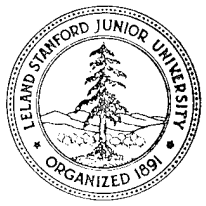
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055



THE APPLICATION OF FORMAL SPECIFICATIONS TO SOFTWARE DOCUMENTATION AND DEBUGGING

Anoop Goyal
Sriram Sankar

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Technical Note: CSL-TN-93-392

(Program Analysis and Verification Group Note No. 63)

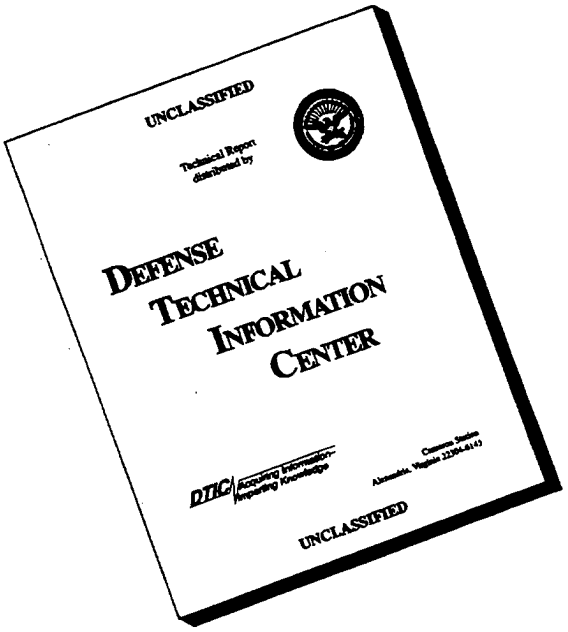
April, 1993

19960729 097

DTIC QUALITY INSPECTED 3

This research was supported by the U.S. Defense Advanced Research Projects Agency/Information Systems Technology Office Contract N00039-91-C-0162.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

The Application of Formal Specifications to Software Documentation and Debugging

Anoop Goyal Sriram Sankar

Technical Note: CSL-TN-93-392
Program Analysis and Verification Group Note No. 63

April 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

This paper illustrates the application of formal specifications to software documentation and debugging by presenting a real-life scenario involving the use of a garbage collection package. It illustrates the advantages of using formal specifications over informal documentation. The paper also illustrates the usefulness of run-time checking tools that compares program behavior with their formal specifications.

The scenario presented in this paper goes through a series of steps that include formal specification, run-time checking, and modification of the specification and program based on the results of run-time checking — the typical steps involved in a debugging process, except that this scenario makes use of formal specifications.

Although various research ideas presented in this paper have been published earlier, this paper assimilates all these ideas into a real-life scenario, and illustrates in an easy-to-understand way that these ideas are really useful to software documentation and debugging.

The example has been developed in Ada, and formally specified using the Anna specification language. The tool used in the example is the Anna Run-Time Consistency Checking System developed at Stanford University.

Key Words and Phrases: algorithmic debugging, Anna, formal specification, formal documentation, run-time consistency checking, software testing.

Copyright © 1993
by
Anoop Goyal Sriram Sankar

Contents

DTIC QUALITY INSPECTED 3

1 Introduction

Formal specifications have been used in many ways in the software development life-cycle for quite some time. Initially, formal specifications were primarily used for program verification [1, 8], where the goal was to prove that a program would run in a manner consistent with its formal specification. Subsequently, formal specifications have been used for other purposes, such as formal documentation [10, 12], requirements analysis, automatic program generation [3, 7, 16], testing (*e.g.*, black-box testing) [2, 6], and run-time monitoring [5, 23, 24]. These applications have resulted in the development of many specification languages such as Anna [10, 12], Larch [4], and Z [22], and tool-sets to support their use.

The Anna specification language was designed by the Program Analysis and Verification Group at Stanford University during the early 80's. The group has also developed a tool-set for Anna that includes a specification analyzer [15] and a run-time monitoring system [17, 19, 20]. We have subsequently worked on a number of applications of Anna and its tool-set which includes: specification methodology and requirements analysis [11, 13]; testing and debugging [9, 18, 21]; and software maintenance [14].

This paper describes a real-life example of the use of the various applications of formal specifications, especially algorithmic debugging [9], described in the earlier publications. In addition, this paper also illustrates the use of formal specifications in software documentation and requirements analysis.

The primary purpose of this paper is to illustrate to the reader that there is a real use for formal methods in the sofware development process. This paper does this by assimilating many of the lessons learned from earlier experiments into this real-life scenario in a simple and easy-to-understand manner.

This example involves the use of a garbage-collection package in a large software project. The garbage collection package was developed in 1985 for use with Ada objects. It has turned out to be very efficient and has been used extensively. However, this package was only informally documented using English comments (*i.e.*, no formal specifications). As the example in this paper will illustrate, the absence of formal specifications turned out to be a serious deficiency in the long run, resulting in wastage of programmer time.

In the example presented in this paper, an application program that uses the garbage collection package was taken through a debugging phase, during which doubts were raised about the correct working of the garbage collection package. Since the application developer did not have a detailed understanding of the implementation of the garbage collection package, and given the large size of the application program, the developer could not narrow down the problem further using traditional debugging techniques. He therefore wrote a formal specification for the garbage collection package in Anna, and used the Anna run-time monitoring system to determine whether or not the implementation of the garbage collection package was consistent with his formal specification. As expected, the run-time monitoring system detected an inconsistency.

This started an interaction between the developer and the implementor of the garbage collection package[1]. The implementor studied the results of the tests conducted by the developer and realized

[1] For convenience, we shall henceforth use the term "developer" for the person who wrote the application program, and "implementor" for the person who wrote the garbage collection package.

that the formal specification written by the developer did not correctly describe the intended behavior of the garbage collection package. The implementor, therefore, rewrote the formal specification to reflect more accurately the intended behavior of the garbage collection package. The ramification of this was that *the developer had not understood the proper behavior of the garbage collection package and was therefore using the package wrongly.* Given the size of the application program, the task of correcting the application program was quite large. However, when the application was run with the Anna run-time monitoring system using the revised specification of the garbage collection package, errors were reported that quickly revealed most locations where corrections needed to be made in the application.

A moral of this story is that the early use of formal specifications in documenting software is crucial to the correct understanding of the behavior of various components in the software system. Incorrect assumptions made about some components can cause significant wastage of programmer time. This example indicates that the informal English comments were imprecise — for both the developer's and the implementor's formal specifications were consistent with the English comments, but inconsistent with respect to each other.

This example also illustrates the use of formal specifications as:

- A *requirements analysis* tool. The Anna specification of the garbage collection package acted as a precise communication medium between the developer and implementor. The use of Anna immediately revealed to the implementor the misunderstanding the developer had of the working of the garbage collection package. Informal documentation and informal communication had obviously not worked. Informal interaction between the developer and implementor had failed even after the developer started to suspect a problem with the garbage collection package. This was partly due to the size of the application and the large number of calls it made to the garbage collection package, which resulted in the problem getting concealed in other extraneous details. The use of formal specifications allowed the developer and implementor to organize their thoughts properly and thus have an effective communication.

- A basis to perform *run-time testing and debugging.* In the example presented in this paper, just writing formal specifications may not have sufficed. It was also important to obtain evidence that the implementation of the garbage collection package/application was not behaving consistently with the specifications. This evidence was used in the first phase of this example to convince the implementor that something may be wrong with the implementation of the garbage collection package, and in the second phase to modify the application to correctly use the garbage collection package.

Outline of this paper. The paper proceeds by presenting an overview of the Anna specification language and the run-time monitoring system in Section 2. Section 3 describes the garbage collection package. The interaction between the developer and the implementor and the detection of various problems in the application and garbage collection package is described in the next three sections — Section 4 describes the developer's attempt at writing a formal specification for the garbage collection package. This is completed by the implementor in Section 5. Section 6 describes how the problem was eventually corrected. Finally, Section 7 concludes the paper.

2 Overview of Anna

Anna [10, 11, 12] (ANNotated Ada) is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. Anna was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs.

Anna is based on first-order logic and its syntax is a straightforward extension of the 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 appearing as formal comments, but otherwise obeying all of the Ada language rules. Virtual Ada text may refer to the underlying Ada program, but is not allowed to affect its computation. The purpose of virtual Ada text is to define concepts 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 underlying Ada program, but that are useful in specifying the behavior of the program.

2.2 Annotations

Annotations are constraints on the underlying Ada program. They are comprised of expressions that are boolean-valued. The location of an annotation in the Ada program together with its syntactic structure indicates the kind of constraints that the annotation imposes on the underlying program. Anna provides different kinds of annotations, each associated with a particular Ada construct. Some examples of annotations are subtype annotations, object annotations, statement annotations, subprogram annotations, exception propagation annotations, and axiomatic annotations. Subprogram annotations are explained in greater detail because they are used in this paper.

Subprogram Annotations

Subprogram annotations follow Ada subprogram declarations. They are constraints on the formal parameters and results of subprogram calls. They may specify conditions under which exceptions are propagated. A subprogram annotation must be true of every call to the subprogram and it acts as a declarative constraint over the subprogram body. An example follows:

```
procedure Increment(X : in out Integer);
--| where
--|    out(X = in X + 1);
```

In this example, the subprogram annotation of the procedure **Increment** constrains the value of the parameter X on return to be one greater than its value when called.

2.3 The Anna Run-Time Monitoring System

The Anna run-time monitoring system is a set of programs that convert Anna annotations into run-time checking code. This checking code is inserted into the underlying Ada program. The resulting Ada program is linked to a special *Anna debugger.*

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. Control can be transferred to the underlying program in which case, control returns to the debugger when the program becomes inconsistent with some annotation. 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 annotations were not present.

3 The Garbage Collection Package

```
generic
    type Item is limited private;
    type Link is access Item;
package Garbage_Collection is
    procedure Free(Item_Pointer : in out Link);
    --| out (Item_Pointer = null);
    -- If Item_Pointer is null, then do nothing. This procedure may raise storage_error.
    procedure Get(New_Item_Pointer : in out Link);
    --| out (New_Item_Pointer /= null);
    -- If New_Item_Pointer is not null then do nothing. This procedure may raise storage_error.
end Garbage_Collection;
```

Figure 1: The Garbage Collection Package Specification

The garbage collection package specification with its original documentation is shown in Figure 1. It is a generic Ada package that is parameterized with two type parameters — **Item** and **Link**. Here **Item** can be any type, and **Link** is an access type whose elements are pointers to elements of type **Item**. The garbage collection package exports two operations — **Get** and **Free**.

Ada provides the **new** operation as the only primitive for memory allocation. The language does not guarantee any memory deallocation capabilities. Hence the need for such a package. **Get** in

the garbage collection package has pretty much the same semantics as Ada's **new** operation, and Free is a deallocation operation. If the particular Ada implementation does provide a deallocation operation as a primitive, Free may use it. Otherwise, the garbage collection package may implement a data structure in which it stores free'd items for later reallocation.

Implementation of the garbage collection package. The particular implementation of the garbage collection package that we used in our experiment assumed that the underlying Ada implementation did not provide a deallocation primitive. It therefore collects all free'd items into a data-structure for later reallocation by Get.

Some details of the garbage collection package implementation are mentioned below. This portion may be skipped during the first reading of the paper. The complete garbage collection package implementation is shown in Figure 2.

The garbage collection package maintains two lists: Free_Nodes_With_Free_Items and Free_Nodes_With_No_Items. Free_Nodes_With_Free_Items is a list containing pointers to the items which were allocated at some time but are now free. That is, Free_Nodes_With_Free_Items is a list of pointers. Storing these pointers takes some space. In the program, it is the structure type called Node_Type, which stores these pointers. In further discussion, we will refer to this as the *base* (which can be used to store pointers to items). Free_Nodes_With_No_Items is a list of *base*'s.

Initially both the lists are empty. When Get is called, it checks whether Free_Nodes_With_Free_Items is empty. If so, it invokes the Ada **new** operation to obtain more memory from the operating system. If Free_Nodes_With_Free_Items is not empty the first pointer to an item is taken from this list and returned to the user. Also, the *base* storing this pointer is removed from Free_Nodes_With_Free_Items and added to Free_Nodes_With_No_Items which allows this *base* to be reused later.

When Free is called, it checks if Free_Nodes_With_No_Items is empty. If so, in invokes the Ada **new** operation to obtain a new *base* from the operating system. Otherwise, a *base* is removed from Free_Nodes_With_No_Items and used to store a pointer to the freed item.

4 Developer's Specification of Garbage Collection Package

The developer had developed an application that made use of the garbage collection package. The garbage collection package was informally specified in English just as shown in Figure 1. The implementor assumed that the behavior of the garbage collection package was quite obvious and had therefore not bothered to specify it in any more detail. And in fact, it had been used for more than six years in various applications with no problem.

However, the developer's application had a problem that the developer felt may be due to the garbage collection package. Due to the large size of the application, and the amount of time it had to be run before the problem showed itself, the developer found it very difficult to apply traditional debugging techniques. He therefore wrote a formal specification for what he assumed was the behavior of the garbage collection package. This specification is shown in Figure 3.

The developer has defined a concept Allocate_Set using virtual Ada text. For this purpose, the developer has also made use of an off-the-shelf sets package. Allocate_Set is the set of all items that

```
package body Garbage_Collection is

    type Node_Type;
    type List_Type is access Node_Type;
    type Node_Type is record
        L : Link;
        Next : List_Type;
    end record;
    Free_Nodes_With_No_Items, Free_Nodes_With_Free_Items : List_Type;

    procedure Free(Item_Pointer : in out Link) is
        Temp : List_Type;
    begin
        if Item_Pointer /= null then
            if Free_Nodes_With_No_Items = null then
                Temp := new Node_Type;
            else
                Temp := Free_Nodes_With_No_Items;
                Free_Nodes_With_No_Items := Free_Nodes_With_No_Items.Next;
            end if;
            Temp.all := (L => Item_Pointer, Next => Free_Nodes_With_Free_Items);
            Free_Nodes_With_Free_Items := Temp;
            Item_Pointer := null;
        end if;
    end Free;

    procedure Get(New_Item_Pointer : in out Link) is
        Temp : List_Type;
    begin
        if New_Item_Pointer = null then
            if Free_Nodes_With_Free_Items = null then
                New_Item_Pointer := new Item;
            else
                Temp := Free_Nodes_With_Free_Items;
                Free_Nodes_With_Free_Items := Free_Nodes_With_Free_Items.Next;
                New_Item_Pointer := Temp.L;
                Temp.all := (L => null, Next => Free_Nodes_With_No_Items);
                Free_Nodes_With_No_Items := Temp;
            end if;
        end if;
    end Get;

end Garbage_Collection;
```

Figure 2: Garbage Collection Package Implementation

```
--: with Sets;
generic
   type Item is limited private;
   type Link is access Item;
package Garbage_Collection is

   --: function "<"(X,Y:Link) return Boolean;
   --: package Link_Set is new Sets(Link, "<");
   --: use Link_Set;
   --: Allocate_Set:Set := Init;
   -- Allocate_Set is the set of pointers to all allocated items at any time.

   procedure Free(Item_Pointer:in out Link);
   --| where
   --|     out(Item_Pointer = null),
   --|     out(not Is_Member(Allocate_Set,in Item_Pointer));

   procedure Get(New_Item_Pointer:in out Link);
   --| where
   --|     out(New_Item_Pointer /= null),
   --|     out(Is_Member(Allocate_Set, New_Item_Pointer)),
   --|     out(not Is_Member(in(Copy(Allocate_Set)), New_Item_Pointer));

end Garbage_Collection;
```

Figure 3: Developer's Specification

are currently allocated (*i.e.*, they have not been freed). The set stores items using pointers to these items, and its initial value is the empty set since no items are allocated when the program starts execution.

The specification of Free states that on completion, Item_Pointer will be **null** and that the item pointed to by the initial value of Item_Pointer is no longer a member of Allocate_Set.

The specification of Get states that on completion, New_Item_Pointer will not be **null**, that New_Item_Pointer will be a member of Allocate_Set, and finally, that New_Item_Pointer was not a member of the Allocate_Set when the procedure Get was called.

The following points should be noted about the above specification:

- In the very last line of the specification of Get, the function Copy has been used. This is because of the ambiguous nature of the expression (**in** Allocate_Set). If Allocate_Set is implemented as a pointer-based data structure, (**in** Allocate_Set) basically captures the initial value of only the pointers. If the data being pointed to has been changed during execution of Get, (**in** Allocate_Set) also changes value. Hence a fresh copy of Allocate_Set (made in the initial state) is used in the specification.
- The specification talks about how Allocate_Set changes when Free and Get are executed. However, Allocate_Set is a virtual Ada text entity that the implementation of these procedures has no knowledge of. Therefore, we must insert virtual Ada text into the implementation of these procedures to perform the necessary operations on Allocate_Set whenever the "real" allocations and deallocations are performed within the body. Figure 4 shows these virtual Ada text insertions.
- A sets package has been used to obtain all the set concepts. The developer has chosen an off-the-shelf sets package that was developed in 1982 and has been in frequent use since then. The point being that the developer wants to have a very high level of confidence in the correct implementation of the sets package so that he can concentrate his attention on the rest of his code.

Execution with run-time monitoring of specifications. The developer transformed the Anna constructs into checking code using the Anna tools and executed his application. After some time, the following annotation was violated during a call to Get:

```
--| out (not Is_Member(in (Copy(Allocate_Set)), New_Item_Pointer))
```

This means that Get was returning a value in New_Item_Pointer that was already present in Allocate_Set — *i.e.*, Get was returning an already allocated item[2].

[2]There is also the possibility that the sets package has a bug, the Anna specification of the garbage collection package is wrong, or that the virtual Ada code in the body of the procedures has not been inserted correctly. However, the developer has decided to assume (for the time-being) that such problems do not exist.

```
package body Garbage_Collection is

   ...

   procedure Free(Item_Pointer:in out Link) is
      Temp:List_Type;
   begin
      if Item_Pointer /= null then
         --: Link_Set.Remove(Allocate_Set, Item_Pointer);
         ...
      end if;
   end Free;

   procedure Get(New_Item_Pointer:in out Link) is
      Temp:List_Type;
   begin
      if New_Item_Pointer = null then
         ...
         --: Link_Set.Insert(Allocate_Set, New_Item_Pointer);
      end if;
   end Get;

end Garbage_Collection;
```

Figure 4: Virtual Ada Text Insertions

5 Implementor's Specification of Garbage Collection Package

The developer informed the implementor of his findings. When the implementor read the developer's formal specification, he found it to be incomplete. The developer had not specified against possible misuse of the garbage collection package. Being the implementor of the garbage collection package, he assumed that the problem was not in his code! The implementor completed the specification of the garbage collection package as shown in Figure 5.

The implementor decided to define another concept using virtual text — the set Free_Set. Free_Set contains all the items that have been allocated at some earlier time, but are currently free'd. This set is also initialized to be the empty set.

The implementor has specified two cases of misuse of the garbage collection package using the two annotations starting with **in** in Free. They state that when Free is called, the parameter Item_Pointer passed to it must *(1)* be a member of Allocate_Set, and *(2)* not be a member of Free_Set.

The implementor has also specified how Free and Get affect Free_Set. The specification states that on return from Free, Item_Pointer must be a member of Free_Set; and that on return from Get, New_Item_Pointer must not be a member of Free_Set and furthermore, if Free_Set was not empty when Get was called, the returned New_Item_Pointer must be from this set.

Here again, the implementation of the garbage collection package has to be instrumented to update Free_Set, just as in the case of Allocate_Set.

```
--: with Sets;
generic
    type Item is limited private;
    type Link is access Item;
package Garbage_Collection is

    --: function "<"(X,Y:Link) return Boolean;
    --: package Link_Set is new Sets(Link, "<");
    --: use Link_Set;
    --: Free_Set, Allocate_Set:Set := Init;
    -- The implementor has defined yet another concept — Free_Set. Free_Set is the set of all items
    -- which were sometime allocated but are currently free.

    procedure Free(Item_Pointer:in out Link);
    --| where
    --|     in(Is_Member(Allocate_Set, Item_Pointer)),
    --|     in(not Is_Member(Free_Set, Item_Pointer)),
    --|     out(Item_Pointer = null),
    --|     out(not Is_Member(Allocate_Set, in Item_Pointer)),
    --|     out(Is_Member(Free_Set, in Item_Pointer));

    procedure Get(New_Item_Pointer:in out Link);
    --| where
    --|     out(New_Item_Pointer /= null),
    --|     out(Is_Member(Allocate_Set, New_Item_Pointer)),
    --|     out(not Is_Member(in(Copy(Allocate_Set)), New_Item_Pointer)),
    --|     out(not Is_Member(Free_Set, New_Item_Pointer)),
    --|     out(in(not Is_Empty(Free_Set)) ->
    --|             (Is_Member(in(Copy(Free_Set)), New_Item_Pointer)));

end Garbage_Collection;
```

Figure 5: Implementor's Specification

Execution with run-time monitoring of specifications. When the transformed version of this program was executed, the following annotation was violated during a call to Free:

```
--| in(Is_Member(Allocate_Set, Item_Pointer))
```

This means that the item being free'd was not in Allocate_Set. Among other possibilities, this indicated that the developer may be calling Free with an item that was already deallocated. When the implementor informed the developer of this, he said that this was indeed the case and that he (the developer) assumed that the garbage collection package could handle multiple consecutive free requests on the same item just as the corresponding UNIX system call did. The implementor's specification clearly mentions that a free'd item may not be freed a second time. However, he failed to write this down in his initial informal specification of the garbage collection package. This misunderstanding between the implementor and the developer, which caused a lot of wasted effort (there is still the issue of correcting the problem now that it has been detected) could have been avoided by enforcing the use of formal specifications in the first place.

6 Correcting the Problem

There were two alternatives to make the garbage collection package and the program compatible with each other. The first was to tailor the garbage collection package to suit the needs of the program. This was possible with only a few changes in the package implementation, and hence, very tempting. But it would seriously affect the time complexity of garbage collection. The current version of the package had a constant time complexity, whereas the modified package would have a time complexity linear on the number of items allocated. This was deemed undesirable.

The second alternative (which was chosen as the solution to the problem) was to change the application to avoid deallocating the same item twice. This would be difficult to do since deallocations were being performed all over the application in non-trivial ways. Here again, Anna and the Anna run-time monitoring system came to use. The strategy followed by the developer was simple: Execute the application using the garbage collection package specified by the implementor (Figure 5) with the Anna checks enabled. Whenever a deallocation to the same item was repeated, the Anna debugger would report a violation. The location of the violation enabled the developer to fix one particular misuse of the garbage collection package. Repeating this process slowly eliminated most of these problems.

However, after a while, a new problem was encountered. The application would be run for a long time and would run out of memory because the implementation of the sets package did not do its own garbage collection. The implementor remedied this problem by rewriting the specification without using the sets package, but rather using the data structures within the garbage collection package to simulate the required sets. This version of the garbage collection package and its implementation is shown in Appendix A.

With this new version, the developer managed to get rid of all multiple deallocations in a relatively short time.

7 Conclusion

Writing specifications for any program, however trivial it might look at the time, is very important. It is of course best to annotate the program completely, but if that is not feasible, it is good to write at least some annotations. There might not be too much difference between non-rigorous and rigorous specifications, but there is a very large difference between non-rigorous specifications and no specifications at all. Writing specifications is neither difficult nor time-consuming, it is just a matter of habit, which needs to be cultivated, just like good programming style.

8 Acknowledgements

Doug Bryan was the "implementor" and James Vera was the "developer". The authors aided Doug and James in the use of Anna and its tool-set. The Anna project has been led by Prof. David Luckham, who has been a source of inspiration for many of our achievements in the application of formal methods to software develoment.

This work was primarily supported by DARPA through ONR N00014-90-J-1232. During the final phases of this work, Anoop Goyal was supported by the Department of Computer Science at Stanford University and Sriram Sankar was supported by Sun Microsystems Laboratories, Inc.

References

[1] R. W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society*, volume 19, pages 19–32. American Mathematical Society, 1967.

[2] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, April 1975.

[3] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge based software assistant. Technical report, Kestrel Institute, 1983.

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

[5] D. P. Helmbold and D. C. Luckham. Runtime detection and description of deadness errors in Ada tasking. Technical Report 83-249, Computer Systems Laboratory, Stanford University, November 1983. (Program Analysis and Verification Group Report 22).

[6] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10:53–66, 1978.

[7] B. Krieg-Brückner. Transformation of interface specifications, 1985. PROSPECTRA Study Note M.1.1.S1-SN-2.0.

[8] R. L. London. A view of program verification. In *Proceedings of the International Conference on Reliable Software*, pages 534–545, April 1975.

[9] D. C. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: An application of formal specification to debugging packages. *IEEE Software*, 8(1):74–84, January 1991. (Also Stanford University Technical Report No. CSL–TR–89–379.).

[10] 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.

[11] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.

[12] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[13] N. Madhav and W. R. Mann. A methodology for formal specification and implementation of Ada packages using Anna. In *Proceedings of the Computer Software and Applications Conference, 1990*, pages 491–496. IEEE Computer Society Press, 1990. (Also Stanford University Computer Systems Laboratory Technical Report No. 90-438).

[14] N. Madhav and S. Sankar. Application of formal specification to software maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 230–241. IEEE Computer Society Press, November 1990.

[15] Walter Mann. The Anna package specification analyzer user's guide. Technical Note CSL–TN–93–390, Computer Systems Lab, Stanford University, January 1993.

[16] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[17] 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.

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

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

[20] Sriram Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 123–129, Victoria, Canada, October 1991. ACM Press.

[21] Sriram Sankar, Anoop Goyal, and Prakash Sikchi. Software testing using algebraic specification based test oracles. Forthcoming Stanford University Technical Report, April 1993.

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

[23] L. G. Stucki and G. L. Foshee. New assertion concepts for self-metric software validation. In *Proceedings of the International Conference on Reliable Software*, pages 59–65, April 1975.

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

A Final Version of Garbage Collection Package

```
generic
   type Item is limited private;
   type Link is access Item;
package Garbage_Collection is

   --: function Is_Member(Atom:Link) return Boolean;
   --: function Cardinality return Integer;

   procedure Free(Item_Pointer:in out Link);
   --| where
   --|     in(not Is_Member(Item_Pointer)),
   --|     out(Item_Pointer = null),
   --|     out(Is_Member(in Item_Pointer));

   procedure Get(New_Item_Pointer:in out Link);
   --| where
   --|     out(New_Item_Pointer /= null),
   --|     out((in(Cardinality) /= 0) -> (Cardinality = in(Cardinality)-1)),
   --|     out(not Is_Member(New_Item_Pointer));

end Garbage_Collection;

package body Garbage_Collection is

   type Node_Type;
   type List_Type is access Node_Type;
   type Node_Type is record
      L:Link;
      Next:List_Type;
   end record;
   Free_Nodes_With_No_Items,Free_Nodes_With_Free_Items:List_Type;

   --: function Is_Member(Atom:Link) return Boolean is
   --:     Temp:List_Type;
   --: begin
   --:     Temp := Free_Nodes_With_Free_Items;
   --:     while (Temp /= null) loop
   --:         if Temp.L = Atom) then
   --:             return True;
   --:         end if;
```

```
--:          Temp := Temp.Next;
--:      end loop;
--:      return False;
--: end Is_Member;

--: function Cardinality return Integer is
--:      Temp:List_Type;
--:      Count:Integer := 0;
--: begin
--:      Temp := Free_Nodes_With_Free_Items;
--:      while (Temp /= null) loop
--:          Count := Count+1;
--:          Temp := Temp.Next;
--:      end loop;
--:      return Count;
--: end Cardinality;

procedure Free(Item_Pointer:in out Link) is
    Temp:List_Type;
begin
    if Item_Pointer /= null then
        if Free_Nodes_With_No_Items = null then
            Temp := new Node_Type;
        else
            Temp := Free_Nodes_With_No_Items;
            Free_Nodes_With_No_Items := Free_Nodes_With_No_Items.Next;
        end if;
        Temp.all := (L=>Item_Pointer, Next=>Free_Nodes_With_Free_Items);
        Free_Nodes_With_Free_Items := Temp;
        Item_Pointer := null;
    end if;
end Free;

procedure Get(New_Item_Pointer:in out Link) is
    Temp:List_Type;
begin
    if New_Item_Pointer = null then
        if Free_Nodes_With_Free_Items = null then
            New_Item_Pointer := new Item;
        else
            Temp := Free_Nodes_With_Free_Items;
            Free_Nodes_With_Free_Items := Free_Nodes_With_Free_Items.Next;
            New_Item_Pointer := Temp.L;
```

```
                Temp.all := (L => null, Next => Free_Nodes_With_No_Items);
                Free_Nodes_With_No_Items := Temp;
            end if;
        end if;
    end Get;

end Garbage_Collection;
```