Application of Formal Specification to Software Maintenance

Neel Madhav Sriram Sankar
Stanford University
e-mail: *lastname*@cs.stanford.edu

Abstract

This paper describes the use of formal specifications and associated tools in addressing various aspects of software maintenance — *corrective*, *perfective*, and *adaptive*. It also addresses the refinement of the software development process to build programs that are easily maintainable. The task of software maintenance in our case includes the task of maintaining the specification as well as maintaining the program.

We focus on the use of *Anna*, a specification language for formally specifying *Ada* programs, to aid us in maintaining Ada programs. These techniques are applicable to most other specification language and programming language environments. The tools of interest are: (1) the *Anna Specification Analyzer* which allows us to analyze the specification for correctness with respect to our informal understanding of program behavior; and (2) the *Anna Consistency Checking System* which monitors the Ada program at runtime based on the Anna specification.

1 Introduction

Specification languages present an opportunity to develop new techniques in all phases of software maintenance. These languages provide a capability for expressing, at an abstract level, *what* a program does. *Formal specifications* are expressed in a machine processable form — they can be parsed, checked for static semantic errors, and in many cases, compiled into runtime tests. Typical examples of specification languages developed during the past few years are Anna [6], Larch [3], RAISE [10] and Z [18].

The processes of software development and maintenance iterate over requirements, design, coding, and testing [1, 11, 12, 13]. Most systems have informal requirements, and ad-hoc design, coding, and testing. The specification language, tools and strategies presented in this paper make the processes of software development and maintenance less ad-hoc. The main point of this paper is that effective and powerful techniques can be developed for software maintenance based on formal specifications.

The software maintenance techniques discussed in this paper use *relaxed* formal methods. Relaxed formal methods use formal specifications in a manner which does not insist upon producing provably correct programs. Our strategies for application of formal specification to software maintenance involve informal reasoning and the support of debugging and analysis tools which use specifications.

There are two aspects to designing new software development and maintenance techniques:

1. *developing a new capability.*
   We have designed the language Anna for specifying Ada [19] programs.

2. *designing methods and tools for applying that capability.*
   A methodology for formal specification and implementation of Ada packages is presented in [8]. This allows a user, given a package specification in Ada, to follow a few simple steps to obtain an Anna formal specification and an implementation of the package.

   We have developed a suite of tools for analyzing specifications prior to program development (the *Anna Specification Analyzer*) [9] and for checking the runtime behavior of programs (once they have been developed) for consistency with specifications (the *Anna Consistency Checking System*) [14, 17].

   The Anna Specification Analyzer can be used to develop adequate formal specifications from an informal understanding of the program functionality. This is done by starting with a specification and using the Specification Analyzer to perform symbolic analysis on the specification. The results of this symbolic analysis are then compared to our informal understanding of the program functionality.

   The Anna Consistency Checking System monitors the Ada program at runtime based on the Anna specification. This is done by checking the proposed specification against the behavior of the software. Inconsistencies are reported through a

specialized Anna debugger. A methodology for using this tool to debug formally specified programs is presented in [5].

Our techniques are applicable to all three kinds of software maintenance:

1. *perfective maintenance.*
   Perfective maintenance involves enhancing (or upgrading) an already existing piece of software. Except for the fact that the original software does not handle the proposed enhancements, it has no other known faults. Enhancement may involve extension or modification of specification or code. Typically perfective maintenance will first involve enhancements to the specification. This can be done with the aid of the Specification Analyzer. Once, we have an enhanced set of specifications, we need to enhance the program code based on the new specification. The Consistency Checking System can aid us in achieving this.

2. *corrective maintenance.*
   Corrective maintenance addresses processing, performance, and implementation failures. Corrective maintenance can involve correction of errors either in the specification or in the program.

   A fault in the specification is detected in the following ways: (1) The program behaves as anticipated, but violates a specification — in this case, the violated specification is incorrect; or (2) The program does not behave as anticipated, but does not violate any specification — in this case, the program is certainly incorrect, and either the specification is also incorrect or is incomplete (does not specify against this behavior).

   A fault in the program is detected automatically by the Consistency Checking System. A methodology called *two-dimensional pinpointing* [5] has been developed to proceed from the initial fault detection to actually pinpointing the error and correcting it.

3. *adaptive maintenance.*
   Adaptive maintenance involves modifying software to overcome changes in the processing (both hardware and software) environment of the program. Ada is considered to be quite portable, however, there are many assumptions that Ada programmers make of the underlying software and hardware environments. These assumptions range from quite general (*e.g.*, the Ada compiler being used is correct), to very specific (*e.g.*, the predefined integer type is represented using 32 bits). In general, we write formal specifications that reflect these assumptions we make, and the Consistency Checking System will notify us automatically of any violations of these assumptions. A real-life example of the application of our system was the detection of an Ada compiler bug when porting software from one machine to another [15].

Section 2 is an overview of Anna. Section 3 describes the Specification Analyzer while Section 4 describes the Anna Consistency Checking System. Sections 5, 6 and 7 deal with application of specifications to perfective, corrective and adaptive maintenance respectively. Section 8 discusses some real-life experiences with formal specifications. Section 9 concludes the paper.

2 An Overview of Anna

Anna (**ANN**otated **A**da) is a language extension of Ada [19] to include facilities for formally specifying the intended behavior of Ada programs. In this section a brief outline of a few kinds of annotations is given. A complete definition of Anna is given in [7].

Anna is based on first-order logic and its syntax is a straightforward extension of Ada syntax. Anna constructs appear as *formal comments* within the Ada source text (within the Ada comment framework). Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are *virtual Ada text*, each line of which begins with the indicator `--:`, and *annotations*, each line of which begins with the indicator `--|`.

2.1 Virtual Ada Text

Virtual Ada text is Ada text which appears as formal comments, but otherwise obeys all the Ada language rules. Virtual text may refer to actual text, but is not allowed to affect the computation of the actual program. The purpose of virtual Ada text is to define *concepts*[1] used in annotations that are not explicitly implemented as part of the program. Virtual Ada text may also be used to *compute* values that are not computed by the actual program, but that are useful in defining the behavior of the program.

Example of virtual text:

```
package QUEUE_MANAGER is
   ...
   type QUEUE is private;
   ...
```

[1] Functions used in annotations are called concepts.

```
   --: function IS_MEMBER(E:ELEMENT;Q:QUEUE)
   --:          return BOOLEAN;
   ...
end QUEUE_MANAGER;
```

In the above example, IS_MEMBER is a virtual function. It is used in annotations of actual subprograms of QUEUE_MANAGER.

2.2 Annotations

Annotations are constraints on the underlying Ada program. They provide a capability to express at an abstract level *what* a program does. Anna provides different kinds of annotations, each associated with a particular Ada construct.

Type Annotations: A type or subtype annotation is a constraint on an Ada type. Type annotations are located immediately after the definition of the type they constrain, and are bound to the type definition by the keyword *where*.

Example of a type annotation:

```
type QUEUE is record
   STORE:QUEUE_ARRAY(1..MAX);
   IN_PTR,OUT_PTR:INTEGER range 1..MAX;
   SIZE:INTEGER range 0..MAX;
end record;
--| where Q:QUEUE =>
--|    (Q.IN_PTR-Q.OUT_PTR-Q.SIZE)
--|          mod MAX = 0;
```

The above type annotation constrains all values of the type QUEUE so that their components, IN_PTR, OUT_PTR and SIZE satisfy the equation in the annotation.

There is another form of type annotation — the *modified type annotation*. They constrain the type at the beginning and end of each package operation. The above type annotation is rewritten as a modified type annotation below:

```
--| where in out Q:QUEUE =>
--|    (Q.IN_PTR-Q.OUT_PTR-Q.SIZE)
--|          mod MAX = 0;
```

Object Annotations: An object annotation is a BOOLEAN expression constrains the values of the variables occurring in the expression throughout the scope of the annotation.

Example of an object annotation:

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE) is
   I:INTEGER := Q.OUT_PTR;
   --| if Q.IN_PTR <= Q.OUT_PTR then
   --|    Q.OUT_PTR <= I <= MAX or
   --|    1 <= I <= Q.IN_PTR
   --| else
   --|    OUT_PTR <= I <= IN_PTR
   --| end if;
begin
   ...
end INSERT;
```

In this example, the variable I is constrained to certain ranges depending on the value of Q.

Subprogram Annotations: *Subprogram annotations* are used to describe the behavior of subprograms. A subprogram annotation may specify an input condition, an output condition, or an exceptional condition. Subprogram annotations of functions may specify the value returned by a function.

Examples of subprogram annotations:

```
function IS_FULL(Q:QUEUE) return BOOLEAN;
--| <<SPEC_IS_FULL>>
--| where
--|    return LENGTH(Q) = MAX;

procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| <<SPEC_INSERT>>
--| where
--|    out (LENGTH(Q) = LENGTH(in Q) + 1),
--|    out (IS_MEMBER(E,Q));
```

The first of the above examples is that of a *result annotation*. It specifies that the value returned by the function IS_FULL is the boolean value LENGTH(Q) = MAX. The annotations on the procedure INSERT are *out* annotations. *Out* annotations of INSERT must be satisfied whenever a call to INSERT terminates normally — *i.e.*, terminates without propagating an exception. The modifier *in* on Q specifies that we are referring to the value of Q on entry to INSERT. Therefore, whenever a call to INSERT terminates normally, the length of the resulting Q must be *one* more than the length of the value of Q on entry. Also, on termination, E must be a member of Q.

The above two examples have an extra feature — annotation names. SPEC_IS_FULL and SPEC_INSERT are annotation names in these examples. They are useful in referring to the annotations from the tools.

Statement Annotations: There are two different kinds of statement annotations — simple statement annotations and compound statement annotations.

Simple statement annotations (or *assertions*) are constraints on a single statement.

Example of a simple statement annotation:

```
Q.SIZE := Q.SIZE + 1;
--| Q.SIZE = in Q.SIZE + 1;
```

This annotation says that after the execution of the assignment statement, the new value of Q.SIZE will be 1 more than its previous value.

Compound statement annotations are constraints on compound statements. They constrain all observable states in the compound statement

Example of a compound statement annotation:

```
--| with
--|     Q.OUT_PTR = in Q.OUT_PTR;
begin
    Q.STORE(Q.IN_PTR) := E;
    Q.IN_PTR := Q.IN_PTR mod MAX+1;
    Q.SIZE := Q.SIZE + 1;
end;
```

The annotation above constrains the block statement above (which consists of three assignment statements) to execute in such a way that Q.OUT_PTR remains the same throughout its execution.

Exception Annotations: *Exception annotations* (or propagation annotations) specify the exceptional behavior of subprograms. There are two different kinds of exception annotations — strong propagation annotations and weak propagation annotations.

A *strong propagation annotation* specifies conditions under which exceptions should be propagated. The conditions are with respect to the initial state of the subprogram. If the conditions are satisfied, then the subprogram must terminate by propagating the specified exception.

Example of a strong propagation annotation:

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| where
--|     IS_FULL(Q) => raise FULL;
```

This annotations specifies that if IS_FULL(Q) is true on entry to INSERT, then INSERT must terminate by propagating the exception FULL.

A *weak propagation annotation* specifies what happens when an exception is propagated. It specifies conditions that must be satisfied by the subprogram if a call terminates by propagating one of the specified exceptions.

Example of a weak propagation annotation:

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| where
--|     raise FULL => Q = in Q;
```

This annotation specifies that if the procedure INSERT terminates by propagating the exception FULL, then INSERT does not change the value of Q.

Note that *out* annotations do not specify abnormal termination.

Axiomatic Annotations: *Axiomatic annotations* (or package axioms) are constraints on operations of a package. They must occur in the package visible part. They begin with the keyword *axiom* followed by a sequence of BOOLEAN expressions which are usually quantified with respect to types defined in the package. The complete Anna program shown below includes an example of an axiomatic annotation.

2.3 A Complete Anna Program

We now present an Ada package formally specified using Anna. This package will be used extensively in examples in the later sections. The package implements a *queue* with the typical operations like *create*, *insert* and *remove*. Most of the Anna examples above are from this package.

The Ada package body, including the Ada private part, contains an implementation, the details of which are hidden from users. This hidden part also contains local annotations specifying how the implementation works. The hidden annotations refer to the hidden implementation details.

Some requirements are specified both as subprogram annotations and axiomatic annotations. The reason for this duplication is (1) to illustrate how the same requirement can be specified in more than one way; and (2) some of our later examples use the subprogram annotation version, while others use the axiomatic annotation version.

The package is shown below. Only relevant portions of the package body are included to save space.

```
generic
    type ELEMENT is private;
    MAX : POSITIVE;
package QUEUE_MANAGER is

    type QUEUE is private;
    EMPTY, FULL : exception;

    -- The following are concepts used in specifications.
    --: function IS_MEMBER(E : ELEMENT; Q : QUEUE)
    --:          return BOOLEAN;
    function LENGTH(Q : QUEUE) return INTEGER;
    function IS_EMPTY(Q : QUEUE) return BOOLEAN;
    --| <<SPEC_IS_EMPTY>> where
    --|     return LENGTH(Q) = 0;
    function IS_FULL(Q : QUEUE) return BOOLEAN;
    --| <<SPEC_IS_FULL>> where
    --|     return LENGTH(Q) = MAX;
    function TOP(Q : QUEUE) return ELEMENT;
    --| <<SPEC_TOP>> where
    --|     IS_EMPTY(Q) => raise EMPTY;

    -- The following are operations specified by concepts.
    function CREATE return QUEUE;
    --| <<SPEC_CREATE>> where
    --|     return Q : QUEUE => LENGTH(Q) = 0;

    procedure INSERT(E : ELEMENT; Q : in out QUEUE);
    --| <<SPEC_INSERT>> where
    --|     IS_FULL(Q) => raise FULL,
    --|     raise FULL => Q = in Q,
    --|     out(LENGTH(Q) = LENGTH(in Q) + 1),
    --|     out(IS_MEMBER(E, Q));

    procedure REMOVE(E : out ELEMENT;
                     Q : in out QUEUE);
    --| <<SPEC_REMOVE>> where
    --|     IS_EMPTY(Q) => raise EMPTY,
    --|     raise EMPTY => Q = in Q,
    --|     out(LENGTH(Q) = LENGTH(in Q) - 1),
    --|     out(E = TOP(in Q));

-- Axiomatic annotations follow. In Anna, the attribute
-- 'OUT is a record of all the output values produced by
-- the subprogram.
--| axiom for all E0, E1 : ELEMENT; Q0 : QUEUE =>
--|     REMOVE'OUT(E0,INSERT'OUT(E1,Q0).Q).Q =
--|         INSERT'OUT(E1,REMOVE'OUT(E0,Q0).Q).Q,
--|     LENGTH(INSERT'OUT(E0, Q0).Q) =
--|         LENGTH(Q0)+1,
--|     LENGTH(REMOVE'OUT(E0, Q0).Q) =
--|         LENGTH(Q0)-1,
--|     TOP(INSERT'OUT(E0, Q0).Q) = TOP(Q0),
--|     IS_MEMBER(E0, INSERT'OUT(E0, Q0).Q),
--|     IS_MEMBER(TOP(Q), Q);

private
    type QUEUE_ARRAY is
            array(INTEGER range <>) of ELEMENT;
    type QUEUE is record
        STORE : QUEUE_ARRAY(1..MAX);
        IN_PTR, OUT_PTR : INTEGER range 1..MAX;
        SIZE : INTEGER range 0..MAX;
    end record;
    --| <<QUEUE_INVARIANT>> where
    --|     in out Q : QUEUE =>
    --|         (Q.IN_PTR - Q.OUT_PTR - Q.SIZE)
    --|             mod MAX = 0;
end QUEUE_MANAGER;

package body QUEUE_MANAGER is

    --: function IS_MEMBER(E : ELEMENT; Q : QUEUE)
    --: return BOOLEAN is ... end IS_MEMBER;

    function LENGTH(Q : QUEUE) return INTEGER is
    ... end LENGTH;

    function TOP(Q : QUEUE) return ELEMENT is
    ... end TOP;

    function IS_EMPTY(Q : QUEUE) return BOOLEAN
    is ... end IS_EMPTY;

    function IS_FULL(Q : QUEUE) return BOOLEAN is
    ... end IS_FULL;

    function CREATE return QUEUE is
    ... end CREATE;

    -- The complete implementation of INSERT is included
    -- below since it is used in a subsequent example.
    procedure INSERT(E : ELEMENT;
                     Q : in out QUEUE) is
    --| <<BODY_INSERT>>
    --| where
    --|     out(Q.STORE(in Q.IN_PTR) = E);
    begin
        if IS_FULL(Q) then
            raise FULL;
        end if;
        Q.STORE(Q.IN_PTR) := E;
        Q.IN_PTR := Q.IN_PTR mod MAX+1;
        Q.SIZE := Q.SIZE + 1;
    end INSERT;
```

```
    procedure REMOVE(E:out ELEMENT;
                     Q:in out QUEUE) is
    ... end REMOVE;

end QUEUE_MANAGER;
```

3 The Specification Analyzer

The Specification Analyzer [9] simulates the visible behavior of a package by symbolic execution of Anna specifications. The Specification Analyzer may be used in two ways — to play *mind games* with a specification to get a *preview* of the behavior of the resulting package, and to *debug* the specification itself.

Symbolic execution differs from actual program execution in that there is no implementation of the program. Logical deduction is used to deduce the state of a package and values returned by its subprograms.

The Specification Analyzer is based on a theorem-prover. A model of the *package state* is built as a set of logical expressions concerning the relationships between various functions defined in the package. The user asks questions of this state, such as querying the value of an expression or a function call. The theorem-prover then attempts to deduce answers to queries based on the current state of the package.

The user may declare variables to create an environment for the package being analyzed. Queries made to the Specification Analyzer can be formulated using these variables. The *computation state* is made up of environment variables and the state of the package being analyzed.

A user can interact with the Specification Analyzer in the following ways:

- load a package specification,
- reset the package state to its initial state,
- define a variable (of a specified type),
- execute a procedure (this updates the state),
- find the value of some expression, involving any environment variables and subprograms from the package begin analyzed (this may cause a state update if the function has a side-effect),
- test whether a boolean condition (made up of environment variables and subprograms in the package) is true of the current state,
- test the consistency of the state of the package being analyzed, *or*
- explicitly change the state of the package by asserting that a boolean expression is true of the new state.

Figure 1 illustrates the interaction a user may have with the Specification Analyzer.

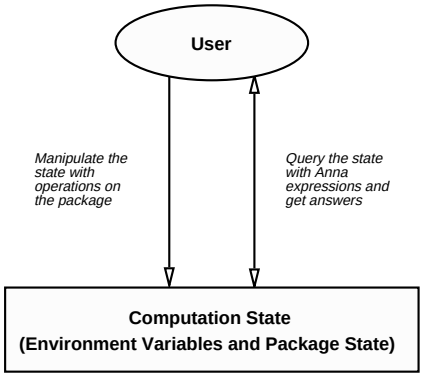

Figure 1: Interaction with the Specification Analyzer.

3.1 A Sample Session

To query the effect of executing the subprogram INSERT on an empty queue with an element E as an argument, the user would take the following steps:

1. Load package QUEUE_MANAGER with ELEMENT = INTEGER and MAX = 10 to get queues of integers of maximum length 10.
2. Declare the variable E : INTEGER.
3. Assign to E some value.
4. Declare the variable Q : QUEUE.
5. Assert the condition IS_EMPTY(Q) = TRUE.
6. Execute the subprogram INSERT(E,Q). This changes the state of the queue package being maintained by the Specification Analyzer.
7. The new state can now be inspected. For example, the user may query what the length of Q is in the new state by querying the value of the expression LENGTH(Q). The Specification Analyzer will print out 1 in this case.

The Specification Analyzer thus allows a user to get a preview of the behavior of the resulting package. The formal specification of the package is treated as a *prototype implementation* of the package.

We now illustrate the detection of an error in the specifications. Assume that the following specification in package QUEUE_MANAGER:

```
--| LENGTH(INSERT'OUT(E0,Q0).Q) =
--|        LENGTH(Q0) + 1
```

was incorrectly written as:

```
--| LENGTH(INSERT'OUT(E0,Q0).Q) =
--|        LENGTH(Q0) - 1
```

The user expects the length to increase as elements get inserted in the queue. However, according to the specification, the length decreases. A simple query which asks the Specification Analyzer for the value of:

```
LENGTH(INSERT'OUT(E,CREATE).Q)
```

would reveal the error in the specification. The user expects the value 1 and the Specification Analyzer returns the value -1. The Specification Analyzer allows the formulation of complicated tests which may reveal more complicated bugs in a specification.

The Specification Analyzer thus allows a user to debug a specification, without any implementation. Any discrepancy between the user's informal requirements and the formal specification will show up when using the Specification Analyzer. It is however, up to the user to design proper test cases to test the specification.

4 The Anna Consistency Checking System

The Anna Consistency Checking System is a set of programs that convert Anna annotations into runtime checking code. This checking code is inserted into the underlying Ada program. When the resulting Ada program is executed, the checking code ensures that any inconsistency in the program with respect to the annotations is detected and reported. The resulting Ada program is linked to a special *Anna debugger*.

To improve the performance of the Anna Consistency Checking System, there is an option to distribute the checking to different processors [16].

When a *transformed* Anna program is executed, the Anna debugger takes control and provides a top-level interface between the user and the program being tested. The debugger provides the following capabilities:

- *Diagnostics.*
  Provides diagnostic messages when the program becomes inconsistent with an annotation. In this case, the annotation violated and the location of violation is displayed to the programmer.
- *Manipulation of annotations.*
  Annotations can be suppressed or unsuppressed, and their effect when they are violated can be changed. For example, annotations can be completely suppressed, *i.e.*, the program will behave as if the annotation were not present.

The programmer interacts with the debugger using menus, choosing displayed options with an input device such as a mouse. The menus display the various options the programmer has in interacting with the debugger. In addition to these menus, there is a window that displays the program execution. When an annotation is violated, two more windows are opened. One of these windows shows the annotation violated while the other shows the local program text around the statement where it was violated. This is illustrated in Figure 2. This Figure illustrates the scenario of the sample session that follows.

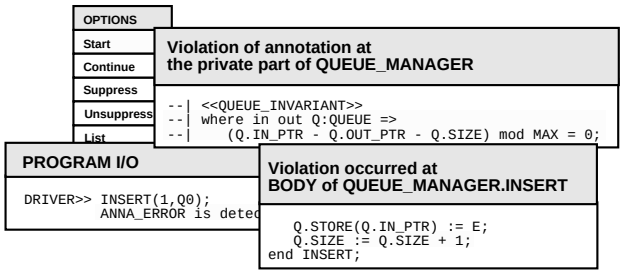

Figure 2: Error Reporting By The Anna Debugger.

4.1 A Sample Session

This session starts with the detection of a fault by the Consistency Checking System. We then proceed methodically to pinpoint the error and correct it. This session consists of performing a *test*, observing the *result*, an *explanation* on how to interpret the result and the actual *action* taken to further pinpoint this error.

The purpose of this session is to demonstrate the use of the Consistency Checking System in pinpointing and correcting errors. A more comprehensive example is given in [5].

Test: Prior to running this test, we introduce an error in the implementation of the INSERT procedure. We delete the line:

```
Q.IN_PTR := Q.IN_PTR mod MAX+1;
```

We then create a mainline for our queue package and transform our program using the Consistency Checking System. We now attempt to perform an INSERT operation on this transformed program after using CREATE to create a new queue.

In this case, our test-data is the sequence of two calls to the queue package: first CREATE and then INSERT. Our tools do not aid us in the selection of test-data. However, a lot of research on test-data generation has been performed (see [2, 4, 20]) which can be used to help in the selection of comprehensive test-data.

Result: The Anna Consistency Checking System detects a violation as shown in Figure 2.

Explanation: This means that after an element was inserted, the invariant condition among Q.IN_PTR, Q.OUT_PTR, and Q.SIZE was violated. There are two possibilities: (1) The queue passed to INSERT was faulty — *i.e.*, CREATE did not perform as expected; and (2) INSERT did not perform as expected.

Action and Justification: We use our judgement and guess that INSERT did not perform as expected. We decide to repair the body of INSERT using a goal-oriented approach. The body of INSERT must achieve the following four goals as a result of updating Q.IN_PTR, Q.OUT_PTR, and Q.SIZE:

```
1. out(Q.STORE(in Q.IN_PTR) = E)
2. out(Q.SIZE = in Q.SIZE + 1)
3. out(Q.IN_PTR = in Q.IN_PTR mod MAX + 1)
4. <<QUEUE_INVARIANT>>
```

Since the fourth goal was violated, the body did not achieve it. A method of fixing the body has to be found so that it achieves the fourth goal. Looking at the body, it is obvious that the body achieves the first and second goals, but not the third one. It can be informally concluded that if an assignment statement is added to satisfy the third goal, then the fourth goal is also met. Hence, the following assignment statement is added to the body of INSERT:

```
Q.IN_PTR := Q.IN_PTR mod MAX + 1;
```

5 Perfective Maintenance

Perfective maintenance involves enhancing functionality, performance (space or time) or maintainability of a program.

There are two ways a program and its specification may need to be changed as part of the perfective maintenance process:

1. *extension.* There may be a need to just *add to* the specification or the program.
2. *modification.*There may be a need to *change* the specification or the program.

Perfective maintenance which enhances the performance of a program requires changing the design and code, but not the specification of the program. Enhancing maintainability of a program involves adding specifications to an unspecified or partially specified program. In both of these cases specifications do not substantially add to the process of change itself. However, the task of testing specifications or programs is facilitated after the change has been made.

We now consider perfective maintenance which involves changing the functionality of a program. The motivating idea is that extensions are relatively easier to handle than modifications. We present strategies using which, given an updated specification, a user may determine whether the implementation needs to be modified or just extended.

The general problem of detecting this for any specifications is intractable. We informally show that if the changes to the specifications are of a certain form, one just needs to extend the implementation.

If the specification of each of the old functions remains the same (the old functions are not redefined), one just needs to extend the implementation. Section 5.1 presents an example of this.

If the specifications of any of the old functions are extended then one *may* need to modify parts of the implementation or just extend the implementation. Section 5.2 is an example of the case where even though specifications of some functions are extended, the implementation need just be extended.

If the specifications of old functions are modified, one needs to (typically) modify the implementation. Section 5.3 is an example of this.

5.1 Extending a Queue

We add a function LAST_ELEMENT to queues. Informally this function would return the element just added to the queue. Formally the specification of the function using axioms may be:

```
function LAST_ELEMENT(Q:QUEUE)
--|                              return ELEMENT;
--| <<SPEC_LAST_ELEMENT>> where
--|     IS_EMPTY(Q) => raise EMPTY;
```

```
...
--| axiom
--|     ...
--|     LAST_ELEMENT(INSERT'OUT(Q0, E0) . Q) =
--|           E0;
```

The axiom says that the element returned is the one just inserted; the exception propagation annotation says that calling the function on an empty queue will result in an exception.

This change need not result in re-coding. All the user needs to do is to implement LAST_ELEMENT by returning the element at IN_PTR in the array used to implement queues.

This change thus was just an extension of both the specification and implementation of queues. The implementation was an extension of the old implementation due to the fact that the addition of LAST_ELEMENT did not change the specification (and thus behavior) of any other subprogram.

5.2 Modifying Queue Specification

We now change queues to double-ended queues. Double-ended queues have a function LAST_ELEMENT as described above and a function REMOVE_LAST which, informally, removes the last element inserted into the queue. Formally the specification of REMOVE_LAST would be:

```
procedure REMOVE_LAST(Q : in out QUEUE);
--| <<SPEC_LAST_ELEMENT>> where
--|     IS_EMPTY(Q) => raise EMPTY;
...
--| axiom
--|     ...
--|     IS_MEMBER(E, REMOVE_LAST'OUT(
--|                   INSERT'OUT(Q0, E0) . Q))
--|     = IS_MEMBER(E, Q0),
--|     TOP(REMOVE_LAST'OUT(
--|           INSERT'OUT(Q0, E0) . Q))
--|     = TOP(Q0),
--|     REMOVE(REMOVE_LAST'OUT(
--|              INSERT'OUT(Q0, E0) . Q))
--|     = REMOVE(Q0),
--|     LENGTH(REMOVE_LAST'OUT(Q))
--|     = LENGTH(Q) - 1;
```

In other words, REMOVE_LAST has the opposite effect of INSERT. In this example, the specification of functions TOP, IS_MEMBER, REMOVE and LENGTH has been extended. All these functions must now be applicable to queues to which REMOVE_LAST has been applied. However, the implementation of queues just needs to be extended; the implementation just needs to decrement IN_PTR. This is because no new queues are created by application of REMOVE_LAST to queues. Any queue which is constructed by application of REMOVE_LAST can also be constructed by using just CREATE and INSERT.

5.3 Modifying Queues Again

We now change the functionality of the function REMOVE to remove elements from the back of the queue instead of the front. The function REMOVE thus behaves exactly like the function REMOVE_LAST. It is quite obvious that the specification and the implementation of REMOVE need to be modified.

6 Corrective Maintenance

Corrective maintenance involves taking care of processing, performance or implementation failures. This can involve either correction of errors in the specification or in the program.

Errors detected in the specification. A fault in the specification is detected in the following ways: (1) The program behaves as anticipated, but violates a specification — in this case, the violated specification is incorrect; or (2) The program does not behave as anticipated, but does not violate any specification either — in this case, the program is certainly incorrect, and either the specification is also incorrect or is incomplete (does not specify against this behavior).

In such a situation, we correct the specification first. We modify the specification based on the error we observe. We now apply the Specification Analyzer on this new specification keeping in mind the error we detected. This will ensure that the specification has indeed been corrected appropriately.

We now apply the Consistency Checking System on the program with the modified specification. If the program did not behave as anticipated earlier, the Consistency Checking System will now detect an anomaly. The paragraphs below explain what is done in this situation.

Errors detected in the program. This happens because of either an error in the program, or in the environment in which the program is running. In either case, the Consistency Checking System reports an inconsistency. In Section 4, we have already demonstrated one possible approach to pinpointing and correcting the error. A real-life experience of using the

Consistency Checking System in going one step further to locate an Ada compiler bug is described in [15].

The above techniques of corrective maintenance can equally well be applied to earlier phases in the software development life-cycle. In software maintenance, their usefulness is made more significant by the fact that it is not always the case that the software developer is also the maintainer of the software. It is also possible that the software developer has forgotten the vast amounts of assumptions made in a large software project.

7 Adaptive Maintenance

Adaptive maintenance involves modifying software to overcome changes in the processing (both hardware and software) environment of the program. To solve this problem, the programmer must define precise interfaces to implementation dependent code. This interface should, however, be implementation independent. The idea being that when a change in the environment is made, only the implementation of this interface needs to be changed. In addition to building such a interface, it helps to formally specify this interface, clearly laying out the assumptions that are being made of the underlying environment. Examples of such specifications appear below. Violations with respect to such specifications indicate a mismatch between what is expected of the environment and the actual environment itself.

Ada has been written keeping portability in mind. In spite of the efforts made by the language design team, there are a variety of reasons why Ada programs are not portable:

Errors in the new environment. The program may not run in the new environment if this new environment contains errors. This may be due to errors in the compiler, runtime system, or even a hardware or operating system error. The usefulness of specifications in detecting such errors is clearly indicated in [15], which describes the detection of a compiler bug when a certain piece of software was ported from a Sun workstation to a Sequent Symmetry machine.

Size of predefined types. Some commonly available Ada compilers represent the predefined type INTEGER using 16 bits, while others use 32 bits. When a program is ported from a 32 bit machine to the 16 bit machine it may stop working. A formal specification that states our assumption regarding the sizes of predefined types can ease the process of detecting such errors during the porting process. An example of such a specification is shown below:

```
--| INTEGER'FIRST <= -2**16+1,
--| INTEGER'LAST >= 2**16-1;
```

An alternate solution that will work in most situations is to define new types and not rely on the bounds of predefined types[2]. For example, we could replace the above specification by:

```
type MY_INTEGER is range -2**16+1..2**16-1;
```

and all occurrences of INTEGER in the program by MY_INTEGER.

Erroneous programs. There are many situations in Ada where certain implementation details are left upto the compiler. If a program is written in such a way that it can behave differently depending on the choice of implementation made by the compiler, this program is considered erroneous. An example of an erroneous program is one whose execution depends on the order of evaluation of the two sides of an assignment statement. For example, if FIND and INSERT are two operations in a symbol table package, then the following statement may be erroneous:

```
FIND("FOO").SIZE :=
        COMPONENT_SIZE_SUM(INSERT("FOO"));
```

Here, we assume that the right hand side is evaluated first. Erroneous programs should not be written in the first place, but mistakes are made especially when developing very large programs. Specifications about the intended behavior of the portion of the program containing the above assignment specification will usually help in detecting the error when this program is compiled to evaluate the left-hand side of assignment statements first.

Foreign language interfaces. When an Ada program is linked with programs written in other languages, we need to ensure that data structures are passed correctly between code written in the different languages. Formal specifications that describe how the foreign language interface should behave ease the detection of any potential problems when, for example, the Ada program calls C routines, and the C compiler is replaced by another C compiler with different storage mechanisms.

[2] We are forced to use predefined types in Ada in some situations. For example, the package TEXT_IO uses the type STRING whose index type is INTEGER.

8 Real-Life Experiences

The use of even extremely trivial annotation constructs has helped us greatly in the maintenance process. We have many large pieces of software that have been lying around for many years, but get periodically upgraded. Many times, we have been warned of inconsistent upgrades as a result of violations with respect to very simple annotation constructs. An example of a commonly occurring annotation construct is an assertion about the value of a pointer variable:

```
--| PTR /= null;
```

The above assertion specifies formally an assumption that the pointer variable PTR references a data structure at this point. Any upgrade that causes PTR to be *null* at this point will immediately cause the reporting of an inconsistency.

One of the authors has developed an overload resolution package for Ada/Anna expressions. The size of this package is around 3000 lines, and contains many annotations specifying assumptions as complex as the overall functionality of the overload resolution package, to assumptions as simple as assertions about a particular program state. Many of these annotations are more than three years old. These annotations have proved to be invaluable in the maintenance of the overload resolution package especially since violations are reported automatically and also since the author has forgotten many of the assumptions he made.

9 Conclusions

The advantages of software maintenance using formal specifications, as exemplified by our methodology and tools, over present software maintenance methods and tools include:

- *The program requirements are stated formally.*
  The set of specifications to be tested at each level of program structure constitutes a formal definition of the behavior to be tested at that level. Consequently, the error detection process may involve several people with understandings of different levels and components of the software. A systems designer may, for example work with high level specifications, and hand over a precise problem — "this specification of that component is violated by this test sequence" — to an expert in a particular component.

- *Faults in the specification can be detected methodically.*
  Currently, there are very few tools that allow us to analyze specifications symbolically. The Specification Analyzer permits us to play mind-games with the specification. The results of these mind games are facts about the program execution that can be determined from the specification. Using this information, it is usually quite easy to detect faults in specifications.

- *Faults in the program are detected automatically.*
  Current debuggers gather information from the lowest implementation levels of a program and its runtime environment. The programmer must deduce what is happening from this information. Some more advanced debuggers can test boolean assertions about program variables. The Consistency Checking System, on the other hand, provides the power to test very general constraints on packages, abstract types, data structures, and subprograms. The task of searching output traces in order to recognize errors is eliminated.

- *Very complex tests can be formulated easily.*
  For example, specifications against side-effects on global data are easily formalized and are then tested automatically. Typical one-line abstract specifications that are checked by our tools could require manual insertion of large numbers (possibly hundreds) of breakpoints, print statements, or assertions, in order to be checked by current kinds of debuggers.

- *Our methods and tools are independent of the language implementation.*

- *Our methods apply to other development processes in the software life-cycle.*
  The techniques used in software maintenance can be applied earlier in software development, for example in analyzing simulations and in using prototypes to develop formal standard specifications for module interfaces. We have also mentioned earlier that techniques developed primarily for debugging can be easily adapted to software maintenance.

10 Acknowledgements

We would like to thank Prof. David Luckham at Stanford for his guidance and support for writing this paper. Walter Mann was the primary developer of the Specification Analyzer. The debugging methodology was developed along with Dr. Shuzo Takahashi and Prof. David Luckham.

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

References

[1] V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, January 1990.

[2] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings, 1987.

[3] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[4] W. C. Hetzel, editor. *Program Test Methods.* Series in Automatic Computation. Prentice-Hall, 1973.

[5] D. C. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: An application of formal specification to debugging packages. Technical Report CSL–TR–89–379, Stanford University, April 1989. To appear in IEEE Software.

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

[7] D. C. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *Anna — A Language for Annotating Ada Programs.* Springer-Verlag — Lecture Notes in Computer Science No. 260, July 1987.

[8] N. Madhav and W. Mann. A methodology for formal specification and implemantation of ada packages, 1990. To appear in COMPSAC 1990, Chicago.

[9] W. R. Mann. Anna package specification analyzer user's guide. Unpublished technical report, 1990.

[10] M. Nielsen, K. Havelund, K. R. Wagner, and C. George. The RAISE language, method and tools. In *Proceedings of the VDM Conference*, pages 376–405. Springer-Verlag — Lecture Notes in Computer Science No. 328, 1988.

[11] W. M. Osborne and E. J. Chikofsky. Fitting pieces to the maintenance puzzle. *IEEE Software*, January 1990.

[12] G. Parikh. The world of software maintenance. In G. Parikh and N. Zvegintzov, editors, *Tutorial on Software Maintenance.* IEEE Computer Society, 1983.

[13] M. G. Rekoff Jr. On reverse engineering. *IEEE Transactions on Systems, Man, and Cybernetics*, March-April 1985.

[14] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[15] S. Sankar. A note on the detection of an Ada compiler bug while debugging an Anna program. *ACM SIGPLAN*, 24(6):23–31, 1989.

[16] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. Submitted for publication in IEEE Computer., 1990.

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

[18] J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics.* Cambridge Unversity Press, 1988. Tracts in Theorectical Computer Science, Volume 3.

[19] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, February 1983. ANSI/MIL-STD-1815A-1983.

[20] R. T. Yeh, editor. *Current Trends in Programming Methodology, Volume 2 — Program Validation.* Prentice-Hall, Inc., 1977.