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 ap-
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 $\Omega$ has a single element.
Of course, how well \( \Omega \) 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 [1], the C operators [2], and the Fortran-77 [2] operators used by Mothra [2]. These operators are complete for the Ada language as defined in the Ada Reference Manual [2]. 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVV</td>
<td>Variable replaced by a variable.</td>
</tr>
<tr>
<td>OVC</td>
<td>Variable replaced by a constant.</td>
</tr>
<tr>
<td>OVA</td>
<td>Variable replaced by an array reference.</td>
</tr>
<tr>
<td>OVR</td>
<td>Variable replaced by a record reference.</td>
</tr>
<tr>
<td>OVP</td>
<td>Variable replaced by a pointer reference.</td>
</tr>
<tr>
<td>OVI</td>
<td>Variable initialization elimination.</td>
</tr>
<tr>
<td>OCV</td>
<td>Constant replaced by a variable.</td>
</tr>
<tr>
<td>OCC</td>
<td>Constant replaced by a constant.</td>
</tr>
<tr>
<td>OCA</td>
<td>Constant replaced by an array reference.</td>
</tr>
<tr>
<td>OCR</td>
<td>Constant replaced by a record reference.</td>
</tr>
<tr>
<td>OCP</td>
<td>Constant replaced by a pointer reference.</td>
</tr>
<tr>
<td>OAV</td>
<td>Array reference replaced by a variable.</td>
</tr>
<tr>
<td>OAC</td>
<td>Array reference replaced by a constant.</td>
</tr>
<tr>
<td>OAA</td>
<td>Array reference replaced by an array reference.</td>
</tr>
<tr>
<td>OAR</td>
<td>Array reference replaced by a record reference.</td>
</tr>
<tr>
<td>OAP</td>
<td>Array reference replaced by a pointer reference.</td>
</tr>
<tr>
<td>OAN</td>
<td>Array name replaced by an array name.</td>
</tr>
<tr>
<td>ORV</td>
<td>Record reference replaced by a variable.</td>
</tr>
<tr>
<td>ORC</td>
<td>Record reference replaced by a constant.</td>
</tr>
<tr>
<td>ORA</td>
<td>Record reference replaced by an array reference.</td>
</tr>
<tr>
<td>ORR</td>
<td>Record reference replaced by a record reference.</td>
</tr>
<tr>
<td>ORP</td>
<td>Record reference replaced by a pointer reference.</td>
</tr>
<tr>
<td>ORF</td>
<td>Record field replaced by a record field.</td>
</tr>
<tr>
<td>ORN</td>
<td>Record name replaced by a record name.</td>
</tr>
<tr>
<td>OPV</td>
<td>Pointer reference replaced by a variable.</td>
</tr>
<tr>
<td>OPC</td>
<td>Pointer reference replaced by a constant.</td>
</tr>
<tr>
<td>OPA</td>
<td>Pointer reference replaced by an array reference.</td>
</tr>
<tr>
<td>OPR</td>
<td>Pointer reference replaced by a record reference.</td>
</tr>
<tr>
<td>OPP</td>
<td>Pointer reference replaced by a pointer reference.</td>
</tr>
<tr>
<td>OPN</td>
<td>Pointer name replaced by a pointer name.</td>
</tr>
</tbody>
</table>

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

- Do mutate initializations (only OCV).
- 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.

<table>
<thead>
<tr>
<th>Statement Modification Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEE</td>
</tr>
<tr>
<td>SRN</td>
</tr>
<tr>
<td>SRR</td>
</tr>
<tr>
<td>SGL</td>
</tr>
<tr>
<td>SRE</td>
</tr>
<tr>
<td>SMR</td>
</tr>
<tr>
<td>SRW</td>
</tr>
<tr>
<td>SIZ</td>
</tr>
<tr>
<td>SOI</td>
</tr>
<tr>
<td>SNI</td>
</tr>
<tr>
<td>SRI</td>
</tr>
<tr>
<td>SES</td>
</tr>
<tr>
<td>SCA</td>
</tr>
<tr>
<td>SER</td>
</tr>
</tbody>
</table>

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 (n);
   ```
   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:

   \[
   \text{ORIGINAL} \quad \text{MUTANT} \\
   \text{WHILE (e) LOOP} \quad \text{LOOP} \\
   \text{\quad :} \quad \text{\quad :} \\
   \text{\quad END LOOP;} \quad \text{EXIT WHEN NOT e;} \\
   \text{\quad END LOOP;} \\
   \]

7. **SRW**: Replace repeat-until with WHILE.
   This is the opposite of SWR. The format of the change will be:

   \[
   \text{ORIGINAL} \quad \text{MUTANT} \\
   \text{LOOP} \quad \text{WHILE (NOT e) LOOP} \\
   \text{\quad :} \quad \text{\quad :} \\
   \text{\quad EXIT WHEN e;} \quad \text{END LOOP;} \\
   \text{\quad END LOOP;} \\
   \]

   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 $\text{loop}_i\text{count}$. $\text{loop}_i\text{count}$ 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 $\text{loop}_i\text{count} = 0$, then \text{RAISE Mut\_Trap};
(b) SOI: One iteration
    After the loop, if $\text{loop}_i\text{count} = 1$, then \text{RAISE Mut\_Trap};
(c) SNI: N iterations
    After the loop, if $\text{loop}_i\text{count} > 1$, then \text{RAISE Mut\_Trap};
(d) SRI: reverse iteration
    Add the keyword \text{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.

    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_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_1;
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;
END CASE;
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.

<table>
<thead>
<tr>
<th>Expression Modification Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAI: Absolute value insertion.</td>
</tr>
<tr>
<td>ENI: Neg-Absolute value insertion.</td>
</tr>
<tr>
<td>EEZ: Exception on zero.</td>
</tr>
<tr>
<td>EOR: Arithmetic operator replacement.</td>
</tr>
<tr>
<td>ERR: Relational operator replacement.</td>
</tr>
<tr>
<td>EMR: Membership test replacement.</td>
</tr>
<tr>
<td>ELR: Logical operator replacement.</td>
</tr>
<tr>
<td>EUR: Unary operator insertion.</td>
</tr>
<tr>
<td>EUR: Unary operator replacement.</td>
</tr>
<tr>
<td>ESR: Subprogram operator replacement.</td>
</tr>
<tr>
<td>EDT: Domain twiddle.</td>
</tr>
<tr>
<td>EAR: Attribute replacement.</td>
</tr>
<tr>
<td>EEO: Exception on overflow.</td>
</tr>
<tr>
<td>EEU: Exception on underflow.</td>
</tr>
</tbody>
</table>

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 \(-\text{ABS}\) 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.
   \(<, >, \geq, \leq\) are only defined for scalar and discrete array types.

6. **EMR**: Membership test replacement.
   Replace each \(\text{IN}\) with \(!\text{IN}\) and each \(!\text{IN}\) with \(\text{IN}\).
   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 \(\neg\) in front of each arithmetic expression and subexpression.
   Note: the unary operator \(+\) is the identity operation.

   Replace each unary operator (+, −, \text{ABS}) with each other unary operator.
   Expressions should be fully parenthesized, since \text{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:

```
<table>
<thead>
<tr>
<th>TYPE</th>
<th>MODIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>- +1 and -1</td>
</tr>
<tr>
<td>Float</td>
<td>- *1.05 and *.95</td>
</tr>
<tr>
<td>Fixed Point</td>
<td>- +Delta and -Delta</td>
</tr>
<tr>
<td>Character Types</td>
<td>- T'SUCC and T'PRED</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>- T'SUCC and T'PRED</td>
</tr>
</tbody>
</table>
```

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:
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.

<table>
<thead>
<tr>
<th>Coverage Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDE</td>
</tr>
<tr>
<td>CCO</td>
</tr>
<tr>
<td>CDC</td>
</tr>
<tr>
<td>CMC</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>Ada Description</th>
<th>C</th>
<th>Fortran-77</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVV Variable replaced by a variable.</td>
<td>Vsr</td>
<td>svr</td>
</tr>
<tr>
<td>OVC Variable replaced by a constant.</td>
<td>Vsr</td>
<td>csr</td>
</tr>
<tr>
<td>OVA Variable replaced by an array reference.</td>
<td>Vsr</td>
<td>asr</td>
</tr>
<tr>
<td>OVR Variable replaced by a record reference.</td>
<td>Vsr</td>
<td>~</td>
</tr>
<tr>
<td>OVP Variable replaced by a pointer reference.</td>
<td>Vsr</td>
<td>~</td>
</tr>
<tr>
<td>OVI Variable initialization elimination.</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>OCV Constant replaced by a variable.</td>
<td>Csr</td>
<td>scr</td>
</tr>
<tr>
<td>OCC Constant replaced by a constant.</td>
<td>Ccr</td>
<td>src</td>
</tr>
<tr>
<td>OCA Constant replaced by an array reference.</td>
<td>~</td>
<td>acr</td>
</tr>
<tr>
<td>OCR Constant replaced by a record reference.</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>OCP Constant replaced by a pointer reference.</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>OAV Array reference replaced by a variable.</td>
<td>Varr</td>
<td>sar</td>
</tr>
<tr>
<td>OAC Array reference replaced by a constant.</td>
<td>Varr</td>
<td>car</td>
</tr>
<tr>
<td>OAA Array reference replaced by an array reference.</td>
<td>Varr</td>
<td>aar</td>
</tr>
<tr>
<td>OAR Array reference replaced by a record reference.</td>
<td>Varr</td>
<td>~</td>
</tr>
<tr>
<td>OAP Array reference replaced by a pointer reference.</td>
<td>Varr</td>
<td>~</td>
</tr>
<tr>
<td>OAN Array name replaced by an array name.</td>
<td>Varr</td>
<td>cnr</td>
</tr>
<tr>
<td>ORV Record reference replaced by a variable.</td>
<td>Vtr</td>
<td>~</td>
</tr>
<tr>
<td>ORC Record reference replaced by a constant.</td>
<td>Vtr</td>
<td>~</td>
</tr>
<tr>
<td>ORA Record reference replaced by an array reference.</td>
<td>Vtr</td>
<td>~</td>
</tr>
<tr>
<td>ORR Record reference replaced by a record reference.</td>
<td>Vtr</td>
<td>~</td>
</tr>
<tr>
<td>ORP Record reference replaced by a pointer reference.</td>
<td>Vtr</td>
<td>~</td>
</tr>
<tr>
<td>ORF Record field replaced by a record field.</td>
<td>VSCR</td>
<td>~</td>
</tr>
<tr>
<td>ORN Record name replaced by a record name.</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>OPV Pointer reference replaced by a variable.</td>
<td>Vptr</td>
<td>~</td>
</tr>
<tr>
<td>OPC Pointer reference replaced by a constant.</td>
<td>Vptr</td>
<td>~</td>
</tr>
<tr>
<td>OPA Pointer reference replaced by an array reference.</td>
<td>Vptr</td>
<td>~</td>
</tr>
<tr>
<td>OPR Pointer reference replaced by a record reference.</td>
<td>Vptr</td>
<td>~</td>
</tr>
<tr>
<td>OPP Pointer reference replaced by a pointer reference.</td>
<td>Vptr</td>
<td>~</td>
</tr>
<tr>
<td>OPN Pointer name replaced by a pointer name.</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>
Statement Modification Operators

<table>
<thead>
<tr>
<th>Ada Description</th>
<th>C</th>
<th>Fortran-77</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEE</td>
<td>Exception on execution.</td>
<td>STRP</td>
</tr>
<tr>
<td>SWN</td>
<td>Replace with NULL.</td>
<td>SSDL</td>
</tr>
<tr>
<td>SBR</td>
<td>Return statement replacement.</td>
<td>SRRS</td>
</tr>
<tr>
<td>SGL</td>
<td>GOTO label replacement.</td>
<td>SGLR</td>
</tr>
<tr>
<td>SRE</td>
<td>Replace with EXIT.</td>
<td>SBR</td>
</tr>
<tr>
<td>SWR</td>
<td>Replace WHILE with repeat-until.</td>
<td>SWDD</td>
</tr>
<tr>
<td>SSW</td>
<td>Replace repeat-until with WHILE.</td>
<td>SDWD</td>
</tr>
<tr>
<td>SZI</td>
<td>Zero iteration loop.</td>
<td>~</td>
</tr>
<tr>
<td>SOI</td>
<td>One iteration loop.</td>
<td>~</td>
</tr>
<tr>
<td>SWI</td>
<td>N iteration loop.</td>
<td>SMTT</td>
</tr>
<tr>
<td>SRI</td>
<td>Reverse iteration loop.</td>
<td>~</td>
</tr>
<tr>
<td>SES</td>
<td>END shift.</td>
<td>SMVB</td>
</tr>
<tr>
<td>SCA</td>
<td>CASE alternative replacement.</td>
<td>SSWM</td>
</tr>
<tr>
<td>SER</td>
<td>RAISE exception handler replacement.</td>
<td>~</td>
</tr>
</tbody>
</table>

Expression Modification Operators

<table>
<thead>
<tr>
<th>Ada Description</th>
<th>C</th>
<th>Fortran-77</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAI</td>
<td>Absolute value insertion.</td>
<td>VDTR</td>
</tr>
<tr>
<td>ENI</td>
<td>Neg-absolute value insertion.</td>
<td>VDTR</td>
</tr>
<tr>
<td>EEZ</td>
<td>Exception on zero.</td>
<td>VDTR</td>
</tr>
<tr>
<td>EOR</td>
<td>Arithmetic operator replacement.</td>
<td>ORAN</td>
</tr>
<tr>
<td>ERR</td>
<td>Relational operator replacement.</td>
<td>ORRN</td>
</tr>
<tr>
<td>EMR</td>
<td>Membership test replacement.</td>
<td>~</td>
</tr>
<tr>
<td>ELR</td>
<td>Logical operator replacement.</td>
<td>OBBN</td>
</tr>
<tr>
<td>EUI</td>
<td>Unary operator insertion.</td>
<td>Uuor</td>
</tr>
<tr>
<td>EUR</td>
<td>Unary operator replacement.</td>
<td>Uuor</td>
</tr>
<tr>
<td>ESR</td>
<td>Subprogram operator replacement.</td>
<td>~</td>
</tr>
<tr>
<td>EDT</td>
<td>Domain twiddle.</td>
<td>VTWD</td>
</tr>
<tr>
<td>EAR</td>
<td>Attribute replacement.</td>
<td>~</td>
</tr>
<tr>
<td>EEO</td>
<td>Exception on overflow.</td>
<td>~</td>
</tr>
<tr>
<td>EEU</td>
<td>Exception on underflow.</td>
<td>~</td>
</tr>
</tbody>
</table>

Coverage Operators

<table>
<thead>
<tr>
<th>Ada Description</th>
<th>C</th>
<th>Fortran-77</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDE</td>
<td>Decision coverage.</td>
<td>Oior</td>
</tr>
<tr>
<td>CCO</td>
<td>Condition coverage.</td>
<td>~</td>
</tr>
<tr>
<td>CDC</td>
<td>Decision/condition coverage.</td>
<td>~</td>
</tr>
<tr>
<td>CMC</td>
<td>Multiple condition coverage.</td>
<td>~</td>
</tr>
</tbody>
</table>

Tasking Operators

<table>
<thead>
<tr>
<th>Ada Description</th>
<th>C</th>
<th>Fortran-77</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEM</td>
<td>ENTRY statement modification.</td>
<td>~</td>
</tr>
<tr>
<td>TAR</td>
<td>ACCEPT statement replacement.</td>
<td>~</td>
</tr>
<tr>
<td>TSA</td>
<td>SELECT alternative replacement.</td>
<td>~</td>
</tr>
</tbody>
</table>

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 [1]. 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.

<table>
<thead>
<tr>
<th>Expression Modification Operators</th>
<th>Coverage Operators</th>
<th>Tasking Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA1 Absolute value insertion.</td>
<td>CMC Multiple Condition coverage.</td>
<td>TEM ENTRY statement modification.</td>
</tr>
<tr>
<td>ENI Neg-absolute value insertion.</td>
<td></td>
<td>TAR ACCEPT statement replacement.</td>
</tr>
<tr>
<td>EEZ Exception on zero.</td>
<td></td>
<td>TSA SELECT alternative replacement.</td>
</tr>
<tr>
<td>EOR Arithmetic operator replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERR Relational operator replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMR Membership test replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELR Logical operator replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUI Unary operator insertion.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUR Unary operator replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESR Subprogram operator replacement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE0 Exception on overflow.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEU Exception on underflow.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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 599 mutants.

```
WITH Text_IO; USE Text_IO;

PROCEDURE Matrix IS
  -- Package instantiations.
  PACKAGE Int_IO IS NEW Integer_IO (Integer);
  USE Int_IO;

  -- Constant declarations.
  INFILE : CONSTANT String := "pi.in";
  OUTFILE : CONSTANT String := "pi.out";
  MATSIZE : CONSTANT Integer := 3;

  -- Type declarations.
  TYPE Matrix_Type IS ARRAY (1..MATSIZE,1..MATSIZE) OF Integer;

  -- Variable declarations.
  Input_File, Output_File : File_Type;
  Matrix_In1, Matrix_In2 : Matrix_Type;
  Mat_Sum, Mat_Prod : Matrix_Type;

  -- Procedure : Read_Mat (Input_File : IN File_Type; --
  --  Mat : OUT Matrix_Type) --
  -- Purpose : Read a matrix from a file. --
  -- Params : Input_File : The file to read from. --
  --  : Mat : The matrix to read into. --
  -- Pre : Input_File must be open --
  --  : The matrix is MATSIZE lines --
  --  : MATSIZE numbers per row --

PROCEDURE Read_Mat (Input_File : IN File_Type; Mat : OUT Matrix_Type) IS
  SZZI  => szi.lci1 : Natural;
  SZZI  => szi.lci2 : Natural;
```
\[ \text{SZI} \] 
\[ \text{FOR } i \text{ IN 1..MATSIZE LOOP} \]
\[ \text{FOR } j \text{ IN 1..MATSIZE LOOP} \]
\[ \text{IF } \text{szi\_lci1} = 0 \text{ RAISE Mut\_Trap} ; \]
\[ \text{SNI 423} \] 
\[ \text{IF } \text{szi\_lci1} = 1 \text{ RAISE Mut\_Trap} ; \]
\[ \text{SNI 432} \] 
\[ \text{IF } \text{szi\_lci1} > 1 \text{ RAISE Mut\_Trap} ; \]
\[ \text{END LOOP} ; \]
\[ \text{SZI} \] 
\[ \text{IF } \text{szi\_lci2} = 0 \text{ RAISE Mut\_Trap} ; \]
\[ \text{SNI 424} \] 
\[ \text{IF } \text{szi\_lci2} = 1 \text{ RAISE Mut\_Trap} ; \]
\[ \text{SNI 433} \] 
\[ \text{IF } \text{szi\_lci2} > 1 \text{ RAISE Mut\_Trap} ; \]
\[ \text{END Read\_Mat} ; \]
PROCEDURE Write_Mat (Output_File : IN File_Type; Mat : IN Matrix_Type) IS
  Szi #=>szi_lci3 : Natural;
  Szi #=>szi_lci4 : Natural;
BEGIN
  Szi #=>szi_lci3 := 0; -- Initialize loop counter.
  FOR i IN 1..Matsize LOOP
    OCC 54 --FOR i IN MATSIZ..MATSIZL LOOP
    OCC 55 --FOR i IN 1..1 LOOP
    SEE 315 --Except_onExec (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..MATSIZL LOOP
    EDT 522 --FOR i IN 0..MATSIZL LOOP
    EDT 523 --FOR i IN 2..MATSIZL LOOP
    EDT 524 --FOR i IN 0..MATSIZL-1 LOOP
    EDT 525 --FOR i IN 0..MATSIZL+1 LOOP
    Szi #=>szi_lci4 := 0; -- Initialize loop counter.
  FOR j IN 1..Matsize LOOP
    OCC 56 --FOR j IN MATSIZ..MATSIZL LOOP
    OCC 57 --FOR j IN 1..1 LOOP
    OCC 58 --FOR j IN 1..1 LOOP
    OCC 59 --FOR j IN 1..i LOOP
    SEE 316 --Except_onExec (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..MATSIZL LOOP
    SES 450 --END LOOP; -- Line 63 moved below 61.
    EDT 526 --FOR j IN 0..MATSIZL LOOP
    EDT 527 --FOR j IN 2..MATSIZL LOOP
    EDT 528 --FOR j IN 1..MATSIZL-1 LOOP
    EDT 529 --FOR j IN 1..MATSIZL+1 LOOP
  OCV 27 --Put (Input_File, Mat (i,j));
  OCC 60 --Put (Output_File, Mat (i,j));
  OCC 61 --Put (Output_File, Mat (j,j));
  OCC 62 --Put (Output_File, Mat (MATSIZ,j));
  OCC 63 --Put (Output_File, Mat (i,1));
  OCC 64 --Put (Output_File, Mat (i,i));
  OCC 65 --Put (Output_File, Mat (i,MATSIZ));
  OCA 160 --Put (Output_File, Mat (Matsiz(i,j),j));
  OCA 161 --Put (Output_File, Mat (i,Matsiz(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));
FUNCTION Add_Mat (Mat1, Mat2 : IN Matrix_Type) RETURN Matrix_Type IS
  rslt_mat : Matrix_Type;
  SIZI
  =>szi_lci5 : Natural;
  SIZI
  =>szi_lci6 : Natural;
BEGIN

SIZI    =>szi_lci5 := 0; -- Initialize loop counter.
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

END LOOP;
SZI 417  --IF (szi_lci4 = 0) RAISE Mut_Traps;
SII 426  --IF (szi_lci4 = 1) RAISE Mut_Traps;
SNI 435  --IF (szi_lci4 > 1) RAISE Mut_Traps;
65  END LOOP;
SIZI 417  --IF (szi_lci4 = 0) RAISE Mut_Traps;
SII 426  --IF (szi_lci4 = 1) RAISE Mut_Traps;
SNI 435  --IF (szi_lci4 > 1) RAISE Mut_Traps;
66  END Write_Mat;
67
for i in 1..MATSIZE+1 loop
  szi := szi_{lc6} := 0;  -- Initialize loop counter.
end loop;
for j in 1..MATSIZE loop
  --for j in 1..1 loop
  srr
  rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,j);
end loop;
for j in 1..MATSIZE loop
  --for j in 1..1 loop
  srr
  rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,j);
end loop;
for j in 1..MATSIZE loop
  --for j in 1..1 loop
  sri
end loop;
for j in reverse 1..MATSIZE loop
  ses
end loop;
for j in 1..MATSIZE+1 loop
  rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,j);
end loop;
DCA 179  -->rslt_mat (i,j) := Mat1 (i,j) + Mat2 (i,Mat2(i,j));
DCA 204  -->rslt_mat (i,j) := i + Mat2 (i,j);
DCA 205  -->rslt_mat (i,j) := j + Mat2 (i,j);
DCA 206  -->rslt_mat (i,j) := MATSIZE + Mat2 (i,j);
DCA 207  -->rslt_mat (i,j) := 1 + Mat2 (i,j);
DCA 208  -->rslt_mat (i,j) := Mat1 (i,j) + i;
DCA 209  -->rslt_mat (i,j) := Mat1 (i,j) + j;
DCA 210  -->rslt_mat (i,j) := Mat1 (i,j) + MATSIZE;
DCA 211  -->rslt_mat (i,j) := Mat1 (i,j) + 1;
DAA 222  -->rslt_mat (i,j) := rslt_mat (i,j) + Mat2 (i,j);
DAA 223  -->rslt_mat (i,j) := Mat2 (i,j) + Mat2 (i,j);
DAA 224  -->rslt_mat (i,j) := Mat1 (i,j) + rslt_mat (i,j);
DAA 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  -->Exception_On_Exec (SEE+9);
SRN 336  -->NULL;
SRR 369  -->RETURN;
SRR 370  -->RETURN (rslt_mat);
SRE 392  -->EXIT;
EA1 459  -->rslt_mat (i,j) := ABS(Mat1 (i,j) + Mat2 (i,j));
EA1 460  -->rslt_mat (i,j) := ABS(Mat1 (i,j)) + Mat2 (i,j);
EA1 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);
EEU 556  -->rslt_mat (i,j) := EEU(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;
SZI 427  -->IF (szi.lci5 = 1) RAISE Mut_Trap;
SN1 436  -->IF (szi.lci5 > 1) RAISE Mut_Trap;
82  END LOOP;
SZI 419  -->IF (szi.lci6 = 0) RAISE Mut_Trap;
SN1 428  -->IF (szi.lci6 = 1) RAISE Mut_Trap;
SN1 437  -->IF (szi.lci6 > 1) RAISE Mut_Trap;
83  RETURN (rslt_mat);
Dan 254  -->RETURN (Matrix_In1);
Dan 255  -->RETURN (Matrix_In2);
Dan 256  -->RETURN (Mat_Sum);
Dan 257  -->RETURN (Mat_Prod);
Dan 258  -->RETURN (Mat1);
Dan 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.
SE1 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;
96  SZ1 497  ==>sz1.lci7 : Natural;
97  SZ1 504  ==>sz1.lci8 : Natural;
98  SZ1 505  ==>sz1.lci9 : Natural;
99  BEGIN
100   SZI 509  ==>sz1.lci7 := 0;  -- Initialize loop counter.
101  FOR i IN 1..MATSIZELOOP
102  Ocv 29  -->FOR i IN tmp_sum..MATSIZELOOP
103  Ocv 30  -->FOR i IN 1..tmp_sum LOOP
104  Ocv 90  -->FOR i IN MATSIZE..MATSIZELOOP
105  Ocv 91  -->FOR i IN 0..MATSIZELOOP
106  Ocv 92  -->FOR i IN 1..1 LOOP
107  Ocv 93  -->FOR i IN 1..0 LOOP
108  See 322  -->Except_On_Exec (SEE+10);
109  Snr 337  -->NULL; -- Replaces statements 97-105
110  Snr 372  -->RETURN; -- Replaces statements 97-105
111  Snr 373  -->RETURN (rslt_mat); -- Replaces statements 97-105
112  Sre 393  -->EXIT; -- Replaces statements 97-105
113  Srr 447  -->FOR i IN REVERSE 1..MATSIZELOOP
114  -->EDT  FOR i IN 0..MATSIZELOOP -- equivalent to OCC 91
115  EdT 542  -->FOR i IN 2..MATSIZELOOP
116  EdT 543  -->FOR i IN 1..MATSIZEL-1 LOOP
-- 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
SRRI 449 -- 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
101

\[\text{tmp\_sum := } \text{tmp\_sum + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 17} \quad \text{->Mat1(i,k) := tmp\_sum + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 18} \quad \text{->Mat2(k,j) := tmp\_sum + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 19} \quad \text{->rslt\_mat(i,j) := tmp\_sum + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 20} \quad \text{->tmp\_sum := Mat1(i,k) + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 21} \quad \text{->tmp\_sum := Mat2(k,j) + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVA 22} \quad \text{->tmp\_sum := rslt\_mat(i,j) + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OCV 36} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (tmp\_sum,k) * Mat2 (k,j)};\]

\[\text{OCV 37} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,tmp\_sum) * Mat2 (k,j)};\]

\[\text{OCV 38} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (tmp\_sum,j)};\]

\[\text{OCV 39} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,tmp\_sum)};\]

\[\text{OCC 113} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (j,k) * Mat2 (k,j)};\]

\[\text{OCC 114} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OCC 115} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (MATSIZE,k) * Mat2 (k,j)};\]

\[\text{OCC 116} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (0,k) * Mat2 (k,j)};\]

\[\text{OCC 117} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (k,k) * Mat2 (k,j)};\]

\[\text{OCC 118} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,j) * Mat2 (k,j)};\]

\[\text{OCC 119} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,j) * Mat2 (k,j)};\]

\[\text{OCC 120} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,1) * Mat2 (k,j)};\]

\[\text{OCC 121} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,MATSIZE) * Mat2 (k,j)};\]

\[\text{OCC 122} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,0) * Mat2 (k,j)};\]

\[\text{OCC 123} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (i,j)};\]

\[\text{OCC 124} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (j,j)};\]

\[\text{OCC 125} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (i,j)};\]

\[\text{OCC 126} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (MATSIZExj)};\]

\[\text{OCC 127} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (O,j)};\]

\[\text{OCC 128} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,i)};\]

\[\text{OCC 129} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,1)};\]

\[\text{OCC 130} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,MATSIZE)};\]

\[\text{OCC 131} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,0)};\]

\[\text{OCC 132} \quad \text{->tmp\_sum := i + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVC 153} \quad \text{->tmp\_sum := j + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVC 154} \quad \text{->tmp\_sum := 1 + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVC 155} \quad \text{->tmp\_sum := MATSIZE + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OVC 156} \quad \text{->tmp\_sum := 0 + Mat1 (i,k) * Mat2 (k,j)};\]

\[\text{OCA 180} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (Mat1(i,k),k) * Mat2 (k,j)};\]

\[\text{OCA 181} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,Mat1(i,k)) * Mat2 (k,j)};\]

\[\text{OCA 182} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (Mat2(k,j),k) * Mat2 (k,j)};\]

\[\text{OCA 183} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,Mat2(k,j)) * Mat2 (k,j)};\]

\[\text{OCA 184} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (rslt\_mat(i,j),k) * Mat2 (k,j)};\]

\[\text{OCA 185} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,rslt\_mat(i,j)) * Mat2 (k,j)};\]

\[\text{OCA 186} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (Mat1(i,k),j)};\]

\[\text{OCA 187} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,Mat1(i,k))};\]

\[\text{OCA 188} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (Mat2(k,j),j)};\]

\[\text{OCA 189} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,Mat2(k,j))};\]

\[\text{OCA 190} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (rslt\_mat(i,j),j));\]

\[\text{OCA 191} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * Mat2 (k,rslt\_mat(i,j))};\]

\[\text{OAV 198} \quad \text{->tmp\_sum := tmp\_sum + tmp\_sum * Mat2 (k,j)};\]

\[\text{OAV 199} \quad \text{->tmp\_sum := tmp\_sum + Mat1 (i,k) * tmp\_sum};\]
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;
SRN 380 -->RETURN;
SRN 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 := EE0(tmp_sum + Mat1 (i,k) * Mat2 (k,j));
EEU 559        -->tmp_sum := EEU(tmp_sum + Mat1 (i,k) * Mat2 (k,j));

END LOOP;

SZI 420        -->IF (szi_lci7 = 0) RAISE MutTrap;
S01 429        -->IF (szi_lci7 = 1) RAISE MutTrap;
SNI 438        -->IF (szi_lci7 > 1) RAISE MutTrap;
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);
UCV 40        -->rslt_mat (tmp_sum,j) := tmp_sum;
UCV 41        -->rslt_mat (i,tmp_sum) := tmp_sum;
OCC 133        -->rslt_mat (j,j) := tmp_sum;
OCC 134        -->rslt_mat (i,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,i) := 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.
EA1 467        -->rslt_mat (i,j) := ABS(tmp_sum);
EN1 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;

END LOOP;

SZI 421        -->IF (szi_lci8 = 0) RAISE MutTrap;
S01 430        -->IF (szi_lci8 = 1) RAISE MutTrap;
SNI 439        -->IF (szi_lci8 > 1) RAISE MutTrap;

END LOOP;
SZI 422  -->IF (szi_lci9 = 0) RAISE Mut_Trap;
SNI 431  -->IF (szi_lci9 = 1) RAISE Mut_Trap;
SNK 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.
SRN 382  -->RETURN;
--SRE -- no SRE mutant, function must have a RETURN.
SES 457  -->END LOOP; -- Line 105 moved below 106.

END Multiply_Mat;

108

109

110  -----------------------------------------------
111  -- Main body of matrix
112  -- Open files, read, and multiply the two matrices.
113  -- Close the files.
114  -----------------------------------------------
115  BEGIN -- Matrix
116  Open (Input_File, In_File, INFILE);
117  OAN 1  -->Open (Output_File, In_File, INFILE);
118  OAN 2  -->Open (Input_File, In_File, OUTFILE);
119  OAN 3  -->Read (Input_File, Matrix_In1);
120  OAN 4  -->Read (Input_File, Matrix_In2);
121  OAN 5  -->Read (Input_File, Mat_Sum);
122  OAN 6  -->Read (Input_File, Mat_Prod);
123  SNK 344  -->NULL;
124  SRN 345  -->EXIT;
125  SRE 396  -->EXIT;
126  Create (Output_File, Out_File, OUTFILE);
127  OAC 141  -->Create (Input_File, Out_File, INFILE);
128  OAC 142  -->Create (Input_File, Out_File, OUTFILE);
129  OAC 143  -->Create (Input_File, Out_File, "Sum of Matrices");
130  OAC 144  -->Create (Input_File, Out_File, "Product of Matrices");
131  OAC 145  -->Except_On_Exec (SEE+14);
132  SRN 342  -->NULL;
133  SRR 382  -->EXIT;
134  SRE 399  -->EXIT;
135  SES 457  -- End of LOOP

136  Read_Mat (Input_File, Matrix_In1);
137  OAN 282  -->Read_Mat (Input_File, Matrix_In2);
138  OAN 283  -->Read_Mat (Input_File, Mat_Sum);
139  OAN 284  -->Read_Mat (Input_File, Mat_Prod);
140  SNK 344  -->NULL;
141  SRN 345  -->EXIT;
142  SRE 400  -->EXIT;
143  ESR 516  -->Write_Mat (Input_File, Matrix_In1);
144  OAC 285  -->Write_Mat (Input_File, Matrix_In2);
145  OAC 286  -->Write_Mat (Input_File, Mat_Sum);
146  OAC 287  -->Write_Mat (Input_File, Mat_Prod);
147  SNK 344  -->NULL;
SRE 401  -->EXIT;
ESR 517  -->Write_Mat (Input_File, Matrix_In2);

121
122  Close (Input_File);
  OVW 5  -->Close (Output_File);
  SRN 346  -->NULL;
  SRE 402  -->EXIT;
123
124
125  -- Add and print two arrays --
126  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);

127  Put (Output_File, "Sum of matrices:"");
  OVW 6  -->Put (Input_File, "Sum of matrices:"");
  OCC 147  -->Put (Output_File, INFILE);
  OCC 148  -->Put (Output_File, OUTFILE);
  OCC 149  -->Put (Output_File, "Product of matrices:"");
  SRN 348  -->NULL;
  SRE 404  -->EXIT;
128  New_line (Output_File);
  OVW 7  -->New_line (Input_File);
  SRN 349  -->NULL;
  SRE 405  -->EXIT;
129  Write_Mat (Output_File, Mat_Sum);
  OVW 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);

130  New_line (Output_File);
  OVW 9  -->New_line (Input_File);
  SRN 351  -->NULL;
  SRE 407  -->EXIT;
131
132  -- Multiply and print two arrays--
133  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);

134  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;
135  New_line (Output_File);
OVV 10  -->New_line (Input_File);
SRN 354  -->NULL;
SRE 410  -->EXIT;
136  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, Matrix_Sum);
SRN 355  -->NULL;
SRE 411  -->EXIT;
ESR 521  -->Read_Mat (Output_File, Mat_Prod);
137  New_line (Output_File);
OVV 12  -->New_line (Input_File);
SRN 356  -->NULL;
SRE 412  -->EXIT;
138  
139  Close (Output_File);
OVV 13  -->Close (Input_File);
SRN 357  -->NULL;
SRE 413  -->EXIT;
140  END Matrix;