertion will 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 Fig. 6. 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 [29]. The mechanism for controlling the checking level at runtime is described below.

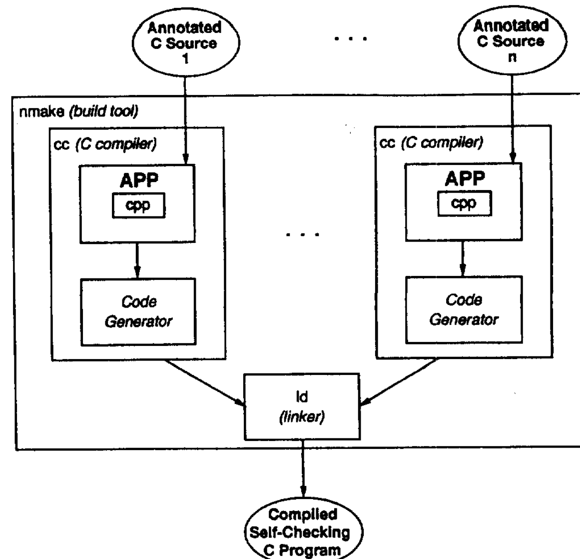


Fig. 7. Generating self-checking C programs with APP.

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.

C. Generating and Running Self-Checking Programs

APP translates an input annotated C program into an equivalent C program with embedded assertion checks. APP has the same command-line interface as *cpp*, the standard preprocessor pass of C compilers (which are usually called *cc*). In particular, APP accepts the macro definition options **−D** and **−U** and the interface or "header" file directory option **−I**, and it performs all of the macro preprocessing of *cpp* in addition to its assertion processing. 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 every C compiler. Furthermore, standard build tools such as *make* [30] and *nmake* [31] can be used to build executable self-checking programs, with only slight modifications to existing *makefiles* or build scripts. These build techniques are illustrated in Fig. 7, which depicts *nmake* compiling the *n* source files of some program with APP and then linking the resulting object files together into a self-checking executable. 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 programming environments.

Execution of a self-checking program proceeds with checking performed at the severity level specified by the user in the UNIX shell environment variable **APP_OPTIONS** (or at the default level if the environment variable is undefined). Note that a self-checking program can be treated like any other program in a C programming environment. For instance, a

self-checking program can be run inside a symbolic debugger such as *dbx*. The debugger can be used to set breakpoints at assertions, single-step through them, trace their execution, and so on, all relative to the contents and line numbering of the original source files in which the assertions were specified.

APP translates its input in a single pass without building an internal parse tree. However, APP uses internal text buffers and buffer stacks for temporary storage of certain regions of the translated source text. This use of buffers arises from the need to order some regions differently from the order in which they appear in the input source text:

- Function pre- and postconditions appear syntactically *before* their associated function body. Yet the precondition checks must be inserted immediately after the declarations in the function body, while the postcondition checks must be inserted at the end of the function body (with **return** statements in the original source translated to **goto**s to the postcondition checks).
- **In** expressions require the generation of temporary variables for their evaluation, and these temporary variables must be placed at the very beginning of the function body to ensure that the entry values of the expressions are captured before any other computation takes place within the body.
- Each quantified expression is translated into a *loop* that evaluates the quantified expression into a temporary variable, followed by a *usage* of the temporary variable in the surrounding context. Multiply-nested quantified expressions, such as the one shown in Fig. 3, are translated in such a way that the loop/usage pair for an inner quantified expression appears between the loop and the usage for the next outer quantified expression.

These regions of translated source text are saved in buffers until it is appropriate to output them.

III. A CLASSIFICATION OF ASSERTIONS

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

Based on this experience, it now would be fruitful to examine these systems and to characterize the kinds of assertions that were most effective in uncovering faults. The categories of assertions described 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. Table I summarizes the assertion classification, which is organized according to the kind of system behavior each class of assertion is intended to capture.

The examples provided for each category are abstracted from actual assertions. A few of the assertions are used to overcome inherent weaknesses in the type system of C; in certain cases such assertions would not be needed in programs written in languages that provide a strong type model, such as Ada.[5] However, most of the assertions described below 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.

When using this classification, it should be remembered that the general goal in writing any assertion should be to specify some required constraint or relationship of the system succinctly and at a relatively high level of abstraction. It is not necessary that this specification be complete in any formal sense; a specification of only the most important aspects of a constraint or relationship can provide a high degree of fault detection ability. Given a particular informal constraint on a function, it may be difficult sometimes to develop a formal assertion of the constraint that is less complex than the function implementation itself. Even so, the redundancy provided by such an assertion may prove useful, in the sense that an inconsistency between the assertion and the function implementation would be symptomatic of some incompleteness in one's understanding of the informal constraint on which they are both based.

A. 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.

1) Consistency Between Arguments: For each function in the system, specify how the value of each of its arguments depends on the values of its other 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. In most cases these assertions will be preconditions on arguments passed by value. 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 Fig. 8, the function takes as arguments an enumeration value specifying

[5] Flater *et al.*, describe a system called *Robust C* that automatically instruments C programs for runtime detection of the most common classes of C coding faults, such as violation of array bounds [32]. And a number of research techniques and commercial tools are available for automatically detecting memory-related coding faults in C programs at runtime (e.g., see Austin *et al.* [33], and *Purify* [34]).

```
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] == '"');
@*/
{
    ...
}
```

Fig. 8. Specifying consistency between function arguments.

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 zeroth component of the character array pointed to by **token**) should be a lower case letter.[6] 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.

2) Dependency of Return Value on Arguments: For each function in the system, provide postconditions that specify how its return value(s) depends on the values of its arguments.

Assertions can be used to specify 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. This relationship need not be specified completely; it suffices to merely state the most important aspects of this relationship. The second return constraint of function **square_root** shown in Fig. 1 and the promises of function **swap** shown in Fig. 2 illustrate this kind of assertion.

3) Effect on Global State: For each function in the system, specify what changes the function makes to the values of the global variables that are visible to it.

Functions in procedural languages often have side effects. Assertions can be used to specify the key ways in which 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 Fig. 9; assume that **symbols** is a hash table that is searched using the routine **hashget**, which returns a nonzero 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 nonzero or "true" result. The promise states that **delete_name** removes the record for its argument from **symbols**, so that upon exit from **delete_name**, a call to **hashget** with the name argument must return zero, and thus the negation of the **hashget** result (obtained using the negation operator !) must be true.

[6]C arrays are always indexed starting at zero.

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

Fig. 9. Specifying the effect on the global state.

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

Fig. 10. Specifying the context in which a function is called.

4) The Context in Which a Function is Called: For each function in the system, specify how the values of its arguments and the values of the global variables visible to it govern when it is valid for the function to be 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, Fig. 10 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 nonzero 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.

5) Frame Specifications: For each function in the system, specify each case when the value of an argument passed to the

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

Fig. 11. Specifying a frame constraint for function **delete_name** of Fig. 9.

function by reference, or the value of a global variable visible to the function, is to be left unchanged by the function.

Functions are often required to leave certain data unchanged. Such requirements, which are called *frame specifications*, are usually implicitly derived or assumed in proof-based reasoning systems, but for purposes of runtime checking they must be stated explicitly. Assertions can be used to state a system's frame specifications. For instance, to specify that the function **delete_name** shown in Fig. 9 should not modify its argument (a string passed by reference), the promise shown in Fig. 11 can be added to its interface assertions. The promise uses the standard C library function **strcmp** (which returns zero 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 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 heap-resident 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**.[7] Notice also that because the function might modify the pointer value of **name**, the exit value of **name** may differ from its entry value. Thus, the **strcmp** expression checks that the *location* in memory designated by **name** upon entry to the function (i.e., **in name**) still contains the *value* it had upon entry (i.e., **in strdup(name)**).

This example also illustrates a rare situation where it is desirable for an assertion expression to produce a side effect, in this case an allocation of heap memory. However, APP is able to compensate for this particular side effect, because it ensures that any heap memory that is dynamically allocated as a result of the evaluation of an assertion expression is deallocated upon exit from the function enclosing the assertion.

6) Subrange Membership of Data: For each function in the system, specify all subrange constraints on the values of its arguments, return value(s) and global variables that are of numeric type. Also specify all subrange constraints on the values used to index its array-valued arguments, return value(s) and global variables.

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 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 overlooked. Assertions can be used to specify

[7] Note that it is not necessary to use the expression **in strdup(in name)**, since the operator **in** distributes across all subexpressions of the expression to which it is applied.

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

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

Fig. 12. Specifying a subrange constraint.

constraints that guard against the mishandling of arrays. For example, Fig. 12 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.[8]

This constraint is expressed using an existentially-quantified expression to state that upon exit from the function some element of the buffer must contain the string terminator. In particular, the expression states that there exists some **i** between zero and **BUFFSIZE**–1 such that the **i**th character of **buffer** is the string-termination character.

7) Enumeration Membership of Data: For each function in the system, specify all membership constraints on the values of its arguments, return value(s) and global variables that are of enumeration type.

As is the case with arrays, enumeration types in C are also weak, in that they are type compatible with 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 Fig. 8 takes an argument whose value belongs to an enumeration type. The assumptions shown in Fig. 13 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**.[9]

8) Non-Null Pointers: For each function in the system, specify which pointer-valued arguments, return value(s) and global variables must not be null.

C programs make very extensive use of pointers to reference arrays and strings, to access dynamically allocated storage, and to pass arguments to functions by reference. Assertions can be used to specify when pointers should be non-null. Such

[8] Anything stored after the first string-termination character would be ignored by C's string-processing functions, so it is not necessary that the function fill the unused portion of the buffer with string-termination characters each time it is called.

[9] Of course, the second form of assertion must be used for enumeration types whose literals are given explicit, noncontiguous internal values, such as `enum Token_Kind {identifier=2, number=4, string=6};`

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

Fig. 13. Specifying an enumeration constraint for function **store_token** of Fig. 8.

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

Fig. 14. Specifying a pointer constraint for function **store_token** of Fig. 8.

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 dereferencing a null pointer. The assumption "**assume** `x && y && x != y`" specified on the function **swap** shown in Fig. 2 illustrates an assertion that constrains a pointer argument to be non-null.

It is also necessary 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 Fig. 8 should be strengthened as shown in Fig. 14 to ensure that the string pointer **token** is non-null (and thus "true") before it is dereferenced in the array subscripting operations.

B. 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.

1) Condition of the Else Part of Complex If Statements: For each **if** statement in the system that contains a final **else** part, explicitly specify the implicit condition of the final **else** part as an initial assertion in that part.

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, nondefault conditions. Assertions can be used to specify the intended default condition explicitly. Suppose that the function **store_token** shown in Fig. 8, 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 Fig. 15. 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 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.

2) Condition of the Default Case of a Switch Statement: For each **switch** statement in the system that contains a **default** case, explicitly specify the implicit condition of the **default** case as an initial assertion in that case. For each **switch** statement without a **default** case, provide a **default** case containing an assertion that always evaluates to false.

```
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] == '"';
   @*/
   ...
}
```

Fig. 15. Specifying the condition of a default **else** branch.

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 Fig. 15. 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 evaluates to false, as shown in Fig. 16.

3) Consistency Between Related Data: For each function body in the system, specify consistency constraints on mutually dependent data at frequent intervals within the code that manipulates that 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 creates an entry in 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 in Fig. 17 can be used to ensure that the two parts of the insertion code have treated 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.

4) Intermediate Snapshot of Computation: For each function body in the system, specify at frequent intervals the key constraints the function body must satisfy.

```
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;
}
```

Fig. 16. Specifying the condition of the default branch of a **switch** statement.

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

Fig. 17. Specifying consistency between related data.

Assertions can be used to summarize periodically the effect of a complex function at key places in its body. The assertion in the body of function **swap** shown in Fig. 2 illustrates this kind of assertion. Because the manipulations of the integer arguments are unintuitive, the body assertion helps to identify the exact point at which one of the promises of **swap** must become satisfied.

IV. EXPERIENCE

The Yeast[10] event-action system [35] serves as an excellent example of a software system developed with APP. Yeast comprises roughly 12,000 lines of C, *yacc* and *lex* code.

[10] Yet another Event-Action Specification Tool.

TABLE I
SUMMARY OF CLASSIFICATION OF ASSERTIONS

Assertion 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 Snapshot of Computation

I developed Yeast with one other person, and each of us developed roughly half of the source code; the other person developed his half 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 3.7% larger than the nonself-checking executables, and they run with no discernible difference in speed.[11]

Since first releasing Yeast to other people within AT&T, we have discovered and removed 19 faults, all of which were interface faults. Ten of these faults were located in my half of the code. In their empirical study, Perry and Evangelist identified 15 different kinds of interface faults in the software change request data that they analyzed [24], [25]; Table II summarizes their interface fault classification.[12] Table III characterizes each of the 19 faults in Yeast according to the Perry and Evangelist fault classification of Table II. Table III also identifies which kinds of assertions in the classification of Table I revealed each fault. The faults are listed in increasing chronological order of discovery. Included in the description of each fault is an indication of whether or not the fault was in a function or functions that had been specified with assertions; this information indicates that there was no clear correlation between the location of a fault and the location of the assertions that revealed it.

[11] In the version of this paper that appeared in the ICSE-14 conference proceedings, the size increase was erroneously reported as 12%, which included both assertion checks *and* code that was inserted by the compiler to support the use of a symbolic debugger.

[12] The reader is referred to Perry and Evangelist's papers for a more detailed description of their fault classification. Other useful fault classifications have been described by Ostrand and Weyuker [36] and Endres [37].

TABLE II
PERRY AND EVANGELIST'S CLASSIFICATION OF INTERFACE FAULTS

Fault Code	Interface Fault
F1	Construction (mismatched separate interface and implementation)
F2	Inadequate Functionality
F3	Disagreements on Functionality
F4	Changes in Functionality (requirements changes)
F5	Added Functionality (requirements changes)
F6	Misuse of Interface
F7	Data Structure Alteration
F8	Inadequate Error Processing
F9	Additions to Error Processing
F10	Inadequate Postprocessing (resource deallocation)
F11	Inadequate Interface Support
F12	Initialization/Value Errors
F13	Violation of Data Constraints
F14	Timing/Performance Problems
F15	Coordination of Changes

TABLE III
SUMMARY OF FAULTS DISCOVERED IN THE YEAST SYSTEM

Fault Number	Fault Code (from Table II)	Violated Assertion(s) (from Table I)	In Function(s) with Assertions?
1	F10	*none*	yes
2	F3	I8	no
3	F12	*none*	yes
4	F14	*none*	no
5	F8	I1, I4, B3	no
6	F2	*none*	yes
7	F7	*none*	yes
8	F2, F12	*none*	yes
9	F12	*none*	yes
10	F1	*none*	no
11	F6	*none*	no
12	F6	I2	yes
13	F12	*none*	no
14	F9	I2	yes
15	F11	I2, I7, B3	no
16	F2	I2	yes
17	F14	*none*	yes
18	F8	B3	no
19	F9	I4, B3	no

TABLE IV
FAULTS IN YEAST THAT COULD HAVE BEEN DETECTED BY ASSERTIONS

Fault Number	Assertions Needed (from Table I)
1	I8, B3
3	I2, B4
7	I3
8	I2
9	B3
13	B4

TABLE V
EFFECTIVE ASSERTIONS FOR DETECTING INTERFACE FAULTS

Fault Code (from Table II)	Effective Assertions (from Table I)
F1	I1, I2
F2	I1, I3, I4, preconditions
F3	I2, I3,I4, I5
F4	*none*
F5	*none*
F6	I
F7	I1, I5, I6, I7, I8, B3
F8	I, B
F9	I, B
F10	I3
F11	I2, I3, postconditions
F12	I1, I3, I6, I7, I8, B3
F13	I2, I3, I5, I6, I7, B3
F14	*none*
F15	I

Of the 19 faults, 8 were discovered by one or more assertion violations. Of the 11 faults that were not detected by assertions, 6 could have been caught by assertions that were not written; Table IV shows which classes of assertions were needed to detect these faults. Of the remaining 5 faults that were not detected by assertions, faults 4 and 11 were detected by a dynamic storage certification routine, while faults 6, 10, and 17 could only be detected by assertion features more powerful than those currently supported by APP (such as event sequencing constraints).

Nearly 50 percent of the faults in the Perry and Evangelist study were faults of inadequate error processing (F8), con-

struction (F1) or inadequate functionality (F2). As Table III and Table IV show, the assertions in the classification of Table I can be used to guard against these common kinds of faults as well as many of the other kinds of faults described by Perry and Evangelist. Table V summarizes these observations by noting which assertions from Table I are best-suited to detecting each kind of interface fault. In some cases the faults described by Perry and Evangelist call for broader classes of assertions, such as preconditions or postconditions; these broader classes are noted in Table V where applicable.

V. Conclusion

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 responses to failed runtime assertion checks. APP fits easily into the process of developing C programs, requiring minimal change to one's accustomed use of C and UNIX programming environments. 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. APP is currently being licensed to universities for research use under certain terms and conditions.

This paper has also described a classification of assertions that is based on experience using APP. 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. One might think that diagnostic messages from multiple violations would be useless, since diagnostics generated by violations subsequent to the first violation might not provide reliable information. However, diagnostics from multiple violations have often provided useful information about the context of the revealed fault, making fault elimination in many cases a simple matter of interpreting the diagnostic messages without the aid of any other debugging tool.

The design of APP was influenced to a great extent by the previous work on Anna. Anna is a rich specification language, and its large number of features were a natural outgrowth of the large number of programming constructs provided in Ada. This is especially noticeable with respect to packages, which are arguably the most important feature of Ada, providing a powerful means for structuring a software system and encapsulating its data types. The availability of packages in Ada required a means for specifying the behavior of a package in totality, both as an algebraic data type and as an object with state. Consequently, a significant subset of Anna deals with specification of package state and package state transitions, and with axiomatic specification of the behavior and result of combining package operations. Just as the richness of Anna derives from the richness of Ada, the simplicity of APP's specification language is well-matched to the simplicity of C. In C the primary construct of interest is the function, and thus APP has been designed primarily to support the specification of function behavior.

What APP lacks in its diversity of specification constructs, it more than makes up for in the greater flexibility it provides to the developer of self-checking programs. In Anna, the response to a failed annotation check is defined by the Anna Reference Manual (which specifies the response to be the raising of the predefined exception ANNA_ERROR) and by the Anna Transformer and Anna Debugger tools (which add generic diagnostic information and a simple debugging interface for running self-checking programs). ANNA_ERROR provides some measure of programmability for defining the response to a failed check. However, it does not identify which particular annotation or annotations were violated, and handlers for ANNA_ERROR may not always have access to the context of a violated annotation (e.g., the values of relevant variables), depending on where the handlers are defined. In contrast, APP provides violation actions in order to allow the specifier complete flexibility in defining the response to a failed assertion check, allowing selection from a wide range of possible responses. The response to a failed assertion check can be tailored to the special nature of the application, to the development task at hand, to production versions of the system, or to other aspects of system development.

Furthermore, APP provides severity levels in order to give the specifier greater control over the amount of assertion checking that is performed at runtime, without having to modify the program or rebuild the self-checking executable. Finally, while Anna supports quantification only over types, APP provides a quantification syntax that is more convenient for describing a set of iterated values and that leads to more computationally-feasible runtime 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, such as loop invariants or inductive assertions. Inductive assertions can be notoriously difficult to construct and do not always capture one's intuitive understanding of intended system behavior. In contrast, the assertions described in this paper represent an attempt to formalize such intuitions. Thus, it remains to be seen whether or not assertions that are constructed especially for program verification are also useful for runtime fault detection.

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 (such as specification of the behavior of sequences of function calls). In addition, a version of APP will be developed for C++. The C++ version of APP will provide additional specification constructs that are suited to specification of class behavior; a good starting point for the design of these constructs would be the package specification features of Anna.

Until verification and other sophisticated static analysis methods become practical for large systems comprising many

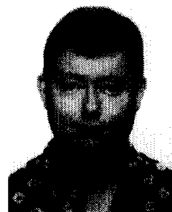
modules and several thousand lines of code, developers of large systems must rely on alternative means of identifying and removing faults in their systems. Assertion checking is one such alternative—it is powerful, practical, scalable and simple to use. While it is hoped that others can benefit from the experience described in this paper, in the future more comprehensive, controlled experimental studies on larger systems with multiple developers will help to further reveal the most effective techniques for using assertions to improve the quality and reliability of software systems.

ACKNOWLEDGMENT

The author is grateful to A. Wolf, D. Perry, E. Amoroso, F. Vokolos, P. Frankl, and D. Taylor, who provided many helpful suggestions and comments on the content of this paper. The comments of the anonymous referees were also helpful in clarifying many of the key points of the paper.

REFERENCES

[1] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, vol. XIX, American Mathematical Society, Apr. 1967, pp. 19–32.
[2] S. Igarashi, R. L. London, and D. C. Luckham, "Automatic program verification I: A logical basis and its implementation," *Acta Informatica*, vol. 4, pp. 145–182, 1975.
[3] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis, "Stanford Pascal Verifier user manual," Tech. Rep. 79-731, Dep. of Computer Science, Stanford University, Mar. 1979, Program Analysis and Verification Group Rep. 11.
[4] B. Meyer, "Applying design by contract," *IEEE Comput.*, vol. 25, pp. 40–51, Oct. 1992.
[5] L. G. Stucki and G. L. Foshee, "New assertion concepts for self-metric software validation," in *Proc. Int. Conf. Reliable Software*, ACM and IEEE Computer Society, Apr. 1975, pp. 59–71.
[6] B. W. Boehm, R. K. McClean, and D.B. Urfrig, "Some experience with automated aids to the design of large-scale reliable software," in *Proc. Int. Conf. Reliable Software*, ACM and IEEE Computer Society, Apr. 1975, pp. 105–113.
[7] S. S. Yau and R. C. Cheung, "Design of self-checking software," in *Proc. Int. Conf. Reliable Software*, ACM and IEEE Computer Society, Apr. 1975, pp. 450–457.
[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1988, 2nd ed.
[9] B. Stroustrup, *The C++ Programming Language*. Reading MA: Addison-Wesley, 1991, 2nd ed.
[10] P. Gautron, "An assertion mechanism based on exceptions," in *Proc. 4th C++ Tech. Conf.*, USENIX Association, Aug. 1992, pp. 245–262.
[11] M. P. Cline and D. Lea, "Using annotated C++," in *Proc. C++ at Work*, Sept. 1990.
[12] R. C. Holt and J. R. Cordy, "The Turing programming language," *Commun. ACM*, vol. 31, no. 12, pp. 1410–1423, Dec. 1988.
[13] B. Meyer, *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[14] D. E. Perry, "The Inscape environment," in *Proc. 11th Int. Conf. Software Eng.*, IEEE Computer Society, May 1989, pp. 2–12.
[15] D. C. Luckham and F. W. von Henke, "An overview of Anna, a specification language for Ada," *IEEE Software*, vol. 2, pp. 9–23, Mar. 1985.
[16] S. Sankar, D. S. Rosenblum, and R. B. Neff, "An implementation of Anna," in *Ada in Use: Proc. Ada Int. Conf.*, May 1985, pp. 285–296, Cambridge Univ. Press.
[17] S. Sankar and D. S. Rosenblum, "The complete transformation methodology for sequential runtime checking of an Anna subset," Tech. Rep. 86-301, Computer Systems Laboratory, Stanford Univ., June 1986, Program Analysis and Verification Group Report 30.
[18] S. Sankar, "Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs," Ph.D. thesis, Dep. of Computer Science, Stanford Univ., Aug. 1989 Tech. Rep. 89-1282.
[19] S. Sankar, "Run-time consistency checking of algebraic specifications," in *Proc. TAV4–The 4th Software Testing, Analysis and Verification Symp.* ACM SIGSOFT, Oct. 1991, pp. 123–129.
[20] D. S. Rosenblum, S. Sankar, and D. C. Luckham, "Concurrent runtime checking of Annotated Ada programs," in *Proc. 6th Conf. Foundations of Software Tech. and Theoretical Comput. Sci.* New York: Springer-Verlag (Lecture Notes in Computer Science No. 241), Dec. 1986, pp. 10–35
[21] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *IEEE Comput.*, vol. 26, pp. 32–41, Mar. 1993.
[22] D. C. Luckham, *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. New York: Springer-Verlag, 1990.
[23] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, "The use of self checks and voting in software error detection: An empirical study," *IEEE Trans. Software Eng.*, vol. SE-16, pp. 432–443, Apr. 1990.
[24] D. E. Perry and W. M. Evangelist, "An empirical study of software interface faults," in *Proc. Int. Symp. New Directions in Comput.*, IEEE Computer Society, Aug. 1985, pp. 32–38.
[25] ——, "An empirical study of software interface faults–an update," in *Proc. 20th Ann. Hawaii Int. Conf. Syst. Sci.*, vol. II, Jan. 1987, pp. 113–126.
[26] C. B. Jones, *Systematic Software Development Using VDM*. Englewood Cliffs, NJ: Prentice-Hall, 1990, 2nd ed.
[27] J. M. Spivey, *The Z Notation: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[28] D. C. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe, *Anna—A Language for Annotating Ada Programs*. Lecture Notes in Computer Science No. 260. New York: Springer-Verlag, 1987.
[29] D. C. Luckham, S. Sankar, and S. Takahashi, "Two-dimensional pinpointing: Debugging with formal methods," *IEEE Software*, vol. 8, pp. 74–84, Jan. 1991.
[30] S. I. Feldman, "Make—A program for maintaining computer programs," *Software—Practice and Experience*, vol. 9, no. 3, pp. 255–265, Mar. 1979.
[31] G. S. Fowler, "A case for make," *Software—Practice and Experience*, vol. 20, no. S1, pp. 35–46, June 1990.
[32] D. W. Flater, Y. Yesha, and E. K. Park, "Extensions to the C programming language for enhanced fault detection," *Software—Practice and Experience*, vol. 23, no. 6, pp. 617–628, June 1993.
[33] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. ACM SIGPLAN'94 Conf. Programm. Language Design and Implementation (PLDI)*, ACM SIGPLAN, June 1994, pp. 290–301, appears in *SIGPLAN Notices* 29(6), June 1994.
[34] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. Winter 1992 USENIX Conf.*, USENIX Association, Jan. 1992, pp. 125–136.
[35] D. S. Rosenblum and B. Krishnamurthy, "An event-based model of software configuration management," in *Proc. 3rd Int. Workshop on Software Config. Management*, P. H. Feiler, Ed., ACM SIGSOFT, June 1991, pp. 94–97.
[36] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *J. Syst. Software*, vol. 4, pp. 289–300, 1984.
[37] A. Endres, "An analysis of errors and their causes in system programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 140–149, June 1975.



David S. Rosenblum (S'83–M'87) received the B.S. *(summa cum laude)* and M.S. degrees, both in computer science, from North Texas State University, Denton, in 1982 and 1983, respectively. He also received the M.S. and Ph.D. degrees in electrical engineering from Stanford University in 1987 and 1988, respectively, where he participated in the Anna and TSL specification language projects.

He is a Member of the Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories, Murray Hill, NJ. His research interests include software testing and analysis, software process, formal specification languages, and specification-based software development tools.

Dr. Rosenblum is a member of ACM, ACM SIGAda, ACM SIGPLAN, ACM SIGSOFT, IEEE Computer Society, and IEEE Computer Society Technical Committee on Software Engineering.