# SOFTWARE TESTING USING ALGEBRAIC SPECIFICATION BASED TEST ORACLES

**Sriram Sankar** **Anoop Goyal** **Prakash Sikchi**

Technical Report: **CSL-TR-93-566**

(Program Analysis and Verification Group Report No. 64)

**April 1993**

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

# Software Testing using Algebraic Specification Based Test Oracles

Sriram Sankar Anoop Goyal Prakash Sikchi

**Technical Report: CSL-TR-93-566**
Program Analysis and Verification Group Report No. 64
April 1993

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

## Abstract

In TAV4, the first author presented a paper describing an algorithm to perform run-time consistency checking of abstract data types specified using algebraic specifications. This algorithm has subsequently been incorporated into a run-time consistency checking tool for the Anna specification language for Ada, and works on a subset of all possible algebraic specifications. The algorithm implementation can be considered a test oracle for algebraic specifications that performs its activities while the formally specified program is running.

This paper presents empirical results on the use of this test oracle on a real-life symbol table implementation. Various issues that arise due to the use of algebraic specifications and the test oracle are discussed. 50 different errors were introduced into the symbol table implementation. On testing using the oracle, 60% of the errors were detected by the oracle, 35% of the errors caused Ada exceptions to be raised, and the remaining 5% went undetected. These results are remarkable, especially since the test input was simply one sequence of symbol table operations performed by a typical client.

The cases that went undetected contained errors that required very specific boundary conditions to be met — an indication that white box test-data generation techniques may be required to detect them. Hence, a combination of white-box test-data generation along with a specification based test oracle may be an extremely versatile combination in detecting errors.

This paper does not address test-data generation, rather it illustrates the usefulness of algebraic specification based test oracles during run-time consistency checking. Run-time consistency checking should be considered a complementary approach to unit testing using generated test-data.

**Key Words and Phrases:** abstract data types, algebraic specifications, Anna, oracles, run-time consistency checking, software testing.

## Contents

# 1 Introduction

When a program is compiled, the compiler checks it for syntactic and static semantic correctness. When the compiled program is run, it is checked for run-time semantic errors (by checking code generated by the compiler). Finally, the programmer checks that the program correctly implements the task on hand by comparing program runs with the specification of the task. All of these activities involve checking the program for consistency with respect to some *specification*.

In languages such as Pascal and Ada, compile-time consistency checking involves activities such as determining the type compatibility of expressions. Run-time consistency checking involves activities such as determining that values assigned to variables are within a specified range, and determining that null pointers are not dereferenced. These consistency checking rules are incorporated into the definition of these languages, thus making programs written in these languages more reliable.

There is, however, a trade-off between reliability and efficiency. Many very useful specification constructs are not built into programming languages due to the overhead of implementing efficient consistency checking schemes with respect to these constructs. However, it might be useful to include these specification constructs into the language anyway and use them during the testing phase only. We have studied the problem of run-time consistency checking of programs with respect to a variety of specification constructs [6, 7, 8]. In [8], we present an algorithm for run-time consistency checking of abstract data types specified using algebraic specifications [3].

In this paper, we present empirical results on the use of algebraic specifications in specifying and testing abstract data types using the run-time consistency checking schemes we have developed. In Section 2, we discuss some issues involved in the writing of algebraic specifications for the purpose of run-time consistency checking. In Section 3, we discuss how algebraic specifications may be written in the specification language Anna [4] for Ada and the details of their semantics. We introduce a symbol table package in this section, which will be the example used throughout the rest of the paper. Following this (Section 4), we present an overview of what constitutes algebraic specification checking and the details of our implementation. In Section 5, we describe some transformations on the symbol table package which are required to be able to use the algebraic specification checking algorithms on this package. We tested the comprehensiveness of algebraic specification checking by introducing errors in the symbol table implementation and simulating the calls made to it by a typical application. About 60% of the errors were detected by the algebraic specification checking algorithm, another 35% caused Ada exceptions to be raised, and the remaining 5% of the errors were not detected. Details of these results are presented in Section 6. Finally, Section 7 concludes this paper and discusses possible areas of future work.

One of the interesting conclusions we make is that a combination of white-box test-data generation techniques in conjunction with specification based run-time consistency checking may be extremely versatile in detecting errors.

**Related work.** There has been other work done in the area of software testing based on algebraic specifications. They include DAISTS [2], EQUATE [9], and ASTOOT [1]. In all of these approaches, algebraic specifications are used as an aid to test-data generation. Our approach does not address test-data generation, rather it takes the form of a test oracle [5] that monitors the execution of a formally specified program. The calls made to abstract data type operations by the program will, therefore, constitute the test data. Our approach should be considered a complementary approach to test-data generation.

# 2 Writing and Using Algebraic Specifications for Testing Purposes

For the purpose of this paper, an algebraic specification is a set of equations whose terms are comprised of the abstract data type operations and variables that are universally quantified over the domain of the abstract data type. An example of an algebraic specification is presented in Section 3.

Algebraic specifications are a convenient way to describe the behavior of an abstract data type without over-constraining the implementation of the abstract data type. Also, for the purpose of software testing, algebraic specifications are very different from the code that forms the implementation — hence chances of repeating the same error in both the specification and the implementation is minimal.

There are however a few problems with using algebraic specifications during program testing. We list some of them below along with possible solutions.

- The general problem of algebraic specification checking is undecidable. When some set of sequences of abstract data type operations have been evaluated, it is necessary to perform proof operations to determine if these sequences result in equivalent abstract data type values. We must accept that algebraic specification checking can only be partial. But, as indicated by the results of this work and the other studies cited above, even this partial checking can be quite useful.

- To perform algebraic specification checking, the oracle must have access to an equality operation to compare two abstract data type values. Quite often, the abstract data type may not define such an operation in which case it must be provided by the programmer. In addition, a copy operation needs to be provided in case abstract data type values need to be saved for use in a later comparison with another value. There does not seem to be any way out of this problem.

- Testing tools based on algebraic specifications usually incorporate some kind of term rewriting capability. By rewriting terms into other terms, the oracle can conclude that the abstract data type values corresponding to these terms must be equal. If the intermediate terms in the rewriting process are undefined — for example, their evaluation raises an exception — the earlier conclusion about the terms being equal is wrong. That is, rewrite based systems can go wrong when the abstract data type operations are only partially defined. Our experience indicates that this problem does not occur in typical algebraic specifications, and in fact one can show that it does not occur in the example presented in this paper.

- Abstract data type operations may read global state and also have side-effects on this state. Hence two different executions of the same operation with the same parameters may result in different results. This causes a problem in writing algebraic specifications for these operations. The Anna specification language solves this problem by formalizing the notion of a package state and considering this to be an implicit parameter of all package operations. The problem of side effects will, however, continue to exist if multiple operations can access the state concurrently.

- To perform algebraic specification checking, it may often be desirable to revert the abstract data type back into an earlier state. For example, we may wish to re-initialize the abstract data type and perform a new sequence of operations. This may not be possible in some larger systems that are continuously running. For example, it may be impossible to revert a file system back to an earlier state for the purpose of testing. Although this problem does not arise in the example we present in this paper, it is a serious issue.

# 3 Algebraic Specifications in Anna

Anna (**ANN**otated **A**da) is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. The primary Anna construct is the *annotation*, which is a boolean-valued constraint on the underlying Ada program. Algebraic specifications can be written in Anna as *axiomatic annotations* which appear within an Ada package interface. The following is the Ada package interface of a symbol table with an algebraic specification written in Anna:

```
generic
    type Attribute is private;
package Symbol_Table_Package is
    procedure Initialize;
    procedure Insert(ID : String; AT : Attribute);
    procedure Replace(ID : String; New_AT : Attribute);
    procedure Search(ID : String; AT : out Attribute);
    Entry_Exists, Entry_Not_Found : exception;
    --| axiom
    --| for all S : Symbol_Table'State;
    --|         ID1, ID2 : String;
    --|         AT1, AT2 : Attribute =>
    --|     S[Insert(ID1, AT1); Insert(ID2, AT2)] = S[Insert(ID2, AT2); Insert(ID1, AT1)],
    --|     S[Insert(ID1, AT1); Replace(ID1, AT2)] = S[Insert(ID1, AT2)],
    --|     S[Insert(ID1, AT1); Replace(ID2, AT2)] = S[Replace(ID2, AT2); Insert(ID1, AT1)];
end Symbol_Table_Package;
```

This symbol table is a simplified form of that used in an Anna transformation tool. The original Anna transformation tool symbol table contained extra functionality to handle multiple levels of symbol tables and block structure. The above package introduces four operations Initialize, Search, Insert, and Replace and two exceptions Entry_Exists and Entry_Not_Found. Following this is an algebraic specification of the symbol table consisting of three equations.

We have omitted other specifications for simplicity — the important ones omitted being that Insert will raise the exception Entry_Exists if an attempt is made to insert the same symbol more than once, and that Replace will raise the exception Entry_Not_Found if its symbol parameter has not already been inserted earlier into the symbol table.

The equations that form the algebraic specification are quantified over all states, S, of the symbol table package, all strings ID1, ID2, and all attributes AT1 and AT2. The first equation says that inserting the symbol ID1 with attribute AT1 followed by inserting the symbol ID2 with attribute AT2 into any symbol table S is equivalent to performing these same insertions in the reverse order. Notice that we really want this equivalence to hold only when ID1 and ID2 are different from each other. However, when ID1 and ID2 are equal, both sides of the equation are undefined (they raise the exception Entry_Exists for they are both attempting to insert the same symbol twice). In Anna, the semantics of axiomatic annotations is that each equation must be true only for those values of the universally quantified variables for which the equation is fully defined. This *partial semantics* of Anna axiomatic annotations increases the number of situations that may be described using algebraic specifications.

The second equation says that inserting a symbol with one attribute and then replacing this attribute with another is equivalent to just inserting the symbol with the second attribute. The third equation says that Insert's and Replace's can be ordered either way to obtain the same result. Here again, this is true only when their symbol parameters are different from each other, and once again this is what the equation states as a result of the *partial semantics* property.

# 4 An Oracle for Anna Algebraic Specifications

We now present an overview of the oracle we implemented to perform run-time consistency checking with respect to algebraic specifications. The oracle maintains the set of all abstract data type terms generated by the program. If the oracle can deduce that any two terms in this set are equal based on the algebraic specification, then the oracle performs a check to ensure that the abstract data type values corresponding to these terms are also equal to each other. If they turn out not to be equal, the program is considered to have violated its algebraic specification. The details of the theorem proving algorithms used by the oracle are described in [6, 8].

In the symbol table package of Section 3, the abstract data type terms are sequences of symbol table operations starting with Initialize, and the abstract data type values are the resulting package states. As an example, consider the following sequence of operations on the symbol table package:

1. Initialize;
2. Insert("X", $AT_1$);
3. Insert("Y", $AT_2$);
4. Replace("Y", $AT_3$);
5. Initialize;
6. Insert("Y", $AT_2$);
7. Insert("X", $AT_1$);
8. Replace("Y", $AT_2$);

There are eight terms of the symbol table abstract data type generated by the program. Terms generated at *(1)* and *(5)* are identical (both contain only the operation Initialize), and hence the package states at these points are compared by the oracle to ensure that they are equal. The terms generated at *(3)* and *(7)* can be proved equal using the first equation of the algebraic specification of the symbol table. Similarly, the term generated at *(8)* can be proved equal to the term generated at *(7)* (and therefore also to the term generated at *(3)*) using the second and the third equations. Hence the package states at *(3)*, *(7)*, and *(8)* are compared by the oracle to ensure that they are equal.

Our oracle works on a subset of all possible abstract data types and their algebraic specifications. This subset is the set of all abstract data types and their algebraic specifications that satisfy the following conditions:

- The abstract data type must be implemented as an Ada package with a trivial state — *i.e.*, the state of the package may not be read or modified by the abstract data type operations. One Ada type within this package must be designated as the abstract data type. In addition, the Ada package may designate an Ada type to be the *auxiliary* type. The use of the auxiliary type is described below.
- The operations in the package must all be functions and are either observers (functions that query properties of abstract data type values) or constructors (functions that return abstract data type values).
- One constructor serves as the initialization routine for abstract data type values and this may have at most one parameter which must be of the auxiliary type. All other constructors must have exactly one parameter of the abstract data type and at most one more parameter which must be of the auxiliary type.
- The algebraic specification may only contain constructors.
- This is more a requirement than a subset condition — the user must provide copy and equality operations for the abstract data type and the auxiliary type.

Extending the oracle subset will complicate the implementation of the oracle. So instead, we have defined a set of transformations that can be used to convert a large number of abstract data types into the subset accepted by the oracle. This is described in the next section.

# 5 Transformations on the Symbol Table Package

Obviously, the symbol table package does not fit within the oracle subset. It implements the data abstraction in the package state — hence the package state is not trivial, and there is no Ada type designated to be the abstract data type. Also, the operations are procedures, not functions, and they each require two parameters — *i.e.*, there are two auxiliary types.

In this section we illustrate a transformation scheme using which a large number of abstract data types can be converted to fit into the oracle subset. The general scheme is shown in Figure 1 with respect to the symbol table abstract data type.

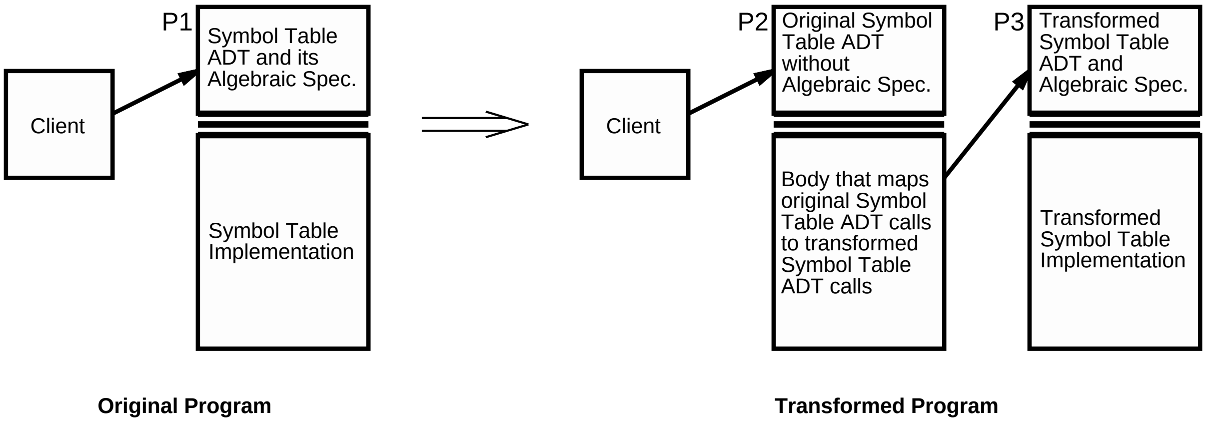


Figure 1: Transformation of Symbol Table

The left part of this figure shows the original program — the symbol table abstract data type and its algebraic specification as shown in Section 3, its implementation which is shown in Appendix A, and the client representing the rest of the program that makes calls to the symbol table. On the right of the figure is the transformed program. There are now two packages, `P2` and `P3`. `P2` has the same interface as that of the original package (`P1`) and hence the client can continue to call this package without any modifications. `P3` is the transformed version of the original abstract data type to fit into the oracle subset. The implementation of `P2` simply maps calls from the client to `P3`. `P3` contains the algebraic specification (which is slightly modified), and its implementation is nearly the same as the original implementation. This new implementation is, however, augmented with copy and equality operations on the abstract data type and the auxiliary type respectively, and is shown in Appendix B. The pragma that follows the copy and equality operations in this implementation indicates to the oracle that these are the operations to be used for copying and performing equality tests.

The details of the transformation are omitted, and the relevant aspects are illustrated by presenting the interface of `P3` below:

```
generic
    type Attribute is private;
package Symbol_Table_Customized is
    type Symbol_Table is private;   -- the ADT.
    type A_String is access String;
    type Auxiliary_Type is record
        ID : A_String;
        AT : Attribute;
    end record;
    function Initialize return Symbol_Table;
    function Insert(S : Symbol_Table; IDandAT : Auxiliary_Type) return Symbol_Table;
    function Replace(S : Symbol_Table; IDandAT : Auxiliary_Type) return Symbol_Table;
    function Search(S : Symbol_Table; ID : String) return Attribute;
```

```
    Entry_Exists, Entry_Not_Found : exception;
    --| axiom
    --| for all S : Symbol_Table;
    --|         ID1andAT1, ID2andAT2 : Auxiliary_Type =>
    --|     Insert(Insert(S, ID1andAT1), ID2andAT2) = Insert(Insert(S, ID2andAT2), ID1andAT1),
    --|     Replace(Insert(S, ID1andAT1), ID2andAT2) = Insert(S, ID2andAT2),
    --|     Replace(Insert(S, ID1andAT1), ID2andAT2) = Insert(Replace(S, ID2andAT2), ID1andAT1);
private
    type Node_Kind is (Normal, Last);
    type Node_Rec(N : Node_Kind);
    type Symbol_Table is access Node_Rec;
    subtype Node is Symbol_Table;
    Block_Size : constant := 4;
    type Node_Rec(N : Node_Kind) is record
        ID : String(1 .. Block_Size);
        LLink, RLink : Node;
        case N is
            when Normal =>
                MLink1 : Node;
            when Last =>
                MLink2 : Attribute;
        end case;
    end record;
end Symbol_Table_Customized;
```

The differences between P1 and P3 are listed below:

- P1 stores the symbol table in its state. P3 has a trivial state, but contains an Ada type Symbol_Table which is designated as the abstract data type. The type describing the package state in the original symbol table appears in the private part of P3 and describes the structure of Symbol_Table.

- P1 has two auxiliary types — String and Attribute. P3 contains a single auxiliary type which is a record with a String and a Attribute component.

- The P1 operations were all procedures. The P3 operations are functions which satisfy the oracle subset restrictions.

- The algebraic specification in P1 specified the effect a sequence of operations had on the package state. In P3, the algebraic specification is an expression specifying the effect of applying the functions on symbol table values.

There is a subtle difference between the algebraic specifications in P1 and P3. In P1, the second equation said that an Insert followed by a Replace on the *same symbol* is equivalent to just an Insert with the parameters of Replace. In P3 the "same symbol" constraint has been omitted for it cannot be stated in the oracle subset. However, as a consequence of the partial semantics of Anna axiomatic annotations, the second equation in P3 states exactly the same constraint. In the case when Insert and Replace do not operate on the same symbol, it is easy to see that at least one side of the equation will be undefined.

Note that the implementation of P3 can be somewhat inefficient. This inefficiency should, however, be acceptable during the testing phase, and in our experiments, there was no noticeable decrease in performance.

## 6 Experimental Results

To perform our experiment, one of the authors created a test-bed — a mainline that performed a sequence of calls on the symbol table package. This sequence contained 75 Initialize, Insert, and Replace operations with a variety of parameters. It was designed to simulate an actual use of the symbol table package.

Another author independently created 50 different mutants of the symbol table implementation. The mutants were created by making minor modifications to the implementations of the Insert and Replace functions — the kind that could typically occur during the development of these functions. We considered only the Insert and Replace functions to introduce errors since these were the only ones mentioned in the algebraic specification.

We tested each of these mutants using the algebraic specification based test oracle on the above-mentioned sequence of operations. The results were quite remarkable:

- In 26 cases, the algebraic specification based test oracle detected an error.
- In 18 cases, an Ada exception was raised, thus indicating an error.
- Errors were detected by both the oracle and by raising Ada exceptions in 3 cases — this was possible because some of the Ada exceptions were being handled and so the testing process could continue.
- The remaining 3 cases ran without any errors being detected.

That is, the oracle was capable of detecting 29 errors — approximately 60% of the errors. These errors would have gone undetected had we not used our oracle.

We performed an analysis of the 3 cases in which no errors were detected. We found that the test input required to detect errors in these mutants would require very specific boundary conditions to be satisfied. This knowledge was available only in the symbol table implementation.

*This gives rise to an interesting idea for future work:* If the test-data is generated using white-box techniques, and the tests are run in the presence of a specification based test oracle, we can ensure that the program runs correctly in all these cases. Chances are that all 50 errors introduced into the symbol table package would have been detected using this approach. We believe that this is a versatile approach for software testing and intend to study this in the future.

## 7 Conclusions and Future Work

Our experiment involving run-time consistency checking of abstract data types with respect to algebraic specifications demonstrates that our test oracle is indeed very useful in detecting errors in implementations of the abstract data type operations. Obviously, the abstract data type must be amenable to being specified using algebraic specifications, and it should be possible to transform it into the subset accepted by the oracle. It is also clear that our approach is a complementary approach to the related work cited in this paper that involve test-data generation.

Some of the more specific conclusions we have drawn based on the experiments are:

- We must extend the oracle subset to allow for an arbitrary number of auxiliary types. Although many examples such as the symbol table can be transformed to fit into the oracle subset, there could be examples that pose problems.
- A more rigorous error reporting scheme is required. Currently, the oracle just stops with an error message after highlighting the operation after which the error occurred. We also need to know how the oracle concluded the error — *e.g.*, the series of relevant rewritings performed.
- We need to study more real-life examples and carefully analyze the problem of partially defined operations.
- Finally, we need to study the use of our oracle in conjunction with white-box test-data generation techniques. The test-data would exercise the program quite comprehensively, and the oracle would ensure that the program meets its specification in all these cases.

## References


[1] R. K. Doong and P. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Victoria, Canada, October 1991. ACM Press.

[2] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[3] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[4] 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.

[5] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[6] 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.

[7] 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).

[8] 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.

[9] S. J. Zeil. Complexity of the EQUATE testing strategy. *Journal of Systems and Software*, 8:91–104, 1988.

# A The Symbol Table Package Implementation

```
package body SYMBOL_TABLE_PACKAGE is

    BLOCK_SIZE: constant := 4;
    type NODE_KIND is (NORMAL, LAST);
    type NODE_REC(N:NODE_KIND);
    type NODE                    is Access NODE_REC;
    type NODE_REC(N:NODE_KIND) is record
        ID          :STRING(1 .. BLOCK_SIZE);
        LLINK, RLINK:NODE;
        case N is
            when NORMAL =>
                MLINK1:NODE;
            when LAST =>
                MLINK2:ATTRIBUTE;
        end case;
    end record;

    STATE:NODE;
    LAST_NODE_VISITED_BY_SEARCH:NODE;

    procedure INITIALIZE is
    begin
        STATE := null;
    end INITIALIZE;

    procedure SEARCH(ID:STRING; AT:out ATTRIBUTE) is

        procedure LOCAL_SEARCH(ID:STRING; NOD:NODE) is
            FIRST:INTEGER := ID'FIRST;
            STR   :STRING(1 .. BLOCK_SIZE);
        begin
            if NOD = null then
                raise ENTRY_NOT_FOUND;
            elsif ID'LAST - FIRST + 1 >= BLOCK_SIZE then
                STR := ID(FIRST .. FIRST + BLOCK_SIZE - 1);
                if STR = NOD.ID then
                    if ID'LAST = FIRST + BLOCK_SIZE - 1 then
                        LOCAL_SEARCH(" ", NOD.MLINK1);
                    else
                        LOCAL_SEARCH(ID(FIRST + BLOCK_SIZE .. ID'LAST),
                                              NOD.MLINK1);
                    end if;
                elsif STR < NOD.ID then
```

```
            LOCAL_SEARCH(ID, NOD.LLINK);
        else
            LOCAL_SEARCH(ID, NOD.RLINK);
        end if;
    else
        STR(1 .. ID'LAST - FIRST + 1) := ID;
        for I in ID'LAST - FIRST + 2 .. BLOCK_SIZE loop
            STR(I) := ' ';
        end loop;
        if STR = NOD.ID then
            AT := NOD.MLINK2;
            LAST_NODE_VISITED_BY_SEARCH := NOD;
        elsif STR < NOD.ID then
            LOCAL_SEARCH(ID, NOD.LLINK);
        else
            LOCAL_SEARCH(ID, NOD.RLINK);
        end if;
    end if;
end LOCAL_SEARCH;

begin
    LAST_NODE_VISITED_BY_SEARCH := null;
    LOCAL_SEARCH(ID, STATE);
end SEARCH;

procedure INSERT(ID:STRING; AT:ATTRIBUTE) is
    L, M:NODE;

    procedure INSERT_REST_OF_ID(ID:STRING; LB:INTEGER := 1) is
        J:INTEGER;
        FIRST:INTEGER := ID'FIRST;
        STR:STRING(1 .. BLOCK_SIZE);
    begin
        for I in LB .. ((ID'LAST - FIRST + 1) / BLOCK_SIZE) - 1 loop
            STR := ID(FIRST + BLOCK_SIZE * I ..
                          FIRST + BLOCK_SIZE * (I + 1) - 1);
            L.MLINK1 := new NODE_REC'(NORMAL, STR, null, null, null);
            L := L.MLINK1;
        end loop;
        J := (ID'LAST - FIRST + 1) rem BLOCK_SIZE;
        STR(1 .. J) := ID(ID'LAST + 1 - J .. ID'LAST);
        for I in J + 1 .. BLOCK_SIZE loop
            STR(I) := ' ';
        end loop;
        L.MLINK1 := new NODE_REC'(LAST, STR, null, null, AT);
    end INSERT_REST_OF_ID;
```

```
procedure LOCAL_INSERT(ID:STRING; NOD:in out NODE) is
    FIRST:INTEGER := ID'FIRST;
    STR:STRING(1 .. BLOCK_SIZE);
begin
    if ID'LAST - FIRST + 1 >= BLOCK_SIZE then
        STR := ID(FIRST .. FIRST + BLOCK_SIZE - 1);
        if STR = NOD.ID then
            if ID'LAST = FIRST + BLOCK_SIZE - 1 then
                LOCAL_INSERT(" ", NOD.MLINK1);
            else
                LOCAL_INSERT(ID(FIRST + BLOCK_SIZE .. ID'LAST),
                                      NOD.MLINK1);
            end if;
        elsif STR < NOD.ID then
            if NOD.LLINK = null then
                NOD.LLINK := new NODE_REC'(NORMAL, STR, null, null, null);
                L := NOD.LLINK;
                INSERT_REST_OF_ID(ID);
            else
                LOCAL_INSERT(ID, NOD.LLINK);
            end if;
        else
            if NOD.RLINK = null then
                NOD.RLINK := new NODE_REC'(NORMAL, STR, null, null, null);
                L := NOD.RLINK;
                INSERT_REST_OF_ID(ID);
            else
                LOCAL_INSERT(ID, NOD.RLINK);
            end if;
        end if;
    else
        STR(1 .. ID'LAST - FIRST + 1) := ID;
        for I in ID'LAST - FIRST + 2 .. BLOCK_SIZE loop
            STR(I) := ' ';
        end loop;
        if STR = NOD.ID then
            raise ENTRY_EXISTS;
        elsif STR < NOD.ID then
            if NOD.LLINK = null then
                NOD.LLINK := new NODE_REC'(LAST, STR, null, null, AT);
            else
                LOCAL_INSERT(ID, NOD.LLINK);
            end if;
        else
            if NOD.RLINK = null then
```

```
                    NOD.RLINK := new NODE_REC'(LAST, STR, null, null, AT);
                else
                    LOCAL_INSERT(ID, NOD.RLINK);
                end if;
            end if;
        end if;
    end LOCAL_INSERT;

begin
    if STATE = null then
        L := new NODE_REC'(NORMAL, "        ", null, null, null);
        M := L;
        INSERT_REST_OF_ID(ID, 0);
        STATE := M.MLINK1;
    else
        LOCAL_INSERT(ID, STATE);
    end if;
end INSERT;

procedure REPLACE(ID:STRING; NEW_AT:ATTRIBUTE) is
    DUMMY:ATTRIBUTE;
begin
    SEARCH(ID, DUMMY);
    if LAST_NODE_VISITED_BY_SEARCH = null then
        raise ENTRY_NOT_FOUND;
    else
        LAST_NODE_VISITED_BY_SEARCH.MLINK2 := NEW_AT;
    end if;
end REPLACE;

end SYMBOL_TABLE_PACKAGE;
```

# B The Transformed Symbol Table Package Implementation

```
package body SYMBOL_TABLE_CUSTOMIZED is

    function EQUAL_SYMBOL_TABLE(X, Y : SYMBOL_TABLE) return BOOLEAN is ... ;
    function COPY_SYMBOL_TABLE(S : SYMBOL_TABLE) return SYMBOL_TABLE is ... ;
    function EQUAL_AUXILIARY_TYPE(X, Y : AUXILIARY_TYPE) return BOOLEAN is ... ;
    function COPY_AUXILIARY_TYPE(X : AUXILIARY_TYPE)
        return AUXILIARY_TYPE is ... ;

    pragma AXIOM_CHECKING_FUNCTIONS(EQUAL_SYMBOL_TABLE,
                                    COPY_SYMBOL_TABLE,
                                    EQUAL_AUXILIARY_TYPE,
                                    COPY_AUXILIARY_TYPE);

    function INITIALIZE return SYMBOL_TABLE is ... ;

    function SEARCH(S : SYMBOL_TABLE; ID : STRING) return ATTRIBUTE is ... ;

    function INSERT(S : SYMBOL_TABLE; IDandAT : AUXILIARY_TYPE)
         return SYMBOL_TABLE is ... ;

    function REPLACE(S : SYMBOL_TABLE; IDandAT : AUXILIARY_TYPE)
         return SYMBOL_TABLE is ... ;

end SYMBOL_TABLE_CUSTOMIZED;
```