Mutation Operators for Ada

A. Jefferson Offutt

Department of ISSE
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
email: ofut@isse.gmu.edu

Jeff Voas
Jeff Payne

Reliable Software Technologies Corp.
21515 Ridgetop Circle
Suite 250
Sterling, VA 20166
703-404-9293
{jepayn, jmvoas}@rstcorp.com

March 1996

Technical Report ISSE-TR-96-09

Abstract

Mutation analysis is a method for testing software. It provides a method for assessing the adequacy of test data. This report describes the mutation operators defined for the Ada programming language. The mutation operators are categorized using syntactic criteria, in a form suitable for an implementor of a mutation-based system, or a tester wishing to understand how mutation analysis can be used to test Ada programs.

Each mutation operator is carefully defined, and when appropriate, implementation notes and suggestions are provided. We include operators for all syntactic elements of Ada, including exception handling, generics, and tasking. A summary table listing all operators for Ada, and compared with C and Fortran operators is also provided. The design described here is the result of deliberations among the authors in which all aspects of the Ada language and software development in Ada were considered. These operators can also be viewed as the culmination of previous mutation operator definitions for other languages. This report is intended to serve as a manual for the Ada mutation operators.

1 MUTATION APPLIED TO ADA

Although mutation has been applied to several languages, the Ada programming language presents several new features, both syntactic and semantic. Most of the mutation operators for other languages are still meaningful for Ada, with appropriate changes in the names. In this section, we discuss these new features and make recommendations for handling them.

1.1 Strong Typing in Ada

Ada's strong typing requires that mutation operators be applied more strictly than in previous languages. Ada's static semantic restrictions, which enhance program reliability and readability, make brute-force applications of mutation operators likely to generate many invalid programs. For example, determining if the replacement of one arithmetic operator by another will yield a valid Ada expression is complicated by Ada's overloading rules.

In our mutation operator definitions, we point out situations when certain operators should not be applied because of the strong typing rules. We also divide the operators up at a fine-grain level to separate types. Although applying these restrictions require more care when building a mutation system, each program will have fewer mutants.

1.2 Syntactically Incorrect Mutants

A *stillborn* mutant is a mutant that is syntactically illegal. A *trivial* mutant is a mutant that is killed by almost any test case. An Ada mutation system will be expected to avoid generating stillborn mutants as often as is possible, and a well-designed set of operators will avoid most trivial mutants. While avoiding stillborn mutants will require more care on the part of the developer, and more analysis on the part of the mutation system, it will result in fewer Ada mutants, which will reduce the cost of testing.

In our definitions of mutation operators, we make several decisions to avoid trivial mutants. These decisions are mentioned explicitly.

1.3 Overloading in Ada

Ada allows operations to be "overloaded", that is, the same name can be used to represent more than one operation. This is primarily used when the same semantic operation is used for different types. For example, a print operation might be written for numeric types and character types, and the same name can be used to implement the operation for both types.

The only time that this feature impacts our operators is when one variable is being replaced by another. In this case, if the types are different, the replacement is normally not done. However, if the replacement results in an operation is valid because of overloading, then the replacement is done.

1.4 Pointers

In Fortran mutation systems such as Mothra [?], there were no pointers. In C, pointers are pervasive and untyped, and a pointer can be derived for any variable by "dereferencing". Additionally, C allows arithmetic operations to be applied to pointers, and even allows pointers to be mixed in expressions with integers and other types. Thus, C mutation systems such as PiSCES [?] must deal with many complexities because of the language.

In Ada, pointers are typed, and a limited number of operators can be applied to pointers. Thus, we are able to, in most cases, treat pointer types as just another kind of operand. A pointer reference is treated just as a variable reference, and we define Operand Replacement Operators to make appropriate substitutions.

1.5 Exception Handling in Ada

Ada includes a novel exception handling mechanism. Each program unit can have an associated *exception handler*, which defines statements to be executed when a particular exception is *raised.* When an exception is raised, if there is a handler for that particular exception, the associated statements are executed, otherwise the exception is propagated to the calling program unit. Some exceptions are built in to the language, and others can be defined by the programmer. The most common mistake is to use the wrong handler, or have the incorrect exception being raised. This situation is explicitly handled by a mutation operator (the SER operator).

1.6 Generic Packages in Ada

Ada allows packages (and procedures) to be *generic*, so that the package can be *instantiated* with parameters such as types etc. Our opinion is that mutations to generic package headers or instantiations would result in trivial mutants, thus we do explicitly mutate for generics.

It is possible that the testing process should be modified to reflect generics, so that a generic package is tested with various instantiations, but this is beyond the scope of mutation testing.

1.7 Tasking in Ada

Tasking is a major feature of the Ada language that is new to mutation systems. As a result, there is little experience with handling such constructs. We present a discussion of the issues involved with handling tasking, and include some initial recommendations for mutation operators in the mutation operator section.

The most obvious and relevant aspect of programs that use tasking is that the execution can be *non-deterministic.* Thus, executing a non-deterministic program on a test case will generate one among a potentially large set of correct outputs. We call this set of correct outputs of a test case on a program the *feasible output set*, and each correct execution will produce one element of this set. A mutant is *equivalent* if it generates the same feasible output set as the original program for all inputs. Unfortunately, this means that equivalent non-deterministic programs can produce different output. A mutant is *weakly equivalent* if its feasible output set is a subset of the feasible output set of the original program. That is, if a mutant always produces some correct output, but is incapable of producing all correct outputs, it is considered to be weakly equivalent.

The non-deterministic nature of tasking programs requires a modified definition of correctness. The output of a mutant program on a test case is *correct* if the output is in the feasible output set of the program. With deterministic programs, mutation systems can easily determine the correctness of a given output; they merely compare the output with the output of the original program on the same test case. With non-deterministic programs, however, that is not possible. We suggest the following approximation.

Run the original program N times on the same test case t to create N outputs O_i, $1 \leq i \leq N$. The set $\Omega = \{O_1, O_2, ..., O_N\}$ is an approximation of the feasible output set. Run the mutant program m on the test case to create $O(m,t)$. If $O(m,t) \in \Omega$, then the mutant is left alive, else it is killed. Note that if P is a deterministic program, mutant output checking is a special case of the non-deterministic case, where Ω has a single element.

Of course, how well Ω approximates the true feasible output set of the program depends largely on the value of N. To get the true feasible output set, we may need to run the program an infinite number of times. A value for N can be estimated at several points during testing:

1. Determined by the mutation system as a constant for all program,
2. Set by the tester for each application,
3. Estimated by the system for each test case. Run the original repeatedly until a small number of executions are made without creating a new output.

The third option is more precise, and should result in a more accurate estimation of the feasible output set, but will be more expensive than the other two options. Moreover, the third option will not always terminate if each execution produces a different output. For example, if the feasible output set is infinite and each element is equally likely to be generated, we can expect each execution to produce a new output. In this case, the approximation approach will not work anyway, because each mutant can be expected to generate a unique output.

The options above are all automatable, and based on enumerating all or part of the feasible output set. Another method of determining whether a mutant's output is correct is based on a semi-automated method. If the tester can describe the feasible output set in some way, then the output can be checked to see if it matches the description. Unfortunately, this description depends on the application program. It is possible that a general-purpose language could be devised that would allow a tester to enter a description of the feasible output set.

2 ADA MUTATION OPERATORS

For a program P, mutation testing produces a set of alternate programs. Each alternate program, P_i, known as a *mutant* of P, is formed by modifying a single statement of P according to some predefined modification rule. These modification rules are called *mutation operators*. The syntactic change itself is called the *mutation*, and the resulting program is the *mutant program*, or simply mutant. The original program plus the mutant programs are collectively known as the *program neighborhood*, N, of P [**?**].

This report defines a set of mutation operators for the Ada programming language. Our operators are partially based on the previous operators defined for Ada [**?**], the C operators [**?**], and the Fortran-77 [**?**] operators used by Mothra [**?**]. These operators are complete for the Ada language as defined in the Ada Reference Manual [**?**]. This is as distinct from the previous Ada operators, which did not cover the entire language. Additionally, the previous operators were designed with very little experience with the Ada language, or writing Ada programs. As a result, our operators are significantly more extensive than the earlier set. Because of the extensive experience we have had with the Fortran mutation operators as implemented in the Mothra testing system, they have influenced our Ada operators most heavily.

We organize our operators differently than authors of previous sets of operators. In particular, we separate our operators primarily on the basis of what type of lexical elements are modified; this gives us four types of operators. Mutation operators within these groups have reasonably uniform semantics and rules for applications. Also, the number of mutants produced are on the same order of magnitude for all operators within our types. We also include one type of operators (Coverage), specifically to include branch coverage testing strategies.

The five types of mutation operators for Ada are:

- Operand Replacement Operators (30 operators)
- Statement Operators (14 operators)
- Expression Operators (14 operators)
- Coverage Operators (4 operators)
- Tasking Operators (3 operators)

We have a total of 65 operators. In the following section, we define each operator in turn. Following that, we present all of our Ada operators in one comprehensive table, shown with the correlating Fortran and C operators, if any. Lastly, we show an Ada package with example mutants.

3 ADA MUTATION OPERATOR DEFINITIONS

This section comprises the major part of this report, both in technical terms, and in bulk. Each type of mutation operator is discussed in a separate subsection, and each individual operator is defined. Each subsection starts with a general discussion about the operator type, then a table is given listing all the operators of that type. Next, each operator is defined in turn.

3.1 Operand Replacement Operators

Each operand replacement operator starts with the letter `O`. There are 29 operand replacement operators. These operators cause each operand to be replaced by each other syntactically legal operand. There are five kinds of operands in Ada:

1. Variables
2. Constants
3. Array References
4. Record References
5. Pointer References

Although in Ada there is no real difference between record and pointer references, we define separate operators so as to have uniform definitions. Replacing these five kinds of operands result in 25 operators; there are four additional operators for three structured types, and one additional operator for variable initialization.

Operand Replacement Operators	
`OVV`	Variable replaced by a variable.
`OVC`	Variable replaced by a constant.
`OVA`	Variable replaced by an array reference.
`OVR`	Variable replaced by a record reference.
`OVP`	Variable replaced by a pointer reference.
`OVI`	Variable initialization elimination.
`OCV`	Constant replaced by a variable.
`OCC`	Constant replaced by a constant.
`OCA`	Constant replaced by an array reference.
`OCR`	Constant replaced by a record reference.
`OCP`	Constant replaced by a pointer reference.
`OAV`	Array reference replaced by a variable.
`OAC`	Array reference replaced by a constant.
`OAA`	Array reference replaced by an array reference.
`OAR`	Array reference replaced by a record reference.
`OAP`	Array reference replaced by a pointer reference.
`OAN`	Array name replaced by an array name.
`ORV`	Record reference replaced by a variable.
`ORC`	Record reference replaced by a constant.
`ORA`	Record reference replaced by an array reference.
`ORR`	Record reference replaced by a record reference.
`ORP`	Record reference replaced by a pointer reference.
`ORF`	Record field replaced by a record field.
`ORN`	Record name replaced by a record name.
`OPV`	Pointer reference replaced by a variable.
`OPC`	Pointer reference replaced by a constant.
`OPA`	Pointer reference replaced by an array reference.
`OPR`	Pointer reference replaced by a record reference.
`OPP`	Pointer reference replaced by a pointer reference.
`OPN`	Pointer name replaced by a pointer name.

Notes:
The strong typing rules of Ada will drastically reduce the number of mutants of this type that are generated.

Do mutate initializations (only `OCC`).

Do mutate references of enumerated types.

Do not mutate types.

Do not mutate declarations.

Do not mutate CASE constants.

Do not mutate loop parameters on FOR statements (it is a declaration).

Variables that are of a type that is declared externally and private are considered to be scalar.

The following named objects are considered as CONSTANT and are mutated using OVC, OC?, ORC, OAC, and OPC:

- Objects declared with the keyword CONSTANT
- Loop parameters
- Parameters of class IN

1. The 25 simple replacement operators are all uniform and merely replace one type with another.
2. `OVI`: Variable initialization elimination.
   Eliminate the initialization part of each variable initialization.
3. `OAN`: Array name replaced by an array name.
   Replace **just** the array name in an array reference by other array names when the base types are the same, and the index types are the same.
4. `ORF`: Record field replaced by a record field.
   Replace a record field reference by another field name of the same record when the second field is of the same type.
5. `ORN`: Record name replaced by a record name.
   Replace **just** the record name in a record reference by other record names when the field names and types are the same.
6. `OPN`: Pointer name replaced by a pointer name.
   Replace **just** the pointer name in a pointer reference by other pointer names when the field names and types are the same.

3.2 Statement Modification Operators

Each statement modification operator starts with the letter `S`. There are 13 statement modification operators. These operators modify entire statements and modify the control structures of Ada. The relevant control structures are:

1. BLOCK
2. CASE
3. EXIT
4. FOR
5. GOTO
6. IF
7. LOOP
8. RAISE
9. RETURN
10. WHILE

We summarize the operators in a table, then discuss each operator in detail.

Statement Modification Operators	
SEE	Exception on execution.
SRN	Replace with NULL.
SRR	Return statement replacement.
SGL	GOTO label replacement.
SRE	Replace with EXIT.
SWR	Replace WHILE with repeat-until.
SRW	Replace repeat-until with WHILE.
SZI	Zero iteration loop.
SOI	One iteration loop.
SNI	N iteration loop.
SRI	Reverse iteration loop.
SES	END shift.
SCA	CASE alternative replacement.
SER	RAISE exception handler replacement.

1. `SEE`: Exception on execution.
   Replace the first statement in each basic block with:
   ```
   RAISE mut_trap;
   ```
   `mut_trap` is a mutation-defined exception. We need to get the mutant number to the handler, which can be in the local procedure or the main program. We recommend having the mutant call a subroutine:
   ```
   Except_on_Exec (42);
   ```
   which then raises the exception.

   Notes:

   Do not replace if elimination of the statement would result in a compile-time error, for example, if the statement is the only RETURN in a function.

   `SEE` should be applied to statements and block statements. For example, the following structure will result in four mutants:

   ```
   WHILE (e1) LOOP
         IF (e2) THEN
            s1;
         ELSE
            s2;
         END IF;
   END LOOP;


   Mutant 1:   Except_on_Exec (n);


   Mutant 2:   WHILE (e1) LOOP
                     Except_on_Exec (n);
               END LOOP;
   ```

```
Mutant 3:   WHILE (e1) LOOP
                  IF (e2) THEN
                     Except_on_Exec (n);
                  ELSE
                     s2;
                  END IF;
            END LOOP;
```

```
Mutant 4:   WHILE (e1) LOOP
                  IF (e2) THEN
                     s1;
                  ELSE
                     Except_on_Exec (n);
                  END IF;
            END LOOP;
```

2. `SRN`: Replace with NULL.

   Replace each statement with NULL.

   The replacement should be done according to the rules of the `SEE` mutation operator, except the replacement should be done on each statement, not each basic block.

   Do not replace if elimination of the statement would result in a compile-time error, for example, if the statement is the only RETURN in a function.

3. `SRR`: Return statement replacement.

   Replace each statement in a FUNCTION or PROCEDURE with RETURN.

   For parameterized RETURN statements (in functions), replace each statement with every RETURN that appears in the function. Do not replace RETURN statements.

   Do not replace if elimination of the statement would result in a compile-time error.

4. `SGL`: GOTO label replacement.

   Replace each GOTO label with all other visible, legal labels.

   The innermost sequence of statements that encloses the target statement must also enclose the GOTO statement (note that the GOTO statement can be a statement of an inner sequence). Furthermore, if a GOTO statement is enclosed by an ACCEPT statement or the body of a program unit, then the target statement must not be outside this enclosing construct; conversely, it follows form the previous rule that if the target statement is enclosed by such a construct, then the GOTO statement cannot be outside.

5. `SRE`: Replace with EXIT.

   This operator replaces statements within loops with EXIT statements. There are three variations.

   (a) Replace each statement in a loop with EXIT;

   (b) Replace each statement in a loop with an EXIT *name*; for each named enclosing loop.

   (c) Replace each statement in a loop with each EXIT WHEN ...; that appears in the loop.

   Notes:

   If there is only one statement in the loop, this change would be equivalent to SRN, so do not generate.

   The C operator SBR only did the second of the three variations.

6. `SWR`: Replace WHILE with repeat-until.

   Although there is no explicit repeat-until statement in Ada, the construct is commonly built using a LOOP and an EXIT. Using the incorrect kind of loop is a common programming mistake. The format of the change is:

```
   ORIGINAL                 MUTANT

WHILE (e) LOOP           LOOP
   .                       .
   :                       :
END LOOP;                EXIT WHEN NOT e;
                         END LOOP;
```

7. `SRW`: Replace repeat-until with WHILE.

   This is the opposite of `SWR`. The format of the change will be:

```
    ORIGINAL                 MUTANT

LOOP                         WHILE (NOT e) LOOP
  .                             .
  :                             :
EXIT WHEN e;                 END LOOP;
END LOOP;
```

   Rather than only applying this operation to loops where the EXIT WHEN statement is the last statement in the loop body, it is applied to **all** EXITs in the loop.

8. Definite loop mutations.

   We have four goals for mutating definite loops (FOR).

(a) Bypass the loop entirely (zero iterations).

(b) Cause the loop to iterate once (one iteration).

(c) Cause the loop to be iterated more than once (N iterations).

(d) Cause the loop to be executed in reverse (reverse iteration).

The first three goals are satisfied by introducing a new loop counter for each loop. For a loop i, associate the counter *loop_i_count*. *loop_i_count* is initialized to zero before the loop begins. It is incremented by one each iteration through the loop.

(a) `SZI`: Zero iterations

After the loop, if $loop\_i\_count = 0$, then RAISE Mut_Trap;

(b) `SOI`: One iteration

After the loop, if $loop\_i\_count = 1$, then RAISE Mut_Trap;

(c) `SNI`: N iterations

After the loop, if $loop\_i\_count > 1$, then RAISE Mut_Trap;

(d) `SRI`: reverse iteration

Add the keyword REVERSE to the loop if it is not there, remove it if it is there.

9. `SES`: END shift.

Move each END statement up and down one statement. This applies to END statements occurring in BLOCK and LOOP statements, but not CASE statements and subprograms.

10. `SCA`: CASE alternative replacement.

First, each case statement alternative with multiple choices is separated into alternatives where each alternative contains only one choice. A range (e.g., 5..20) is considered to be only one choice. Next, substitute each statement sequence with each other sequence in the CASE statement.

Do not mutate choices.

Example:

```
CASE Var1 is
   WHEN A | B  => statements_1;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_3;
END CASE;
```

This case statement creates 8 mutants:

```
CASE Var1 is
   WHEN A      => statements_2;
   WHEN B      => statements_1;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_3;
   WHEN B      => statements_1;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
   WHEN B      => statements_2;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
   WHEN B      => statements_3;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
   WHEN B      => statements_1;
   WHEN C      => statements_1;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
   WHEN B      => statements_1;
   WHEN C      => statements_3;
   WHEN OTHERS => statements_3;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
   WHEN B      => statements_1;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_1;
END CASE;

CASE Var1 is
   WHEN A      => statements_1;
```

```
   WHEN B      => statements_1;
   WHEN C      => statements_2;
   WHEN OTHERS => statements_2;
END CASE;
```

11. SER: RAISE exception handler replacement.

    For each explicit RAISE statement, replace the name of the exception by other exceptions. Replace programmer-defined exceptions only by other programmer-defined exceptions, and built-in exceptions by other built-in exceptions.

3.3 Expression Modification Operators

Each expression modification operator starts with the letter E. There are 14 expression modification operators. These operators modify expression operators and entire expressions. We summarize the operators in a table, then discuss each operator in detail.

Expression Modification Operators	
EAI	Absolute value insertion.
ENI	Neg-Absolute value insertion.
EEZ	Exception on zero.
EOR	Arithmetic operator replacement.
ERR	Relational operator replacement.
EMR	Membership test replacement.
ELR	Logical operator replacement.
EUI	Unary operator insertion.
EUR	Unary operator replacement.
ESR	Subprogram operator replacement.
EDT	Domain twiddle.
EAR	Attribute replacement.
EEO	Exception on overflow.
EEU	Exception on underflow.

1. EAI: Absolute value insertion.
   Insert the unary operator ABS in front of every arithmetic expression and subexpression. Do not mutate if the expression can be statically determined to be greater than or equal to zero. For example, we can determine this for the following cases:

   - Constants
   - The type is a nonnegative subtype of Integer (for example, Natural or Positive)
   - Loop variable where the lower bound is greater than or equal to zero.

   Do not mutate if the change will make a discrete range in a FOR statement have a NULL range (this would be equivalent to an SZI mutant).

   Do not mutate CONSTANTS (this would be equivalent to a EUI mutant).

2. `ENI`: Neg-absolute value insertion.
   Insert $-$ABS in front of every arithmetic expression and subexpression. Do not mutate if the expression can be statically determined to be less than or equal to zero. For example, we can determine this for the following cases:

   - Constants
   - The type is a negative subtype of Integer.

   Do not mutate if the change will make a discrete range in a FOR statement have a NULL range (this would be equivalent to an SZI mutant).

   Do not mutate CONSTANTS (this would be equivalent to a EUI mutant).

3. `EEZ`: Exception on zero.
   Insert the subprogram Except_on_Zero in front of every arithmetic expression and subexpression. Except_on_Zero(E); raises EEZ_Exception if E is 0, else it returns E. Do not mutate if the expression can be statically determined to be not equal to 0. For example, we can determine this for the following cases:

   - Constants.
   - The type is a subtype of Integer that does not include 0 (for example, Positive).
   - Loop variable where the range does not include 0.

4. `EOR`: Arithmetic operator replacement.
   Replace each binary arithmetic operator ($+, -, *, /$, MOD, REM, $**$) with each other binary arithmetic operator that is syntactically legal.

   Strong typing notes:

   - MOD and REM are only defined for Integer types.
   - $**$ requires the right operand to be Integer.
   - $*$ allows Fixed Point and Integer to be mixed.
   - / allows Fixed Point on the left and Integer on the right.

5. `ERR`: Relational operator replacement.
   Replace each relational operator with each other relational operator that is syntactically legal.

   $<$, $>$, $>=$, $<=$ are only defined for scalar and discrete array types.

6. `EMR`: Membership test replacement.
   Replace each IN with NOT IN and each NOT IN with IN.

   Note: This operator is subsumed by the CDE operator and should not be used if CDE is.

7. `ELR`: Logical operator replacement.
   Replace each logical operator (AND, OR, XOR, AND THEN, OR ELSE) with each other logical operator.

   AND, OR, and XOR are defined for Boolean expressions and one-dimensional arrays of type Boolean.

8. `EUI`: Unary operator insertion.
   Insert the unary operator $-$ in front of each arithmetic expression and subexpression.

   Note: the unary operator $+$ is the identity operation.

9. `EUR`: Unary operator replacement.
   Replace each unary operator $(+, -, ABS)$ with each other unary operator.

   Expressions should be fully parenthesized, since ABS has higher precedence than $+$ and $-$.

10. ESR: Subprogram operator replacement.
Replace each function and subroutine name with each other function or subroutine name that has the same syntactic signature and comes from the same package.

Also replace with = and / = if the signature is appropriate (= and / = are implicitly defined for all types).

Do not consider the parameter class in the signature comparison.

Example:
Package Matrix Specification:

```
Matrix_Type ...
"+" (M1, M2: Matrix_Type) RETURN Matrix_Type;
"*" (M1, M2: Matrix_Type) RETURN Matrix_Type;
"<" (M1, M2: Matrix_Type) RETURN Boolean;
```

Matrix Use:

```
A, B, C : Matrix_Type;
   .
   :
C := A + B;       ==> mutation ==>  C := A * B;
   .
   :
IF (A < B) ...    ==> mutation ==>  IF (A = B) ...
                  ==> mutation ==>  IF (A /= B) ...
```

11. EDT: Domain twiddle.

Each innermost expression (operand: constant, variable, array reference, record reference, pointer reference) is *twiddled*, that is, modified by a small amount. For each operand, the modification produces two mutants, one where the modification is in a positive direction, the other in a negative direction. This amount depends on the type:

TYPE		MODIFICATION
Integer	–	+1 and -1
Float	–	*1.05 and *.95
Fixed Point	–	+Delta and -Delta
Character Types	–	T'SUCC and T'PRED
Enumerated Types	–	T'SUCC and T'PRED

The twiddle must not create a value out of the range for that type. For example, if X is of type Natural and has the value 0, the mutant -1 is not generated.

Notes:

Do not twiddle array subscripts – most changes would cause an out-of-bounds failure.

Do not mutate if the change would result in mutant that is equivalent to another twiddle on the same expression (for example, (X+Y) ==> ((X-1)+Y) and ((X+1)+Y), but not (X+(Y-1)) and (X+(Y+1))).

Do not mutate loop parameters if the change would result in a NULL range (this would be equivalent to an OCC mutant).

12. `EAR`: Attribute replacement.

    Each attribute is replaced by each other syntactically legal attribute. Attributes are defined in Appendix A of the Ada Reference Manual.

13. `EEO`: Exception on overflow.

    Insert the subprogram Except_on_OverFlow in front of every arithmetic expression. Except_on_OverFlow(E) raises EEO_Exception if the expression results in an overflow, else it returns the value of the expression. Do not mutate if the expression can be statically determined to not overflow. For example, we can determine this for the following cases:

    - Constants.
    - Loop variable.

    Notes:

    It might be possible to allow the Ada runtime system to detect overflow problems, and define a handler for the overflow. The ADA reference manual, section 4.5.7, paragraph 7, says:

    > "If the result overflows, NUMERIC_ERROR should be raised, but will not necessarily be raised. That is, it is not strictly required."

    The ADA reference manual, section 13.7.3, says:

    > "If an overflow occurs, and there is no NUMERIC_ERROR, T'MACHINE-OVERFLOWS is FALSE, else TRUE."

    This does not make sense to me, for two reasons:

    (a) The value for T'MACHINE_OVERFLOWS seems to be backwards.

    (b) NUMERIC_ERROR is not required because detecting overflow is hard in some situations. But setting this attribute requires overflow to be detected.

14. `EEU`: Exception on underflow.

    Insert the subprogram Except_on_UnderFlow in front of every arithmetic expression. Except_on_UnderFlow(E) raises EEU_Exception if the expression results in an underflow, else it returns the value of the expression. Do not mutate if the expression can be statically determined to not underflow. For example, we can determine this for the following cases:

- Constants.
- Loop variable.

3.4 Coverage Operators

The previous operators do not cover the branch coverage criteria [?] as do the Mothra operators [?]. For the Ada operators, we have chosen to define separate operators expressly for this purpose. This is so that the tester can explicitly choose to cover one or more of the branch coverage criteria, without having to use other operators.

The coverage criteria we consider are based on the following definitions:

Definition: A *Condition* in a program is a pair of algebraic expressions related by one of the relational operators $\{>, <, =, \geq, \leq, \neq\}$.

Conditions evaluate to one of the binary values TRUE or FALSE and can be modified by the negation operator NOT.

Definition: A *Decision* is a list of one or more conditions connected by the two logical operators AND and OR and used in a statement that affects the flow of control of the program. Decisions represent branches in the control flow of the program.

Statement Coverage (SC) requires that every statement in the program be executed at least once. *Decision Coverage (DC)* requires that every decision evaluate to both TRUE and FALSE at least once. DC is also known as *branch testing* and *all-edges* [?]. *Condition Coverage (CC)* requires that each condition in each decision evaluate to both TRUE and FALSE at least once. *Decision / Condition Coverage (DCC)* requires that each condition in each decision evaluate to both TRUE and FALSE at least once, and that every decision evaluate to both TRUE and FALSE at least once. DCC combines DC and CC. *Modified Condition / Decision Coverage (MC/DC)* requires that every decision and every condition within the decision has taken every outcome at least once, and every condition has been shown to independently affect its decision. *Multiple-Condition Coverage (MCC)* requires that all possible combinations of condition outcomes in each decision be covered, that is, the entire truth table for the decision has been satisfied. MCC is also known as *extended branch coverage* [?].

We have designed four Ada operators specifically to cover these coverage criteria. Each coverage operator starts with the letter `C`. We summarize the four coverage operators in a table, then discuss each operator in detail. The `SEE` operator satisfies statement coverage, so it is not included in the coverage operators set.

Coverage Operators	
`CDE`	Decision coverage.
`CCO`	Condition coverage.
`CDC`	Decision/condition coverage.
`CMC`	Multiple condition coverage.

1. `CDE`: Decision coverage
   Each decision must evaluate to both TRUE and FALSE. Replace each decision by TRUE and FALSE.

2. `CCO`: Condition coverage
   Each condition must evaluate to both TRUE and FALSE. Replace each condition by TRUE and FALSE.

   Note: there will be some redundancy. This could be reduced by having the mutation system suppress some mutants.

3. `CDC`: Decision/condition coverage
   Decision/condition coverage combines decision and condition. The `CDC` operator simply turns on `CCO` and `CDE`. We define it separately as a convenience.

4. `CMC`: Multiple condition coverage
   All combinations of conditions must be exercised separately, which yields, for a decision with n conditions, 2^n combinations. Another way of stating the MCC requirement is that the entire truth table for the decision must be covered.

   We define the `CMD` operator in two separate ways. The second way is more efficient.

   1. Mothra-like definition

      (a) Replace each condition with TRUE and FALSE (`CCO`), and
      (b) Replace each subset of conditions with TRUE and FALSE, and
      (c) Replace each subset of logical connectors with $\neq$ (XOR) and $=$ (NOT XOR).

   This will satisfy MCC, but will result in a fair amount of redundancy. There will be far more than 2^n mutants.

   2. Pisces-like implementation At each decision, create 2^N mutants. Assume the decision is D, with N conditions $C_i, 1 \leq i \leq N$.

   Create the entire truth table for D, TT, where TT_j is the truth assignment needed for mutant j, $1 \leq j \leq 2^N$. TA is the current truth assignment for D.

   The implementation of the mutant is as follows:

```
IF (TA = TT_i) then
    IF applying weak mutation
        Kill mutant j
    ELSE
        RETURN NOT (TA)
    END IF
END IF
```

   This could be done in an evaluative way, as Pisces does, or using Schema.

   Note: If `CMC` is used, `CDE`, `CCO`, and `CDC` are redundant and should not be used.

3.5 Tasking Operators

We have designed three Ada operators specifically to cover tasking. Each coverage operator starts with the letter `T`. We summarize the tasking operators in a table, then discuss each operator in detail.

Tasking Operators	
`TEM`	ENTRY statement modification.
`TAR`	ACCEPT statement replacement.
`TSA`	SELECT alternative replacement.

1. `TEM`: ENTRY statement modification
   Each ENTRY call is modified just as procedure calls are modified by the ESR operator. Replace each ENTRY call name with each other ENTRY name that has the same syntactic signature and comes from the same task.

   Also replace conditional and timed entry calls by simple entries.

2. `TAR`: ACCEPT statement replacement
   Replace entry names by other visible entries of the same time.

3. `TSA`: SELECT alternative replacement
   Each SELECT alternative is modified just as the CASE statement is modified by the SCA operator. First, each SELECT statement alternative with multiple choices is separated into alternatives where each alternative contains only one choice. Next, substitute each statement sequence with each other sequence in the SELECT statement.

4 COMPARISON OF ADA, C AND FORTRAN-77 OPERATORS

This section contains a table that attempts to relate our Ada mutation operators with the previous operators for Ada [?], and the C [?] and Fortran-77 [?] operators. The character ~ means that there is no corresponding operator.

Operand Replacement Operators			
Ada	Description	C	Fortran-77
OVV	Variable replaced by a variable.	Vsrr	svr
OVC	Variable replaced by a constant.	Vsrr	csr
OVA	Variable replaced by an array reference.	Vsrr	asr
OVR	Variable replaced by a record reference.	Vsrr	~
OVP	Variable replaced by a pointer reference.	Vsrr	~
OVI	Variable initialization elimination.	~	~
OCV	Constant replaced by a variable.	Ccsr	scr
OCC	Constant replaced by a constant.	Cccr	src
OCA	Constant replaced by an array reference.	~	acr
OCR	Constant replaced by a record reference.	~	~
OCP	Constant replaced by a pointer reference.	~	~
OAV	Array reference replaced by a variable.	Varr	sar
OAC	Array reference replaced by a constant.	Varr	car
OAA	Array reference replaced by an array reference.	Varr	aar
OAR	Array reference replaced by a record reference.	Varr	~
OAP	Array reference replaced by a pointer reference.	Varr	~
OAN	Array name replaced by an array name.	Varr	cnr
ORV	Record reference replaced by a variable.	Vtrr	~
ORC	Record reference replaced by a constant.	Vtrr	~
ORA	Record reference replaced by an array reference.	Vtrr	~
ORR	Record reference replaced by a record reference.	Vtrr	~
ORP	Record reference replaced by a pointer reference.	Vtrr	~
ORF	Record field replaced by a record field.	VSCR	~
ORN	Record name replaced by a record name.	~	~
OPV	Pointer reference replaced by a variable.	Vprr	~
OPC	Pointer reference replaced by a constant.	Vprr	~
OPA	Pointer reference replaced by an array reference.	Vprr	~
OPR	Pointer reference replaced by a record reference.	Vprr	~
OPP	Pointer reference replaced by a pointer reference.	Vprr	~
OPN	Pointer name replaced by a pointer name.	~	~

Statement Modification Operators			
Ada	Description	C	Fortran-77
SEE	Exception on execution.	STRP	SAN
SRN	Replace with NULL.	SSDL	SDL
SRR	Return statement replacement.	SRSR	RSR
SGL	GOTO label replacement.	SGLR	GLR
SRE	Replace with EXIT.	SBR	~
SWR	Replace WHILE with repeat-until.	SWDD	~
SRW	Replace repeat-until with WHILE.	SDWD	~
SZI	Zero iteration loop.	~	der
SOI	One iteration loop.	~	~
SNI	N iteration loop.	SMTT	~
SRI	Reverse iteration loop.	~	~
SES	END shift.	SMVB	~
SCA	CASE alternative replacement.	SSWM	~
SER	RAISE exception handler replacement.	~	~
Expression Modification Operators			
Ada	Description	C	Fortran-77
EAI	Absolute value insertion.	VDTR	ABS
ENI	Neg-absolute value insertion.	VDTR	ABS
EEZ	Exception on zero.	VDTR	ABS
EOR	Arithmetic operator replacement.	ORAN	AOR
ERR	Relational operator replacement.	ORRN	ROR
EMR	Membership test replacement.	~	~
ELR	Logical operator replacement.	OBBN	LCR
EUI	Unary operator insertion.	Uuor	UOI
EUR	Unary operator replacement.	Uuor	~
ESR	Subprogram operator replacement.	~	~
EDT	Domain twiddle.	VTWD	crp
EAR	Attribute replacement.	~	~
EEO	Exception on overflow.	~	~
EEU	Exception on underflow.	~	~
Coverage Operators			
Ada	Description	C	Fortran-77
CDE	Decision coverage.	Oior	lcr
CCO	Condition coverage.	~	ror
CDC	Decision/condition coverage.	~	lcr, ror
CMC	Multiple condition coverage.	~	ror, lcr
Tasking Operators			
Ada	Description	C	Fortran-77
TEM	ENTRY statement modification.	~	~
TAR	ACCEPT statement replacement.	~	~
TSA	SELECT alternative replacement.	~	~

5 ADA SELECTIVE MUTATION

The Fortran-77 mutation system, Mothra [?], uses 22 mutation operators, of which the 6 most populous account for 40 to 60% of all mutants. Recent experimental research [?, ?] has indicated that of the 22 mutation operators used by Mothra, 17 of them (including the 6 most populous) seem to be in some sense redundant; that is, test sets that are generated to kill only mutants generated from the other 5 mutant operators are very effective in killing mutants generated from the 17. *Selective mutation* is an approximation technique that selects only mutants that are truly distinct from other mutants [?]. In experimental trials, selective mutation provides almost the same coverage as non-selective mutation, with significant reductions in cost.

Specifically, the results indicate that the mutation operators that replace all operands with all syntactically legal operands add very little to the effectiveness of mutation testing. Additionally, the mutation operators that modify entire statements add very little. The 5 selective operators for Fortran-77 are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector (AND and OR) with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions. This report lists the mutation operators for Ada that should be included in the selective set.

5.1 List of Selective Operators

We leave out all operand replacement operators, and most of the statement operators. Most of the expression operators are included in the selective set. Because there has been no experience with tasking mutation operators, we leave them in. Further experimentation is needed to verify whether these are necessary.

Expression Modification Operators	
EAI	Absolute value insertion.
ENI	Neg-absolute value insertion.
EEZ	Exception on zero.
EOR	Arithmetic operator replacement.
ERR	Relational operator replacement.
EMR	Membership test replacement.
ELR	Logical operator replacement.
EUI	Unary operator insertion.
EUR	Unary operator replacement.
ESR	Subprogram operator replacement.
EEO	Exception on oVerflow.
EEU	Exception on underflow.
Coverage Operators	
CMC	Multiple Condition coverage.
Tasking Operators	
TEM	ENTRY statement modification.
TAR	ACCEPT statement replacement.
TSA	SELECT alternative replacement.

6 COMPREHENSIVE ADA MUTATION EXAMPLE

In this section, we present an example of a mutated Ada program. We show a small Ada program with all mutants displayed “in-line”, that is, with the changes shown in the text of the program. The program reads two matrices, adds and multiplies them together, and prints the results. The lines in the program are numbered (for simplicity, all text are numbered, including comments and blanks), and mutants are shown in the program.

Each mutant is represented by the modified statement in the program. The mutated statement is shown just below the original statement. Each mutated statement includes the mutation operator name (e.g., OVV, OVC), a unique integer that identifies the mutant, and the string `-->`. For example, line 43 contains the header of a `FOR` loop. The first mutant of that line is shown immediately below it. It is an OCC (Operand–Constant replaced by a Constant) mutant, number 42, and the constant 1 is replaced by the named constant `MATSIZE`. The program has 71 executable statements and 559 mutants.

```
 1      -----------------------------------------------------------
 2      --  Programmer : Jeff Offutt                             --
 3      --  Program    : Matrix                                  --
 5      --  Purpose    : Add and multiply two matrices.          --
 6      --  Date       : 2/21/94                                 --
 7      -----------------------------------------------------------
 8
 9      WITH Text_IO; USE Text_IO;
10
11      PROCEDURE Matrix IS
12
13         -- Package instantiations.
14         PACKAGE Int_IO IS NEW Integer_IO (Integer);
15         USE Int_IO;
16
17         -- Constant declarations.
18         INFILE  : CONSTANT String  := "p1.in";
19         OUTFILE : CONSTANT String  := "p1.out";
20         MATSIZE : CONSTANT Integer := 3;
21
22         -- Type declarations.
23         TYPE Matrix_Type IS ARRAY (1..MATSIZE,1..MATSIZE) OF Integer;
24
25         -- Variable declarations.
26         Input_File, Output_File : File_Type;
27         Matrix_In1, Matrix_In2  : Matrix_Type;
28         Mat_Sum, Mat_Prod       : Matrix_Type;
29
30
31         -------------------------------------------------------
32         -- Procedure : Read_Mat (Input_File : IN  File_Type;    --
33         --                        Mat        : OUT Matrix_Type)  --
34         -- Purpose   : Read a matrix from a file.              --
35         -- Params    : Input_File : The file to read from.     --
36         --           : Mat        : The matrix to read into.   --
37         -- Pre       : Input_File must be open                 --
38         --           : The matrix is MATSIZE lines             --
39         --           : MATSIZE numbers per row                 --
40         -------------------------------------------------------
41         PROCEDURE Read_Mat (Input_File : IN File_Type; Mat : OUT Matrix_Type) IS
 SZI          ==>szi_lci1 : Natural;
 SZI          ==>szi_lci2 : Natural;
```

```
42          BEGIN

  SZI          ==>szi_lci1 := 0;  -- Initialize loop counter.
43             FOR i IN 1..MATSIZE LOOP
  OCC  42      -->FOR i IN MATSIZE..MATSIZE LOOP
  OCC  43      -->FOR i IN 1..1 LOOP
  SEE 312      -->Except_On_Exec (SEE+0);  -- First SEE mutant=320 + offset.
  SRN 327      -->NULL; -- Replaces statements 43,44,45,46,47
  SRR 358      -->RETURN; -- Replaces statements 43,44,45,46,47
  SRE 383      -->EXIT; -- Replaces statements 43,44,45,46,47
  SRI 441      -->FOR i IN REVERSE 1..MATSIZE LOOP

  SZI             ==>szi_lci2 := 0;  -- Initialize loop counter.
44                FOR j IN 1..MATSIZE LOOP
  OCC  44         -->FOR j IN MATSIZE..MATSIZE LOOP
  OCC  45         -->FOR j IN i..MATSIZE LOOP
  OCC  46         -->FOR j IN 1..1 LOOP
  OCC  47         -->FOR j IN 1..i LOOP
  SEE 313         -->Except_On_Exec (SEE+1);
  SRN 328         -->NULL; -- Replaces statements 44,45,46
  SRR 359         -->RETURN; -- Replaces statements 44,45,46
  SRE 384         -->EXIT; -- Replaces statements 44,45,46
  SRI 442         -->FOR j IN REVERSE 1..MATSIZE LOOP

45                   Get (Input_File, Mat (i,j));
  OCV  26            -->Get (Output_File, Mat (i,j));
  OCC  48            -->Get (Input_File, Mat (1,j));
  OCC  49            -->Get (Input_File, Mat (j,j));
  OCC  50            -->Get (Input_File, Mat (MATSIZE,j));
  OCC  51            -->Get (Input_File, Mat (i,1));
  OCC  52            -->Get (Input_File, Mat (i,i));
  OCC  53            -->Get (Input_File, Mat (i,MATSIZE));
  OCA 158            -->Get (Input_File, Mat (Mat(i,j),j));
  OCA 159            -->Get (Input_File, Mat (i,Mat(i,j)));
  OAN 230            -->Get (Input_File, Matrix_In1 (i,j));
  OAN 231            -->Get (Input_File, Matrix_In2 (i,j));
  OAN 232            -->Get (Input_File, Mat_Sum (i,j));
  OAN 233            -->Get (Input_File, Mat_Prod (i,j));
  SEE 314            -->Except_On_Exec (SEE+2);
  SRN 329            -->NULL;
  SRR 360            -->RETURN;
  SRE 385            -->EXIT;
46                END LOOP;
  SZI 414         -->IF (szi_lci1 = 0) RAISE Mut_Trap;
  SOI 423         -->IF (szi_lci1 = 1) RAISE Mut_Trap;
  SNI 432         -->IF (szi_lci1 > 1) RAISE Mut_Trap;
47             END LOOP;
  SZI 415      -->IF (szi_lci2 = 0) RAISE Mut_Trap;
  SOI 424      -->IF (szi_lci2 = 1) RAISE Mut_Trap;
  SNI 433      -->IF (szi_lci2 > 1) RAISE Mut_Trap;
48          END Read_Mat;
49
50          ---------------------------------------------------------
51          -- Procedure : Write_Mat (Output_File : IN File_Type;   --
52          --                        Mat         : IN Matrix_Type) --
53          -- Purpose   : Write a matrix to a file.                --
```

```
54          -- Params    : Output_File : The file to write to.      --
55          --           : Mat         : The matrix to write from.  --
56          -- Pre       : Output_File must be open                 --
57          ----------------------------------------------------------
58          PROCEDURE Write_Mat (Output_File : IN File_Type; Mat : IN Matrix_Type) IS
  SZI          ==>szi_lci3 : Natural;
  SZI          ==>szi_lci4 : Natural;
59          BEGIN

  SZI          ==>szi_lci3 := 0;  -- Initialize loop counter.
60             FOR i IN 1..MATSIZE LOOP
  OCC  54      -->FOR i IN MATSIZE..MATSIZE LOOP
  OCC  55      -->FOR i IN 1..1 LOOP
  SEE 315      -->Except_On_Exec (SEE+3);
  SRN 330      -->NULL; -- Replaces statements 60-65
  SRR 361      -->RETURN; -- Replaces statements 60-65
  SRE 386      -->EXIT; -- Replaces statements 60-65
  SRI 443      -->FOR i IN REVERSE 1..MATSIZE LOOP
  EDT 522      -->FOR i IN 0..MATSIZE LOOP
  EDT 523      -->FOR i IN 2..MATSIZE LOOP
  EDT 524      -->FOR i IN 0..MATSIZE-1 LOOP
  EDT 525      -->FOR i IN 0..MATSIZE+1 LOOP

  SZI             ==>szi_lci4 := 0;  -- Initialize loop counter.
61                FOR j IN 1..MATSIZE LOOP
  OCC  56         -->FOR j IN MATSIZE..MATSIZE LOOP
  OCC  57         -->FOR j IN i..MATSIZE LOOP
  OCC  58         -->FOR j IN 1..1 LOOP
  OCC  59         -->FOR j IN 1..i LOOP
  SEE 316         -->Except_On_Exec (SEE+4);
  SRN 331         -->NULL; -- Replaces statements 61-63
  SRR 362         -->RETURN; -- Replaces statements 61-63
  SRE 387         -->EXIT; -- Replaces statements 61-63
  SRI 444         -->FOR j IN REVERSE 1..MATSIZE LOOP
  SES 450         -->END LOOP; -- Line 63 moved below 61.
  EDT 526         -->FOR j IN 0..MATSIZE LOOP
  EDT 527         -->FOR j IN 2..MATSIZE LOOP
  EDT 528         -->FOR j IN 1..MATSIZE-1 LOOP
  EDT 529         -->FOR j IN 1..MATSIZE+1 LOOP

62                   Put (Output_File, Mat (i,j));
  OCV  27            -->Put (Input_File, Mat (i,j));
  OCC  60            -->Put (Output_File, Mat (1,j));
  OCC  61            -->Put (Output_File, Mat (j,j));
  OCC  62            -->Put (Output_File, Mat (MATSIZE,j));
  OCC  63            -->Put (Output_File, Mat (i,1));
  OCC  64            -->Put (Output_File, Mat (i,i));
  OCC  65            -->Put (Output_File, Mat (i,MATSIZE));
  OCA 160            -->Put (Output_File, Mat (Mat(i,j),j));
  OCA 161            -->Put (Output_File, Mat (i,Mat(i,j)));
  OAC 201            -->Put (Output_File, i);
  OAC 202            -->Put (Output_File, j);
  OAC 203            -->Put (Output_File, 1);
  OAN 234            -->Put (Output_File, Matrix_In1 (i,j));
  OAN 235            -->Put (Output_File, Matrix_In2 (i,j));
  OAN 236            -->Put (Output_File, Mat_Sum (i,j));
```

```
  OAN 237            -->Put (Output_File, Mat_Prod (i,j));
  SEE 317            -->Except_On_Exec (SEE+5);
  SRN 332            -->NULL;
  SRR 363            -->RETURN;
  SRE 388            -->EXIT;
  EAI 458            -->Put (Output_File, ABS(Mat (i,j)));
  ENI 468            -->Put (Output_File, -ABS(Mat (i,j)));
  EEZ 478            -->Put (Output_File, EEZ(Mat (i,j)));
  EUI 506            -->Put (Output_File, -Mat (i,j));
  EDT 530            -->Put (Output_File, Mat (i,j)+1);
  EDT 531            -->Put (Output_File, Mat (i,j)-1);

63                 END LOOP;
  SZI 416          -->IF (szi_lci3 = 0) RAISE Mut_Trap;
  SOI 425          -->IF (szi_lci3 = 1) RAISE Mut_Trap;
  SNI 434          -->IF (szi_lci3 > 1) RAISE Mut_Trap;
64                 New_Line (Output_File);
  OCV  28          -->New_Line (Input_File);
  SEE 318          -->Except_On_Exec (SEE+6);
  SRN 333          -->NULL;
  SRR 364          -->RETURN;
  SRE 389          -->EXIT;
  SES 451          -->END LOOP; -- Line 63 moved below 64.

65              END LOOP;
  SZI 417          -->IF (szi_lci4 = 0) RAISE Mut_Trap;
  SOI 426          -->IF (szi_lci4 = 1) RAISE Mut_Trap;
  SNI 435          -->IF (szi_lci4 > 1) RAISE Mut_Trap;
66           END Write_Mat;
67
68           ---------------------------------------------------------
69           -- Function : Add_Mat (Mat1, Mat2 : IN Matrix_Type)     --
70           -- Purpose  : Add two matrices                          --
71           -- Params   : Mat1, Mat2 : The matrices to add.         --
72           -- Return   : The matrix sum.                           --
73           -- Pre      : Mat1 and Mat2 are initialized.            --
74           ---------------------------------------------------------
75           FUNCTION Add_Mat (Mat1, Mat2 : IN Matrix_Type) RETURN Matrix_Type IS
76              rslt_mat : Matrix_Type;
  SZI           ==>szi_lci5 : Natural;
  SZI           ==>szi_lci6 : Natural;
77           BEGIN

  SZI           ==>szi_lci5 := 0;  -- Initialize loop counter.
78              FOR i IN 1..MATSIZE LOOP
  OCC  66       -->FOR i IN MATSIZE..MATSIZE LOOP
  OCC  67       -->FOR i IN 1..1 LOOP
  SEE 319       -->Except_On_Exec (SEE+7);
  SRN 334       -->NULL; -- Replaces statements 78-82
  SRR 365       -->RETURN; -- Replaces statements 78-82
  SRR 366       -->RETURN (rslt_mat); -- Replaces statements 78-82
  SRE 390       -->EXIT; -- Replaces statements 78-82
  SRI 445       -->FOR i IN REVERSE 1..MATSIZE LOOP
  EDT 532       -->FOR i IN 0..MATSIZE LOOP
  EDT 533       -->FOR i IN 2..MATSIZE LOOP
  EDT 534       -->FOR i IN 1..MATSIZE-1 LOOP
```

```
  EDT 535      -->FOR i IN 1..MATSIZE+1 LOOP

  SZI             ==>szi_lci6 := 0;  -- Initialize loop counter.
79                FOR j IN 1..MATSIZE LOOP
  OCC  68         -->FOR j IN MATSIZE..MATSIZE LOOP
  OCC  69         -->FOR j IN i..MATSIZE LOOP
  OCC  70         -->FOR j IN 1..1 LOOP
  OCC  71         -->FOR j IN 1..i LOOP
  SEE 320         -->Except_On_Exec (SEE+8);
  SRN 335         -->NULL; -- Replaces statements 79-81
  SRR 367         -->RETURN; -- Replaces statements 79-81
  SRR 368         -->RETURN (rslt_mat); -- Replaces statements 79-81
  SRE 391         -->EXIT; -- Replaces statements 79-81
  SRI 446         -->FOR j IN REVERSE 1..MATSIZE LOOP
  SES 452         -->END LOOP; -- Line 81 moved below 79.
  EDT 536         -->FOR j IN 0..MATSIZE LOOP
  EDT 537         -->FOR j IN 2..MATSIZE LOOP
  EDT 538         -->FOR j IN 1..MATSIZE-1 LOOP
  EDT 539         -->FOR j IN 1..MATSIZE+1 LOOP

80                   rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,j);
  OCC  72            -->rslt_mat (MATSIZE,j) := Mat1 (i,j) + Mat2 (i,j);
  OCC  73            -->rslt_mat (1,j) := Mat1 (i,j) + Mat2 (i,j);
  OCC  74            -->rslt_mat (j,j) := Mat1 (i,j) + Mat2 (i,j);
  OCC  75            -->rslt_mat (i,MATSIZE) := Mat1 (i,j) + Mat2 (i,j);
  OCC  76            -->rslt_mat (i,1) := Mat1 (i,j) + Mat2 (i,j);
  OCC  77            -->rslt_mat (i,i) := Mat1 (i,j) + Mat2 (i,j);
  OCC  78            -->rslt_mat (i,j) := Mat1 (MATSIZE,j) + Mat2 (i,j);
  OCC  79            -->rslt_mat (i,j) := Mat1 (1,j) + Mat2 (i,j);
  OCC  80            -->rslt_mat (i,j) := Mat1 (j,j) + Mat2 (i,j);
  OCC  81            -->rslt_mat (i,j) := Mat1 (i,MATSIZE) + Mat2 (i,j);
  OCC  82            -->rslt_mat (i,j) := Mat1 (i,1) + Mat2 (i,j);
  OCC  83            -->rslt_mat (i,j) := Mat1 (i,i) + Mat2 (i,j);
  OCC  84            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (MATSIZE,j);
  OCC  85            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (1,j);
  OCC  86            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (j,j);
  OCC  87            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,MATSIZE);
  OCC  88            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,1);
  OCC  89            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,i);
  OCA 162            -->rslt_mat (rslt_mat(i,j),j) := Mat1 (i,j) + Mat2 (i,j);
  OCA 163            -->rslt_mat (i,rslt_mat(i,j)) := Mat1 (i,j) + Mat2 (i,j);
  OCA 164            -->rslt_mat (Mat1(i,j),j) := Mat1 (i,j) + Mat2 (i,j);
  OCA 165            -->rslt_mat (i,j) := Mat1 (i,Mat1(i,j)) + Mat2 (i,j);
  OCA 166            -->rslt_mat (Mat2(i,j),j) := Mat1 (i,j) + Mat2 (i,j);
  OCA 167            -->rslt_mat (i,Mat2(i,j)) := Mat1 (i,j) + Mat2 (i,j);
  OCA 168            -->rslt_mat (i,j) := Mat1 (rslt_mat(i,j),j) + Mat2 (i,j);
  OCA 169            -->rslt_mat (i,j) := Mat1 (i,rslt_mat(i,j)) + Mat2 (i,j);
  OCA 170            -->rslt_mat (i,j) := Mat1 (Mat1(i,j),j) + Mat2 (i,j);
  OCA 171            -->rslt_mat (i,j) := Mat1 (i,Mat1(i,j)) + Mat2 (i,j);
  OCA 172            -->rslt_mat (i,j) := Mat1 (Mat2(i,j),j) + Mat2 (i,j);
  OCA 173            -->rslt_mat (i,j) := Mat1 (i,Mat2(i,j)) + Mat2 (i,j);
  OCA 174            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (rslt_mat(i,j),j);
  OCA 175            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,rslt_mat(i,j));
  OCA 176            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (Mat1(i,j),j);
  OCA 177            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,Mat1(i,j));
  OCA 178            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (Mat2(i,j),j);
```

```
OCA 179            -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,Mat2(i,j));
OAC 204            -->rslt_mat (i,j) := i + Mat2 (i,j);
OAC 205            -->rslt_mat (i,j) := j + Mat2 (i,j);
OAC 206            -->rslt_mat (i,j) := MATSIZE + Mat2 (i,j);
OAC 207            -->rslt_mat (i,j) := 1 + Mat2 (i,j);
OAC 208            -->rslt_mat (i,j) := Mat1 (i,j) + i;
OAC 209            -->rslt_mat (i,j) := Mat1 (i,j) + j;
OAC 210            -->rslt_mat (i,j) := Mat1 (i,j) + MATSIZE;
OAC 211            -->rslt_mat (i,j) := Mat1 (i,j) + 1;
OAA 222            -->rslt_mat (i,j) := rslt_mat (i,j) + Mat2 (i,j);
OAA 223            -->rslt_mat (i,j) := Mat2 (i,j) + Mat2 (i,j);
OAA 224            -->rslt_mat (i,j) := Mat1 (i,j) + rslt_mat (i,j);
OAA 225            -->rslt_mat (i,j) := Mat1 (i,j) + Mat1 (i,j);
OAN 238            -->Matrix_In1 (i,j) := Mat1 (i,j) + Mat2 (i,j);
OAN 239            -->Matrix_In2 (i,j) := Mat1 (i,j) + Mat2 (i,j);
OAN 240            -->Mat_Sum (i,j) := Mat1 (i,j) + Mat2 (i,j);
OAN 241            -->Mat_Prod (i,j) := Mat1 (i,j) + Mat2 (i,j);
OAN 242            -->rslt_mat (i,j) := Matrix_In1 (i,j) + Mat2 (i,j);
OAN 243            -->rslt_mat (i,j) := Matrix_In2 (i,j) + Mat2 (i,j);
OAN 244            -->rslt_mat (i,j) := Mat_Sum (i,j) + Mat2 (i,j);
OAN 245            -->rslt_mat (i,j) := Mat_Prod (i,j) + Mat2 (i,j);
OAN 246            -->rslt_mat (i,j) := Mat2 (i,j) + Mat2 (i,j);
OAN 247            -->rslt_mat (i,j) := rslt_mat (i,j) + Mat2 (i,j);
OAN 248            -->rslt_mat (i,j) := Mat1 (i,j) + Matrix_In1 (i,j);
OAN 249            -->rslt_mat (i,j) := Mat1 (i,j) + Matrix_In2 (i,j);
OAN 250            -->rslt_mat (i,j) := Mat1 (i,j) + Mat_Sum (i,j);
OAN 251            -->rslt_mat (i,j) := Mat1 (i,j) + Mat_Prod (i,j);
OAN 252            -->rslt_mat (i,j) := Mat1 (i,j) + Mat1 (i,j);
OAN 253            -->rslt_mat (i,j) := Mat1 (i,j) + rslt_mat (i,j);
SEE 321            -->Except_On_Exec (SEE+9);
SRN 336            -->NULL;
SRR 369            -->RETURN;
SRR 370            -->RETURN (rslt_mat);
SRE 392            -->EXIT;
EAI 459            -->rslt_mat (i,j) := ABS(Mat1 (i,j) + Mat2 (i,j));
EAI 460            -->rslt_mat (i,j) := ABS(Mat1 (i,j)) + Mat2 (i,j);
EAI 461            -->rslt_mat (i,j) := Mat1 (i,j) + ABS(Mat2 (i,j));
ENI 469            -->rslt_mat (i,j) := -ABS(Mat1 (i,j) + Mat2 (i,j));
ENI 470            -->rslt_mat (i,j) := -ABS(Mat1 (i,j)) + Mat2 (i,j);
ENI 471            -->rslt_mat (i,j) := Mat1 (i,j) + -ABS(Mat2 (i,j));
EEZ 479            -->rslt_mat (i,j) := EEZ(Mat1 (i,j) + Mat2 (i,j));
EEZ 480            -->rslt_mat (i,j) := EEZ(Mat1 (i,j)) + Mat2 (i,j);
EEZ 481            -->rslt_mat (i,j) := Mat1 (i,j) + EEZ(Mat2 (i,j));
EOR 488            -->rslt_mat (i,j) := Mat1 (i,j) - Mat2 (i,j);
EOR 489            -->rslt_mat (i,j) := Mat1 (i,j) * Mat2 (i,j);
EOR 490            -->rslt_mat (i,j) := Mat1 (i,j) / Mat2 (i,j);
EOR 491            -->rslt_mat (i,j) := Mat1 (i,j) MOD Mat2 (i,j);
EOR 492            -->rslt_mat (i,j) := Mat1 (i,j) REM Mat2 (i,j);
EOR 493            -->rslt_mat (i,j) := Mat1 (i,j) ** Mat2 (i,j);
EUI 507            -->rslt_mat (i,j) := -(Mat1 (i,j) + Mat2 (i,j));
EUI 508            -->rslt_mat (i,j) := -Mat1 (i,j) + Mat2 (i,j);
EUI 509            -->rslt_mat (i,j) := Mat1 (i,j) + -Mat2 (i,j);
EDT 540            -->rslt_mat (i,j) := Mat1 (i,j)+1 + Mat2 (i,j);
EDT 541            -->rslt_mat (i,j) := Mat1 (i,j)-1 + Mat2 (i,j);
EEO 556            -->rslt_mat (i,j) := EEO(Mat1 (i,j) + Mat2 (i,j));
EEU 558            -->rslt_mat (i,j) := EEU(Mat1 (i,j) + Mat2 (i,j));
```

```
81                END LOOP;
  SZI 418         -->IF (szi_lci5 = 0) RAISE Mut_Trap;
  SOI 427         -->IF (szi_lci5 = 1) RAISE Mut_Trap;
  SNI 436         -->IF (szi_lci5 > 1) RAISE Mut_Trap;
82             END LOOP;
  SZI 419      -->IF (szi_lci6 = 0) RAISE Mut_Trap;
  SOI 428      -->IF (szi_lci6 = 1) RAISE Mut_Trap;
  SNI 437      -->IF (szi_lci6 > 1) RAISE Mut_Trap;
83             RETURN (rslt_mat);
  OAN 254      -->RETURN (Matrix_In1);
  OAN 255      -->RETURN (Matrix_In2);
  OAN 256      -->RETURN (Mat_Sum);
  OAN 257      -->RETURN (Mat_Prod);
  OAN 258      -->RETURN (Mat1);
  OAN 259      -->RETURN (Mat2);
  --SEE -- no SEE mutant, function must have a RETURN.
  --SRN -- no SRN mutant, function must have a RETURN.
  SRR 371      -->RETURN;
  --SRE -- no SRE mutant, function must have a RETURN.
  SES 453      -->END LOOP; -- Line 82 moved below 83.

84          END Add_Mat;
85
86          ---------------------------------------------------------
87          -- Function : Multiply_Mat (Mat1, Mat2 : IN Matrix_Type)--
88          -- Purpose  : Multiply two matrices                     --
89          -- Params   : Mat1, Mat2 : The matrices to multiply.    --
90          -- Return   : The matrix multiplied.                    --
91          -- Pre      : Mat1 and Mat2 are initialized.            --
92          ---------------------------------------------------------
93          FUNCTION Multiply_Mat (Mat1, Mat2 : IN Matrix_Type) RETURN Matrix_Type IS
94             rslt_mat : Matrix_Type;
95             tmp_sum  : Integer;
  SZI          ==>szi_lci7 : Natural;
  SZI          ==>szi_lci8 : Natural;
  SZI          ==>szi_lci9 : Natural;
96          BEGIN

  SZI          ==>szi_lci7 := 0;  -- Initialize loop counter.
97             FOR i IN 1..MATSIZE LOOP
  OCV  29      -->FOR i IN tmp_sum..MATSIZE LOOP
  OCV  30      -->FOR i IN 1..tmp_sum LOOP
  OCC  90      -->FOR i IN MATSIZE..MATSIZE LOOP
  OCC  91      -->FOR i IN 0..MATSIZE LOOP
  OCC  92      -->FOR i IN 1..1 LOOP
  OCC  93      -->FOR i IN 1..0 LOOP
  SEE 322      -->Except_On_Exec (SEE+10);
  SRN 337      -->NULL; -- Replaces statements 97-105
  SRR 372      -->RETURN; -- Replaces statements 97-105
  SRR 373      -->RETURN (rslt_mat); -- Replaces statements 97-105
  SRE 393      -->EXIT; -- Replaces statements 97-105
  SRI 447      -->FOR i IN REVERSE 1..MATSIZE LOOP
  --EDT           FOR i IN 0..MATSIZE LOOP -- equivalent to OCC 91
  EDT 542      -->FOR i IN 2..MATSIZE LOOP
  EDT 543      -->FOR i IN 1..MATSIZE-1 LOOP
```

```
  EDT 544      -->FOR i IN 1..MATSIZE+1 LOOP

  SZI             ==>szi_lci8 := 0;  -- Initialize loop counter.
 98               FOR j IN 1..MATSIZE LOOP
  OCV  31         -->FOR j IN tmp_sum..MATSIZE LOOP
  OCV  32         -->FOR j IN 1..tmp_sum LOOP
  OCC  94         -->FOR j IN MATSIZE..MATSIZE LOOP
  OCC  95         -->FOR j IN 0..MATSIZE LOOP
  OCC  96         -->FOR j IN i..MATSIZE LOOP
  OCC  97         -->FOR j IN 1..1 LOOP
  OCC  98         -->FOR j IN 1..0 LOOP
  OCC  99         -->FOR j IN 1..i LOOP
  SEE 323         -->Except_On_Exec (SEE+11);
  SRN 338         -->NULL; -- Replaces statements 98-104
  SRR 374         -->RETURN; -- Replaces statements 98-104
  SRR 375         -->RETURN (rslt_mat); -- Replaces statements 98-104
  SRE 394         -->EXIT; -- Replaces statements 98-104
  SRI 448         -->FOR j IN REVERSE 1..MATSIZE LOOP
  --EDT              FOR j IN 0..MATSIZE LOOP  -- equiv to OCC 95
  EDT 545         -->FOR j IN 2..MATSIZE LOOP
  EDT 546         -->FOR j IN 1..MATSIZE-1 LOOP
  EDT 547         -->FOR j IN 1..MATSIZE+1 LOOP

 99                  tmp_sum := 0;
  OVA  14            -->Mat1(i,k) := 0;
  OVA  15            -->Mat2(k,j) := 0;
  OVA  16            -->rslt_mat(i,j) := 0;
  OCV  33            -->tmp_sum := tmp_sum;
  OCC 100            -->tmp_sum := i;
  OCC 101            -->tmp_sum := j;
  OCC 102            -->tmp_sum := 1;
  OCC 103            -->tmp_sum := MATSIZE;
  SRN 339            -->NULL;
  SRR 376            -->RETURN;
  SRR 377            -->RETURN (rslt_mat);
  SRE 395            -->EXIT;
  EDT 548            -->tmp_sum := -1;

  SZI                ==>szi_lci9 := 0;  -- Initialize loop counter.
100                  FOR k IN 1..MATSIZE LOOP
  OCV  34            -->FOR k IN tmp_sum..MATSIZE LOOP
  OCV  35            -->FOR k IN 1..tmp_sum LOOP
  OCC 104            -->FOR k IN MATSIZE..MATSIZE LOOP
  OCC 105            -->FOR k IN 0..MATSIZE LOOP
  OCC 106            -->FOR k IN i..MATSIZE LOOP
  OCC 107            -->FOR k IN j..MATSIZE LOOP
  OCC 108            -->FOR k IN 1..1 LOOP
  OCC 109            -->FOR k IN 1..0 LOOP
  OCC 110            -->FOR k IN 1..i LOOP
  OCC 111            -->FOR k IN 1..j LOOP
  OCC 112            -->FOR k IN 1..MATSIZE LOOP
  SEE 324            -->Except_On_Exec (SEE+12);
  SRN 340            -->NULL; -- Replaces statements 100-102
  SRR 378            -->RETURN; -- Replaces statements 100-102
  SRR 379            -->RETURN (rslt_mat); -- Replaces statements 100-102
  SRE 396            -->EXIT; -- Replaces statements 100-102
```

```
   SRI 449            -->FOR k IN REVERSE 1..MATSIZE LOOP
   SES 454            -->END LOOP; -- Line 102 moved below 100.
   EDT 549            -->FOR k IN 2..MATSIZE LOOP
   EDT 550            -->FOR k IN 1..MATSIZE-1 LOOP
   EDT 551            -->FOR k IN 1..MATSIZE+1 LOOP

101                      tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,j);
   OVA  17               -->Mat1(i,k) := tmp_sum + Mat1 (i,k) * Mat2 (k,j);
   OVA  18               -->Mat2(k,j) := tmp_sum + Mat1 (i,k) * Mat2 (k,j);
   OVA  19               -->rslt_mat(i,j) := tmp_sum + Mat1 (i,k) * Mat2 (k,j);
   OVA  20               -->tmp_sum := Mat1(i,k) + Mat1 (i,k) * Mat2 (k,j);
   OVA  21               -->tmp_sum := Mat2(k,j) + Mat1 (i,k) * Mat2 (k,j);
   OVA  22               -->tmp_sum := rslt_mat(i,j) + Mat1 (i,k) * Mat2 (k,j);
   OCV  36               -->tmp_sum := tmp_sum + Mat1 (tmp_sum,k) * Mat2 (k,j);
   OCV  37               -->tmp_sum := tmp_sum + Mat1 (i,tmp_sum) * Mat2 (k,j);
   OCV  38               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (tmp_sum,j);
   OCV  39               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,tmp_sum);
   OCC 113               -->tmp_sum := tmp_sum + Mat1 (j,k) * Mat2 (k,j);
   OCC 114               -->tmp_sum := tmp_sum + Mat1 (1,k) * Mat2 (k,j);
   OCC 115               -->tmp_sum := tmp_sum + Mat1 (MATSIZE,k) * Mat2 (k,j);
   OCC 116               -->tmp_sum := tmp_sum + Mat1 (0,k) * Mat2 (k,j);
   OCC 117               -->tmp_sum := tmp_sum + Mat1 (k,k) * Mat2 (k,j);
   OCC 118               -->tmp_sum := tmp_sum + Mat1 (i,i) * Mat2 (k,j);
   OCC 119               -->tmp_sum := tmp_sum + Mat1 (i,j) * Mat2 (k,j);
   OCC 120               -->tmp_sum := tmp_sum + Mat1 (i,1) * Mat2 (k,j);
   OCC 121               -->tmp_sum := tmp_sum + Mat1 (i,MATSIZE) * Mat2 (k,j);
   OCC 122               -->tmp_sum := tmp_sum + Mat1 (i,0) * Mat2 (k,j);
   OCC 123               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (i,j);
   OCC 124               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (j,j);
   OCC 125               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (1,j);
   OCC 126               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (MATSIZE,j);
   OCC 127               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (0,j);
   OCC 128               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,i);
   OCC 129               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,1);
   OCC 130               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,MATSIZE);
   OCC 131               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,0);
   OCC 132               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,k);
   OVC 153               -->tmp_sum := i + Mat1 (i,k) * Mat2 (k,j);
   OVC 154               -->tmp_sum := j + Mat1 (i,k) * Mat2 (k,j);
   OVC 155               -->tmp_sum := 1 + Mat1 (i,k) * Mat2 (k,j);
   OVC 156               -->tmp_sum := MATSIZE + Mat1 (i,k) * Mat2 (k,j);
   OVC 157               -->tmp_sum := 0 + Mat1 (i,k) * Mat2 (k,j);
   OCA 180               -->tmp_sum := tmp_sum + Mat1 (Mat1(i,k),k) * Mat2 (k,j);
   OCA 181               -->tmp_sum := tmp_sum + Mat1 (i,Mat1(i,k)) * Mat2 (k,j);
   OCA 182               -->tmp_sum := tmp_sum + Mat1 (Mat2(k,j),k) * Mat2 (k,j);
   OCA 183               -->tmp_sum := tmp_sum + Mat1 (i,Mat2(k,j)) * Mat2 (k,j);
   OCA 184               -->tmp_sum := tmp_sum + Mat1 (rslt_mat(i,j),k) * Mat2 (k,j);
   OCA 185               -->tmp_sum := tmp_sum + Mat1 (i,rslt_mat(i,j)) * Mat2 (k,j);
   OCA 186               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (Mat1(i,k),j);
   OCA 187               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,Mat1(i,k));
   OCA 188               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (Mat2(k,j),j);
   OCA 189               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,Mat2(k,j));
   OCA 190               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (rslt_mat(i,j),j);
   OCA 191               -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat2 (k,rslt_mat(i,j));
   OAV 198               -->tmp_sum := tmp_sum + tmp_sum * Mat2 (k,j);
   OAV 199               -->tmp_sum := tmp_sum + Mat1 (i,k) * tmp_sum;
```

```
OAC 212                -->tmp_sum := tmp_sum + i * Mat2 (k,j);
OAC 213                -->tmp_sum := tmp_sum + j * Mat2 (k,j);
OAC 214                -->tmp_sum := tmp_sum + 1 * Mat2 (k,j);
OAC 215                -->tmp_sum := tmp_sum + 0 * Mat2 (k,j);
OAC 216                -->tmp_sum := tmp_sum + MATSIZE * Mat2 (k,j);
OAC 217                -->tmp_sum := tmp_sum + Mat1 (i,k) * i;
OAC 218                -->tmp_sum := tmp_sum + Mat1 (i,k) * j;
OAC 219                -->tmp_sum := tmp_sum + Mat1 (i,k) * 1;
OAC 220                -->tmp_sum := tmp_sum + Mat1 (i,k) * 0;
OAC 221                -->tmp_sum := tmp_sum + Mat1 (i,k) * MATSIZE;
OAA 226                -->tmp_sum := tmp_sum + rslt_mat (i,k) * Mat2 (k,j);
OAA 227                -->tmp_sum := tmp_sum + Mat2 (i,k) * Mat2 (k,j);
OAA 228                -->tmp_sum := tmp_sum + Mat1 (i,k) * rslt_mat (k,j);
OAA 229                -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat1 (k,j);
OAN 260                -->tmp_sum := tmp_sum + Matrix_In1 (i,k) * Mat2 (k,j);
OAN 261                -->tmp_sum := tmp_sum + Matrix_In2 (i,k) * Mat2 (k,j);
OAN 262                -->tmp_sum := tmp_sum + Mat_Sum (i,k) * Mat2 (k,j);
OAN 263                -->tmp_sum := tmp_sum + Mat_Prod (i,k) * Mat2 (k,j);
OAN 264                -->tmp_sum := tmp_sum + Mat2 (i,k) * Mat2 (k,j);
OAN 265                -->tmp_sum := tmp_sum + rslt_mat (i,k) * Mat2 (k,j);
OAN 266                -->tmp_sum := tmp_sum + Mat1 (i,k) * Matrix_In1 (k,j);
OAN 267                -->tmp_sum := tmp_sum + Mat1 (i,k) * Matrix_In1 (k,j);
OAN 268                -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat_Sum (k,j);
OAN 269                -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat_Prod (k,j);
OAN 270                -->tmp_sum := tmp_sum + Mat1 (i,k) * Mat1 (k,j);
OAN 271                -->tmp_sum := tmp_sum + Mat1 (i,k) * rslt_mat (k,j);
SRN 341                -->NULL;
SRR 380                -->RETURN;
SRR 381                -->RETURN (rslt_mat);
SRE 397                -->EXIT;
EAI 462                -->tmp_sum := ABS(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
EAI 463                -->tmp_sum := ABS(tmp_sum) + Mat1 (i,k) * Mat2 (k,j);
EAI 464                -->tmp_sum := tmp_sum + ABS(Mat1 (i,k) * Mat2 (k,j));
EAI 465                -->tmp_sum := tmp_sum + ABS(Mat1 (i,k)) * Mat2 (k,j);
EAI 466                -->tmp_sum := tmp_sum + Mat1 (i,k) * ABS(Mat2 (k,j));
ENI 472                -->tmp_sum := -ABS(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
ENI 473                -->tmp_sum := -ABS(tmp_sum) + Mat1 (i,k) * Mat2 (k,j);
ENI 474                -->tmp_sum := tmp_sum + -ABS(Mat1 (i,k) * Mat2 (k,j));
ENI 475                -->tmp_sum := tmp_sum + -ABS(Mat1 (i,k)) * Mat2 (k,j);
ENI 476                -->tmp_sum := tmp_sum + Mat1 (i,k) * -ABS(Mat2 (k,j));
EEZ 482                -->tmp_sum := EEZ(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
EEZ 483                -->tmp_sum := EEZ(tmp_sum) + Mat1 (i,k) * Mat2 (k,j);
EEZ 484                -->tmp_sum := tmp_sum + EEZ(Mat1 (i,k) * Mat2 (k,j));
EEZ 485                -->tmp_sum := tmp_sum + EEZ(Mat1 (i,k)) * Mat2 (k,j);
EEZ 486                -->tmp_sum := tmp_sum + Mat1 (i,k) * EEZ(Mat2 (k,j));
EOR 494                -->tmp_sum := tmp_sum - Mat1 (i,k) * Mat2 (k,j);
EOR 495                -->tmp_sum := tmp_sum * Mat1 (i,k) * Mat2 (k,j);
EOR 496                -->tmp_sum := tmp_sum / Mat1 (i,k) * Mat2 (k,j);
EOR 497                -->tmp_sum := tmp_sum MOD Mat1 (i,k) * Mat2 (k,j);
EOR 498                -->tmp_sum := tmp_sum REM Mat1 (i,k) * Mat2 (k,j);
EOR 499                -->tmp_sum := tmp_sum ** Mat1 (i,k) * Mat2 (k,j);
EOR 500                -->tmp_sum := tmp_sum + Mat1 (i,k) + Mat2 (k,j);
EOR 501                -->tmp_sum := tmp_sum + Mat1 (i,k) - Mat2 (k,j);
EOR 502                -->tmp_sum := tmp_sum + Mat1 (i,k) / Mat2 (k,j);
EOR 503                -->tmp_sum := tmp_sum + Mat1 (i,k) MOD Mat2 (k,j);
EOR 504                -->tmp_sum := tmp_sum + Mat1 (i,k) REM Mat2 (k,j);
```

```
   EOR 505               -->tmp_sum := tmp_sum + Mat1 (i,k) ** Mat2 (k,j);
   EUI 510               -->tmp_sum := -(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
   EUI 511               -->tmp_sum := -tmp_sum + Mat1 (i,k) * Mat2 (k,j);
   EUI 512               -->tmp_sum := tmp_sum + -(Mat1 (i,k) * Mat2 (k,j));
   EUI 513               -->tmp_sum := tmp_sum + -Mat1 (i,k) * Mat2 (k,j);
   EUI 514               -->tmp_sum := tmp_sum + Mat1 (i,k) * -Mat2 (k,j);
   EDT 552               -->tmp_sum := (tmp_sum + Mat1 (i,k) * Mat2 (k,j))-1;
   EDT 553               -->tmp_sum := (tmp_sum + Mat1 (i,k) * Mat2 (k,j))+1;
   EEO 557               -->tmp_sum := EEO(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
   EEU 559               -->tmp_sum := EEU(tmp_sum + Mat1 (i,k) * Mat2 (k,j));

102                    END LOOP;
   SZI 420             -->IF (szi_lci7 = 0) RAISE Mut_Trap;
   SOI 429             -->IF (szi_lci7 = 1) RAISE Mut_Trap;
   SNI 438             -->IF (szi_lci7 > 1) RAISE Mut_Trap;
   SES 456             -->END LOOP; -- Line 104 moved below 102.

103                 rslt_mat (i,j) := tmp_sum;
   OVA  23          -->rslt_mat (i,j) := Mat1(i,k);
   OVA  24          -->rslt_mat (i,j) := Mat2(k,j);
   OVA  25          -->rslt_mat (i,j) := rslt_mat(i,j);
   OCV  40          -->rslt_mat (tmp_sum,j) := tmp_sum;
   OCV  41          -->rslt_mat (i,tmp_sum) := tmp_sum;
   OCC 133          -->rslt_mat (j,j) := tmp_sum;
   OCC 134          -->rslt_mat (1,j) := tmp_sum;
   OCC 135          -->rslt_mat (MATSIZE,j) := tmp_sum;
   OCC 136          -->rslt_mat (0,j) := tmp_sum;
   OCC 137          -->rslt_mat (i,i) := tmp_sum;
   OCC 138          -->rslt_mat (i,1) := tmp_sum;
   OCC 139          -->rslt_mat (i,MATSIZE) := tmp_sum;
   OCC 140          -->rslt_mat (i,0) := tmp_sum;
   OCA 192          -->rslt_mat (Mat1(i,k),j) := tmp_sum;
   OCA 193          -->rslt_mat (Mat2(k,j),j) := tmp_sum;
   OCA 194          -->rslt_mat (rsltmat(i,j),j) := tmp_sum;
   OCA 195          -->rslt_mat (i,Mat1(i,k)) := tmp_sum;
   OCA 196          -->rslt_mat (i,Mat2(k,j)) := tmp_sum;
   OCA 197          -->rslt_mat (i,rsltmat(i,j)) := tmp_sum;
   OAV 200          -->tmp_sum := tmp_sum;
   OAN 272          -->Matrix_In1 (i,j) := tmp_sum;
   OAN 273          -->Matrix_In2 (i,j) := tmp_sum;
   OAN 274          -->Mat_Sum (i,j) := tmp_sum;
   OAN 275          -->Mat_Prod (i,j) := tmp_sum;
   SEE 325          -->Except_On_Exec (SEE+13);
   SES 455          -->END LOOP; -- Line 102 moved below 103.
   EAI 467          -->rslt_mat (i,j) := ABS(tmp_sum);
   ENI 477          -->rslt_mat (i,j) := -ABS(tmp_sum);
   EEZ 487          -->rslt_mat (i,j) := EEZ(tmp_sum);
   EUI 515          -->rslt_mat (i,j) := -tmp_sum;
   EDT 554          -->rslt_mat (i,j) := tmp_sum-1;
   EDT 555          -->rslt_mat (i,j) := tmp_sum+1;

104                 END LOOP;
   SZI 421          -->IF (szi_lci8 = 0) RAISE Mut_Trap;
   SOI 430          -->IF (szi_lci8 = 1) RAISE Mut_Trap;
   SNI 439          -->IF (szi_lci8 > 1) RAISE Mut_Trap;
105              END LOOP;
```

```
   SZI 422      -->IF (szi_lci9 = 0) RAISE Mut_Trap;
   SOI 431      -->IF (szi_lci9 = 1) RAISE Mut_Trap;
   SNI 440      -->IF (szi_lci9 > 1) RAISE Mut_Trap;
106             RETURN (rslt_mat);
   OAN 276      -->RETURN (Matrix_In1);
   OAN 277      -->RETURN (Matrix_In2);
   OAN 278      -->RETURN (Mat_Sum);
   OAN 279      -->RETURN (Mat_Prod);
   OAN 280      -->RETURN (Mat1);
   OAN 281      -->RETURN (Mat2);
   --SEE -- no SEE mutant, function must have a RETURN.
   --SRN -- no SRN mutant, function must have a RETURN.
   SRR 382      -->RETURN;
   --SRE -- no SRE mutant, function must have a RETURN.
   SES 457      -->END LOOP; -- Line 105 moved below 106.

107          END Multiply_Mat;
108
109
110       ---------------------------------------------------------
111       -- Main body of matrix                                 --
112       -- Open files, read, and and multiply the two matrices. --
113       -- Close the files.                                    --
114       ---------------------------------------------------------
115       BEGIN  -- Matrix
116          Open   (Input_File,  In_File,  INFILE);
   OVV   1   -->Open   (Output_File,  In_File,  INFILE);
   OCC 141   -->Open   (Input_File,  In_File,  OUTFILE);
   OCC 142   -->Open   (Input_File,  In_File,  "Sum of Matrices");
   OCC 143   -->Open   (Input_File,  In_File,  "Product of Matrices");
   SEE 326   -->Except_On_Exec (SEE+14);
   SRN 342   -->NULL;
   SRE 398   -->EXIT;
117          Create (Output_File, Out_File, OUTFILE);
   OVV   2   -->Create (Input_File, Out_File, OUTFILE);
   OCC 144   -->Create (Output_File, Out_File, INFILE);
   OCC 145   -->Create (Output_File, Out_File, "Sum of Matrices");
   OCC 146   -->Create (Output_File, Out_File, "Product of Matrices");
   SRN 343   -->NULL;
   SRE 399   -->EXIT;
118
119          Read_Mat (Input_File, Matrix_In1);
   OVV   3   -->Read_Mat (Output_File, Matrix_In1);
   OAN 282   -->Read_Mat (Input_File, Matrix_In2);
   OAN 283   -->Read_Mat (Input_File, Mat_Sum);
   OAN 284   -->Read_Mat (Input_File, Mat_Prod);
   SRN 344   -->NULL;
   SRE 400   -->EXIT;
   ESR 516   -->Write_Mat (Input_File, Matrix_In1);

120          Read_Mat (Input_File, Matrix_In2);
   OVV   4   -->Read_Mat (Output_File, Matrix_In2);
   OAN 285   -->Read_Mat (Input_File, Matrix_In1);
   OAN 286   -->Read_Mat (Input_File, Mat_Sum);
   OAN 287   -->Read_Mat (Input_File, Mat_Prod);
   SRN 345   -->NULL;
```

```
   SRE 401   -->EXIT;
   ESR 517   -->Write_Mat (Input_File, Matrix_In2);


             Close (Input_File);
   OVV   5   -->Close (Output_File);
   SRN 346   -->NULL;
   SRE 402   -->EXIT;


             -- Add and print two arrays --
             Mat_Sum := Add_Mat (Matrix_In1, Matrix_In2);
   OAN 288   -->Matrix_In1 := Add_Mat (Matrix_In1, Matrix_In2);
   OAN 289   -->Matrix_In2 := Add_Mat (Matrix_In1, Matrix_In2);
   OAN 290   -->Mat_Prod := Add_Mat (Matrix_In1, Matrix_In2);
   OAN 291   -->Mat_Sum := Add_Mat (Matrix_In2, Matrix_In2);
   OAN 292   -->Mat_Sum := Add_Mat (Mat_Sum, Matrix_In2);
   OAN 293   -->Mat_Sum := Add_Mat (Mat_Prod, Matrix_In2);
   OAN 294   -->Mat_Sum := Add_Mat (Matrix_In1, Matrix_In1);
   OAN 295   -->Mat_Sum := Add_Mat (Matrix_In1, Mat_Sum);
   OAN 296   -->Mat_Sum := Add_Mat (Matrix_In1, Mat_Prod);
   SRN 347   -->NULL;
   SRE 403   -->EXIT;
   ESR 518   -->Mat_Sum := Multiply_Mat (Matrix_In1, Matrix_In2);

             Put (Output_File, "Sum of matrices:");
   OVV   6   -->Put (Input_File, "Sum of matrices:");
   OCC 147   -->Put (Output_File, INFILE);
   OCC 148   -->Put (Output_File, OUTILE);
   OCC 149   -->Put (Output_File, "Product of matrices:");
   SRN 348   -->NULL;
   SRE 404   -->EXIT;
             New_line  (Output_File);
   OVV   7   -->New_line  (Input_File);
   SRN 349   -->NULL;
   SRE 405   -->EXIT;
             Write_Mat (Output_File, Mat_Sum);
   OVV   8   -->Write_Mat (Input_File, Mat_Sum);
   OAN 297   -->Write_Mat (Output_File, Matrix_In1);
   OAN 298   -->Write_Mat (Output_File, Matrix_In2);
   OAN 299   -->Write_Mat (Output_File, Mat_Prod);
   SRN 350   -->NULL;
   SRE 406   -->EXIT;
   ESR 519   -->Read_Mat (Output_File, Mat_Sum);

             New_line  (Output_File);
   OVV   9   -->New_line  (Input_File);
   SRN 351   -->NULL;
   SRE 407   -->EXIT;

             -- Multiply and print two arrays--
             Mat_Prod := Multiply_Mat (Matrix_In1, Matrix_In2);
   OAN 300   -->Matrix_In1 := Multiply_Mat (Matrix_In1, Matrix_In2);
   OAN 301   -->Matrix_In2 := Multiply_Mat (Matrix_In1, Matrix_In2);
   OAN 302   -->Mat_Sum := Multiply_Mat (Matrix_In1, Matrix_In2);
   OAN 303   -->Mat_Prod := Multiply_Mat (Matrix_In2, Matrix_In2);
```

```
   OAN 304   -->Mat_Prod := Multiply_Mat (Matrix_Sum, Matrix_In2);
   OAN 305   -->Mat_Prod := Multiply_Mat (Matrix_Prod, Matrix_In2);
   OAN 306   -->Mat_Prod := Multiply_Mat (Matrix_In1, Matrix_In1);
   OAN 307   -->Mat_Prod := Multiply_Mat (Matrix_In1, Matrix_Sum);
   OAN 308   -->Mat_Prod := Multiply_Mat (Matrix_In1, Matrix_Prod);
   SRN 352   -->NULL;
   SRE 408   -->EXIT;
   ESR 520   -->Mat_Prod := Add_Mat (Matrix_In1, Matrix_In2);

             Put (Output_File, "Product of matrices:");
   OCC 150   -->Put (Output_File, OUTFILE);
   OCC 151   -->Put (Output_File, INFILE);
   OCC 152   -->Put (Output_File, "Sum of matrices:");
   SRN 353   -->NULL;
   SRE 409   -->EXIT;
             New_line  (Output_File);
   OVV  10   -->New_line  (Input_File);
   SRN 354   -->NULL;
   SRE 410   -->EXIT;
             Write_Mat (Output_File, Mat_Prod);
   OVV  11   -->Write_Mat (Input_File, Mat_Prod);
   OAN 309   -->Write_Mat (Output_File, Matrix_In1);
   OAN 310   -->Write_Mat (Output_File, Matrix_In2);
   OAN 311   -->Write_Mat (Output_File, Mat_Sum);
   SRN 355   -->NULL;
   SRE 411   -->EXIT;
   ESR 521   -->Read_Mat (Output_File, Mat_Prod);

             New_line  (Output_File);
   OVV  12   -->New_line  (Input_File);
   SRN 356   -->NULL;
   SRE 412   -->EXIT;
             Close (Output_File);
   OVV  13   -->Close (Input_File);
   SRN 357   -->NULL;
   SRE 413   -->EXIT;
          END Matrix;
```