EXCEPTION FLOW ANALYSIS IN ADA

by
Deborah A. Baker
and
Stanley M. Sutton, Jr.

CU-CS-319-86 February, 1986

University of Colorado, Department of Computer Science Boulder, Colorado.

Abstract

This paper describes a technique for the analysis of the propagation and handling of exceptions in Ada[1] programs. Exception facilities in a programming language can serve to enhance the reliability of programs written in the language. However, the Ada exception handling facility is also capable of adding greatly to the complexity of large Ada programs. Hence the need arises for tools and methods to aid in understanding Ada programs that utilize the exception handling facility.

Exception flow analysis can assist in determining, for instance, which exception handlers may handle the raising of an exception. The need for such analysis arises because the handler for an exception is determined dynamically in Ada. Exception flow analysis can also solve other problems related to the propagation of exceptions, such as which exceptions would cause the program to crash and which exception handlers will never be used. The algorithm that is used in exception flow analysis is based upon an algorithm for analysis of intraprocedural data flow. An exception flow analysis tool is under development.

Index Terms

Ada, data flow, exception flow, exception handling, reliability, software tools, static analysis, testing and verification

---

[1] Ada is a registered trademark of the US Government, ADA Joint Program Office.

1. Introduction

This section presents the motivation behind static analysis, generally describes the Ada exception facility, and presents the problem of exception flow analysis.

1.1. Static Analysis

The reliability of software systems continues to be a concern of software engineering. Anecdotes concerning failures of software systems are legion [10]. Furthermore, software systems are quite complex. A number of software validation techniques have been developed that factor the complexity as well as find errors in software. Use of these techniques can significantly raise one's confidence that a software system is correct (with respect to its specifications). Use of validation techniques is often most effective if supported by automated tools that bear the burden of bookkeeping and otherwise attend to tedious detail.

Static analysis is one type of software validation. Particular tools that perform static analysis do not execute the software system that is the object of analysis. Rather, some model of the program is studied. Static analysis techniques have been shown to be useful because they can demonstrate the absence of classes of errors. A very simple kind of static analysis is syntax analysis: a program is analyzed and either syntax errors are found, or it is demonstrated that no syntax errors exist.

Another form of static analysis is data flow analysis [11]. First studied in the context of code optimization, data flow analyzers can detect the presence, or show the absence, of such data flow anomalies as uninitialized variables and dead definitions of a variable.

1.2. The Ada Exception Facility

The exception facility in the Ada programming language [1] is a mechanism intended to provide some programming support for dealing with various erroneous, unusual or merely noteworthy situations, called *exceptions*, which might occur during the execution of an Ada program. Some exceptions are predefined in Ada; others may be defined by users. An exception is *raised* when it occurs. The raising of an exception can happen

1. explicitly, via a **raise** statement,
2. as the result of an action that breaks some rule of the language, such as the predefined exception NUMERIC_ERROR being raised in response to an attempt to divide by zero, or
3. as the result of being propagated from some other place in the program.

When an exception is raised during the execution of statements or the elaboration of declarations, the execution or elaboration is abandoned. Certain components of Ada programs, called *frames*, may have *exception handlers*. A frame is either a block statement or the body of a subprogram, package, task, or generic unit. An exception handler provides for some response to the exception to be made or some corrective action to be taken. If an exception, x, is raised during the execution of statements in a frame, control is transferred to the handler for x in that frame, if there is one. If an exception x is raised during the execution of statements in a frame with no handler for x, during the elaboration of declarations, or during the execution of the statements in an exception handler, then the exception is *propagated*. A propagated exception is raised again at the point to which it is propagated. This point, and hence the exception handler that may be invoked in response to the exception, is determined dynamically according to rules specified in the language definition [1]. When an exception is raised

in no case does control return to the point where it was raised. The propagation of an exception from a task body causes the task to become complete. The propagation of an exception from a library package, or a subprogram that is the main program, causes execution of the main program to be abandoned.

There are many important and interesting issues that can be raised concerning the design of exception facilities. Among these issues are:

1. static versus dynamic association of handlers for an exception with operations that might raise the exception (this happens dynamically in Ada),

2. allowing the operation that raised an exception to resume, be retried or to terminate (resumption is not allowed in Ada), and

3. the benefits of default handlers and parameters for handlers (Ada supports neither default handlers nor parameters to exception handlers).

These issues have been addressed elsewhere [5, 7, 13]; a critique of the design of Ada is not our purpose here. Our goal is to design and develop an Ada exception flow analysis tool using existing data flow analysis techniques.

1.3. Exception Flow Analysis

Exception facilities in a programming language can serve to enhance the reliability of programs written in that language [4]. However, extensive use of exception-related facilities can greatly complicate large Ada programs. New tools and analysis techniques are required enhance our ability to understand Ada programs that utilize exceptions and exception handlers.

The propagation of exceptions is similar to the propagation of data values (this is discussed in more detail in sections 2.1 and 2.2). It is because of this apparent similarity that exception flow analysis was approached as a data flow problem. Data

flow analysis is well understood and accepted. By drawing a parallel between exception flow analysis and data flow analysis, the results of research concerning data flow can be applied to exception propagation (for example, complexity results concerning algorithms do not need to be rederived). A general framework for static analysis is developed in [3].

Other work on the Ada exception facility has centered on specifications and axiomatics. Programs annotated using Anna [9] can include specifications about exception propagation. A propagation specification can take one of two forms: a condition that will hold if a particular exception is propagated or a condition under which an exception must be propagated. Axioms have been developed for the semantics of **raise** statements as well as for the semantics of blocks that contain exception handlers [8]. These axioms can be used in conjunction with assertions (such as those expressible in Anna) to show, for example, that a particular exception will never occur.

Exception flow analysis can provide answers to many questions concerning the handling, raising and propagation of exceptions in Ada programs. The primary information derived from exception flow analysis is where each exception is handled and where it can be propagated from each place at which it can be raised. Other information that can be derived from exception flow analysis of a program includes

1. the exceptions that may be raised in, or propagated into, a given part of the program,
2. the source of exceptions propagated into a given part of the program,
3. the user-defined exceptions that are never raised,
4. the exceptions that are handled in a given frame,

5. the exception handlers that are unnecessary because the handled exceptions cannot be raised in, or propagated into, the frame of the handler,

6. the exceptions that are propagated from a given part of the program, either because they are not handled, or are handled but then reraised,

7. the exceptions that may cause a task to become complete, and

8. the exceptions that may cause the execution of the program to be abandoned.

While complete or partial answers to some of these questions can be found by a simple textual scan of the program, the more difficult questions can only be answered by a more sophisticated analysis such as flow analysis.

In light of the questions above, it can be seen that the objectives of data flow analysis and exception flow analysis differ in part. Data flow analysis is directed toward the discovery of anomalous conditions such as the use of a variable before it is defined or the occurrence of two consecutive definitions for a variable without an intervening use. Such conditions clearly represent actual or potential problems. Some of the conditions that can be identified through exception flow analysis are also clearly anomalous. These include user-defined exceptions that are never raised or handlers for exceptions that can never occur in the frame of the handler. Other exception-related conditions are less clearly anomalous. Many frames will not have handlers for all of the exceptions that may occur in them, but such an absence is often not a problem. An exception may be propagated into a number of contexts from a given context, but only certain propagation paths may critically compromise the execution of the program. The overall goal of exception flow analysis is to provide sufficient information to programmers to enable them to identify conditions that may be important to them and to have

confidence that they have not overlooked situations with which they ought to be concerned.

1.4. This Paper

The main part of the remainder of this paper is organized as follows: Section 2 discusses exception flow analysis as a form of static analysis, Section 3 presents the exception flow analysis algorithm, and Section 4 discusses issues in the implementation of an exception flow analysis tool based on the algorithm.

2. Exception Flow Analysis as a Form of Static Analysis

Exception flow analysis requires a graphical representation of Ada programs, but this representation must be different from that used for data flow analysis. Once an appropriate graphical representation for exception flow analysis is established, it is possible to consider data flow problems that may be used as models for the exception flow problems. The *live variable* data flow problem is the most appropriate model for exception flow; its exception flow analog is the *live exception* problem.

2.1. The Context Graph Model of Ada Programs

Intraprocedural data flow analysis is performed over a flow graph of the procedure [6]. Each node in the graph represents one or more statements that are always executed as a group. A node has a single entry point at the first statement, and once a node is entered all of the statements are executed in succession.

A flow graph is inappropriate for exception flow analysis because the raising of an exception causes suspension of the normal flow of control. Exceptions are not propagated through successive statements in the way that data values may be

considered to be. Following the raising of an exception, control may pass to a very different part of a program than that in which the exception was raised. Furthermore, exceptions may be raised during the elaboration of declarations. Consequently it is necessary for exception flow analysis to consider parts of programs that are irrelevant for data flow analysis (e.g. declarative parts) and it is expedient for exception flow analysis to ignore parts of programs that are relevant for data flow analysis (e.g. flow of control within a sequence of statements). For these reasons it is also clear that exception flow analysis is not an intraprocedural problem but is rather a global problem to be solved over a whole program.

Intuitively, a graphical representation of an Ada program that will be useful for exception flow analysis will model those aspects of the program that are interesting with respect to the propagation of exceptions. Such a graph can be constructed by letting the nodes represent those places in a program in which exceptions may be raised and between which they may be propagated and by letting the edges represent possible paths of propagation. The Ada Reference Manual [1] defines rules for the propagation of exceptions from and to package declarations, package bodies, task declarations, task bodies, subprogram bodies, block statements, and **accept** statements. We call these portions of Ada programs *exception contexts*, or simply *contexts*. These contexts comprise the types of nodes in the graphs used for exception flow analysis, which are called *context graphs*. The rules that govern the paths of propagation of exceptions between contexts depend upon the calling structure of the program and lexical containment of contexts. These rules determine the edges in the context graph.

A context graph could be built for any context. We will restrict our attention,

however, to context graphs built for main programs (where main programs are subprograms that are library units).

The formal definitions for a context graph are as follows:

A *context* is either a package declaration, package body, task declaration, task body, subprogram body, block statement, or **accept** statement.

A context c_1 *contains* a context c_2 if

1. c_1 is a package declaration and c_2 is a package declaration or a task declaration,

2. c_1 is a subprogram body, package body, task body or block statement and c_2 is a package declaration, task declaration, package body, task body or a block statement,

3. c_1 is an **accept** statement and c_2 is a block statement or an **accept** statement, or

4. c_1 is a task body or a block statement and c_2 is an **accept** statement,

and if c_2 occurs at the outermost lexical level in c_1.

A context c_1 *calls* a context c_2 if

1. c_1 is a subprogram body, package body, task body, block statement or an **accept** statement, and c_2 is a subprogram body or an **accept** statement, or

2. c_1 is a package declaration or a task declaration and c_2 is a subprogram body

and if c_1 contains a subprogram call or task entry call of the subprogram or **accept** statement represented by c_2 at the outermost lexical level within its sequence of statements.

A *context graph* is a triple (S, N, E). S is the start node, representing the environment. N is a set of nodes representing contexts. A node n is a member of N if and only if one of the following conditions holds:

- n represents the main program,
- n represents a library package or subprogram that is included in a **with** clause of the main program or in a **with** clause of some other context represented in N, or
- n represents a context that is called or contained by some context in N.

E is a set of directed edges (n_1, n_2) where

- n_1 is S and n_2 represents the main program or n_2 represents a subprogram or package that appears in a **with** clause (as above), or
- n_1 and n_2 are members of N and
  - the context represented by n_1 contains the context represented by n_2 or
  - the context represented by n_1 calls the context represented by n_2.

Generic units are not included in the set of contexts because generic instantiations can be resolved into one of the other types of contexts; generic units are not discussed further. Subprogram declarations are also not included in the set of contexts, even though subprogram declarations may occur as library units in Ada programs. This is because no exceptions may be raised by the elaboration of a subprogram declaration and there are no rules in the language for propagation of exceptions into or from them.

Some types of contexts are of particular interest. Package bodies, subprogram bodies, task bodies, and block statements may include exception handlers (these contexts, along with generic unit bodies, are called *frames* [1]). Additionally, package declarations and

subprogram bodies may be library units; when an exception is propagated from a library package or from the main subprogram the execution of the program is abandoned.

The contexts that are also frames have a significant internal structure consisting of a declarative part, a sequence of statements, and exception handlers (any of which may be empty). Exceptions may be raised in or propagated into each of these parts. However, only exceptions occurring in the sequence of statements of such a context may be handled by an exception handler in the context. This complicates exception flow analysis, and it may suggest that each of these parts should be treated as a separate context. However, the point at which an exception occurs within a frame does not affect the contexts to which the exception may be propagated; all exceptions raised within a frame may be propagated to the same set of receiving contexts. For this reason frames are considered as individual contexts and are not subdivided.

2.2. Overview of Data Flow Problems

We have posed some questions about exception flow which we wish to treat as questions of data flow. There are four basic types of data flow problem that can be considered for exception flow analysis [6]. These are available expressions, reaching definitions, very busy expressions, and live variables.

These four basic problems can be classified in various ways [3, 6]. One classification is by the direction of propagation of information through the flow graph during analysis. Available expressions and reaching definitions are *top-down* problems in which the analysis progresses down the flow graph and the information for a given node is

determined by operations over its predecessors in the graph. Very busy expressions and live variables are *bottom-up* problems in which analysis progresses up the flow graph and the information for a given node is determined by operations over its successors. Another classification is by the way in which information is combined as it is propagated from node to node. Available expressions and very busy expressions are *set intersection* problems in which the set of values propagated into a node is determined as the intersection of the sets of values propagated from its predecessors or successors, respectively. Reaching definitions and live variables are *set union* problems in which the set of values propagated into a node is determined as the union of the sets of values propagated from its predecessors or successors, respectively. Additionally, the data flow problems may be distinguished by the types of operations on which the solutions depend. The two basic operations on variables are *definitions* and *uses*. A definition is any operation on a variable that may modify its value. A use is any reference of a variable that cannot modify its value. The reaching definitions and available expressions problems are determined fundamentally by definitions, independent of uses, whereas the very busy expressions and live variables problems depend fundamentally on both definitions and uses.

To use data flow analysis results as a basis for exception flow analysis, it must be determined which type of problem is likely to provide the most help. Detailed analysis of all of these types of problems is beyond the scope of this paper. However, their general characteristics provide a basis for selecting among them in light of the nature of exception flow. In data flow analysis the definition of a variable is passed down the flow graph. The references to a variable should follow its definition, that is, references

should occur below corresponding definitions in the flow graph. On the other hand, in exception flow analysis the propagation of an exception is represented by values passed up the context graph. The handling of an exception should occur subsequent to its raising, but in this case, since an exception is propagated up the context graph, its handler should occur above its raise. This suggests that a bottom-up analysis will be useful. Furthermore, an exception propagated into a node from any of it successors will affect the execution or elaboration of the context represented by that node. Thus, a set-union analysis will probably be needed because a set-intersection analysis will be too restrictive. Finally, the propagation of exceptions depends both on their raising and handling, so a data flow problem that depends on just one type of operation will be an inadequate model. The live variable problem is the only one of the four types of data flow problem that meets all of these criteria: bottom up, set union, and two operations. Detailed analysis of the other types of data flow problem confirms that they are less useful than the live variable problem for exception flow analysis [2], and these other problems will not be discussed further. The live variable problem and its exception flow analog are presented below.

2.3. The Live Variable Problem

The solution to the live variable problem is a set of variables (those that are live) for each node in the flow graph. These sets may be computed for the top or bottom of each node; only the bottom sets are of interest here.

Formally, a variable v is *live* at the bottom of a node i if and only if there is a definition-free path from i to a use of v in some node j below i in the flow graph [6]. The solution sets of variables are contained in the array LVBOT, where LVBOT(i) is

the set of variables that are live at node i. The solution depends on two additional arrays of sets, PRESERVED and XUSES. PRESERVED(i) is the set of variable definitions that reach node i and that are not *killed* in node i by redefinition of the variables; these are the definitions that are passed through the node. XUSES(i) is the set of variables that are used at node i without first being redefined; this is the set of variables whose uses are *exposed* to definitions at nodes along paths entering i from above. The set of successors of node i is denoted by succ(i). The solution to the live variable problem is the smallest set satisfying the equation:

$$\text{LVBOT(i)} = \cup_{\text{j} \in \text{succ(i)}} [(\text{LVBOT(j)} \cap \text{PRESERVED(j)}) \cup \text{XUSES(j)}] \qquad (1)$$

(where LVBOT(i) is empty if i has no successors).

2.4. The Live Exception Problem

The *live exception* problem is an exception flow problem that is analogous to the live-variable data-flow problem. To draw an analogy between data flow and exception flow it is necessary to establish some correspondence between the definition and use of variables and the raising and handling of exceptions. Because the solution to the LVBOT problem depends directly on both definitions and uses, it is straightforward to formulate an analogous exception flow problem that depends on both raises and handles. Let LEBOT be the name of the problem in which handles correspond to definitions and raises correspond to uses. Then PRESERVED has as its analog NOTHANDLED, where NOTHANDLED(i) is the set of exceptions that are propagated into node i and that are not killed by a handler in node i; these are the exceptions that are propagated through the node. Similarly, XUSES has as its analog XRAISES, where XRAISES(i) is the set of exceptions that are raised in node i without being handled. This is the set of exceptions whose raises are exposed to handlers along paths entering i

from above. This gives the equation

$$\text{LEBOT(i)} = \cup_{\text{j} \in \text{succ(i)}} [(\text{LEBOT(j)} \cap \text{NOTHANDLED(j)}) \cup \text{XRAISES(j)}] \tag{2}$$

(where LEBOT(i) is empty if i has no successors).

LEBOT(i) thus represents those exceptions which are not handled after they are raised below node i; in other words, for each exception x in LEBOT(i), there is a handler-free path from some raise of x at a node below i in the context graph to node i.

These definitions and equation (2) are not the most intuitive way to view exception flow analysis. In particular, XRAISES, although directly analogous to XUSES, is not a very natural concept. It partitions the exceptions raised in a context into two classes based on a characteristic of the context which is unrelated to the exception raising (i.e. whether those exceptions are handled or not).

A more intuitive definition of the live exception problem is the following:

$$\text{LEBOT(i)} = \cup_{\text{j} \in \text{succ(i)}} [(\text{LEBOT(j)} \cup \text{RAISED(j)}) \cap \text{NOTHANDLED(j)}], \tag{3}$$

where RAISED(i) represents the set of exceptions that are raised in node i and LEBOT(i) and NOTHANDLED(i) are as previously defined. Distributing $\cap$ over $\cup$ in equation (3) gives

$$\text{LEBOT(i)} = \cup_{\text{j} \in \text{succ(i)}} [(\text{LEBOT(j)} \cap \text{NOTHANDLED(j)}) \cup (\text{RAISED(j)} \cap \text{NOTHANDLED(j)})]. \tag{4}$$

Notice that

$$\text{RAISED(i)} \cap \text{NOTHANDLED(i)} = \text{XRAISES(i)}. \tag{5}$$

Equation (5) shows that equations (2), (3) and (4) are equivalent definitions of LEBOT. We will use equation (3) in our algorithm as it seems to us to be the most intuitive.

The solution to the live exception problem (LEBOT) provides the information necessary to answer many of the questions about exception flow. The solution sets include all exceptions that may be raised or propagated into a node. The solution sets do not include exceptions that are not defined for the successors of a node. Additionally, the solution sets include any exception for which there is a handler-free path from the point of the raise into the node, regardless of the number of such paths or the occurrence of handlers along other paths. (These two criteria, although obviously necessary, are not met by the exception flow analogs of most of the other data flow problems [2]). The sources of exceptions propagated into a node are easy to record. By comparing information about the exceptions propagated into a node with information on which exceptions are raised, handled, and reraised in the node, it is possible to determine which exceptions are propagated from the node. By computing this information for the main subprogram and for library packages is possible to identify the exceptions that may cause the execution of the program to be abandoned. By comparing the handlers in a node with the exceptions that may actually be propagated into or raised in the node it is simple to determine unnecessary handlers. The live exception problem is thus the central problem of exception flow analysis.

3. The Exception Flow Analysis Algorithm

An existing general-purpose data flow algorithm for live variable analysis [6] was adapted for our exception flow analysis problem. There are several classes of algorithms for solving the live variable problem. A round robin algorithm, in which information is propagated until stabilization is reached, is relatively easy to code, and with such an algorithm it is not necessary to be concerned with the reducibility of the graph. For

these reasons this algorithm was adapted for exception flow analysis despite the fact it is $O(n^2)$ in the worst case (where n is the number of nodes, or contexts, in the graph). Other algorithms, with better worst case behavior, could be adapted to improve the performance of the exception flow analyzer.

The results are presented in a bit vector framework. In the bit vector framework the solution sets are computed using arrays of bit vectors to represent sets of the appropriate type. Each array contains one bit vector for each node in the flow graph. Exception flow analysis can also be presented in the more general semi-lattice theoretic framework [3].

In Figures 3-1 and 3-2, the exception flow analysis algorithm is presented. It is written in Ada. The algorithm was adapted from the round robin live variable analysis algorithm presented in [6]. The adaptations include substitution of a context graph for a flow graph, solving an equivalent equation (as explained in section 2.4), and renaming variables to reflect the change in focus. Nodes of the graph are repeatedly visited in post order until a pass through the nodes has occurred in which no information is propagated.

There is, of course, more to the exception flow analyzer than the code shown in Figures 3-1 and 3-2. There are packages that define abstract data types for BitVector, ListOfContexts, ContextGraph and Context. There is the building of the context graph (and intermediate structures) and the computation of NOTHANDLED and RAISED. There is a report generator that derives, from LEBOT, answers to many questions concerning exception propagation and handling as discussed in section 1.3.

```
procedure efa (NOTHANDLED, RAISED: in ExceptionVector;
               C: in ContextGraph;
               NumberOfContexts: in positive;
               LEBOT: out ExceptionVector) is
               -- LEBOT, NOTHANDLED and RAISED are defined in section 2.4;
               -- C is the context graph to be examined;
               -- NumberOfContexts indicates the number of contexts in C;
               -- LEBOT is computed in this procedure;

-- Assumptions:
--   an ordering of the contexts from 1 to NumberOfContexts in
--      post order exists and is represented in the ContextGraph;
--   aliasing has been resolved;

-- Notes on types:
--  BitVector is defined as array(1..NumberOfExceptions) of boolean;
--  ExceptionVector is defined as array(1..NumberOfContexts) of BitVector;
--  nodes in ListOfContexts and ContextGraph are of type Context;
--
--  BitVector operations: zero, bv_and, bv_or;
--  ListOfContexts operations: Head, Next;
--  ContextGraph operation: Successors;
--      Successors takes a context graph and an index position
--         of a context in that graph as parameters and returns
--         a list of contexts that are successors to that context;
--  Context operation: NodeNumber;
--      NodeNumber takes a context as its parameter and returns
--         the index position of that context;
--
--  packages defining these types must appear in with and use clauses;

   newLE: BitVector;
   change: boolean;
   succ: ListOfContexts;
   k: 1 .. NumberOfContexts;
```

Figure 3-1: Ada program for exception flow analysis, part 1

4. Implementation Issues

Exception flow analysis is simpler than data flow analysis in some respects. Data flow analysis is complicated by the use of pointers and arrays in programs because the objects referenced by these constructs are determined dynamically and cannot be evaluated statically. In Ada there are no exception access types and no arrays of

```
begin
   -- initialization
   for j in 1..NumberOfContexts loop
      -- initially, no exceptions are in the set represented by LEBOT;
      LEBOT(j) := zero;
   end loop;
   change := true;

   -- propagate information by visiting the contexts in C in post order
   --   until a pass through the contexts is made with no information
   --   propagation;
   while change loop
      change := false;

      -- post order visit
      for j in reverse 1..NumberOfContexts loop
         newLE := zero;

         -- the successors of a context are those contexts that
         --   it calls or contains;
         succ := Successors(C, j);

         while succ /= null loop
            k := NodeNumber(Head(succ));
            newLE := bv_or(newLE, bv_and(
                              bv_or(LEBOT(k), RAISED(k)), NOTHANDLED(k)));
            succ := Next(succ);
         end loop;

         if newLE /= LEBOT(j) then
            LEBOT(j) := newLE;
            change := true;
         end if;

      end loop;
   end loop;
end efa;
```

Figure 3-2: Ada program for exception flow analysis, part 2

exceptions [1]. On the other hand, Ada allows task access types and arrays of tasks (and of task access types); these present problems not inherent in data flow analysis. The problems with task access types are addressed by allocating one context for each task type declaration and one context for each occurrence of a new operator for a task.

Analysis is simplified by the fact that exceptions are defined by task type rather than task object, so that all tasks of a given type raise the same exceptions (note that these exceptions may be propagated from an **accept** statement within the task even though they may not be propagated from the task itself). Furthermore, if an exception is raised during the activation of a task, the task becomes complete and the exception TASKING_ERROR is raised at the point of activation, regardless of the type or identity of the task. Thus, it is not necessary to discriminate among task objects of the same type.

We have made a number of assumptions during the design of the prototype exception flow analysis tool. These assumptions include simplifications, but they are nevertheless pessimistic about the raising of exceptions. The more important assumptions include:

1. The predefined exceptions can be raised at any point in any Ada program. This is clearly pessimistic as, for instance, an Ada program that does not use the tasking facility will never raise TASKING_ERROR.

2. Optimizing compilers will have no effects on the raising, propagation, and handling of exceptions except possibly to reduce the number of points at which exceptions may be raised [1]. While the effects of optimizing compilers are not considered in the prototype analyzer, this assumption produces a conservative analysis.

3. No exception checks are suppressed.

4. All program paths are executable. It is possible with data flow analysis to identify program paths that are not executable and then eliminate those paths from analysis, but the prototype will not do this.

5. Generic units are not analyzed. Generic instantiations can be analyzed, but they are assumed to be already resolved into subprograms or packages, as appropriate.

With respect to these assumptions, exception flow analysis can be made more precise

in several ways. These are discussed in the next section.

Much of the information about a context that is needed in exception flow analysis does not depend on the program in which the context appears. This includes information that can be determined solely from a context itself or from the contexts that it contains. To avoid recomputation of this information we construct context graphs in two phases. First, the program-independent information is calculated and collected in *containment trees*. Context graphs are then constructed by combining containment trees using program-dependent information about the calling structure of the program. Containment trees may be saved in a library for use in building context graphs for multiple program configurations.

5. Conclusions

This paper has discussed a new form of static analysis: exception flow analysis. By making an analogy between data flow and exception propagation, it has been possible to adapt the results of research into data flow problems and data flow algorithms.

A prototype exception flow analyzer ("AXA" for Ada Exception Analyzer) is under development at the University of Colorado. It is this prototype that has been discussed throughout this paper. Improvements to this tool are planned. The improvements will remove various sources of imprecision that produce overly pessimistic analyses. The analyzer can be made more precise in several ways. At least some of the predefined exceptions can be eliminated from consideration for many contexts simply by noting the kinds of expressions, statements and declarations in a context and discounting those predefined exceptions that cannot be raised. Also, if the presence of a SUPPRESS

pragma is noted, then a report on exception flow analysis may be appropriately qualified concerning the corresponding predefined exception. Exception flow analysis can be coupled with data flow analysis to avoid evaluation of program paths that are unreachable.

Data flow analysis is used to improve the code of programs. These improvements can take the form of optimizations by optimizing compilers as well as changes made by a software engineer as the result of some analysis. For instance, when two definitions of a variable are not separated by a use of that variable, the first definition can safely be removed from the program (or, of course, this may indicate that the name of the variable is misspelled or that a statement is missing). Exception flow analysis can similarly be used to improve Ada programs. For instance, in a frame in which an exception x is neither raised nor propagated, there is no need for a handler for x. Also, the definition of an exception x can be eliminated if x is never raised.

The optimization of exception flow analysis tools and the use of exception flow analysis to optimize code generation are subjects for further research. Once a practical exception flow analysis tool is available it will be possible to integrate exception flow analysis in the software development process and to investigate the use of exception facilities in large software systems.

Acknowledgments

Members of the Software Development Workshop at the University of Colorado, Ginger Barnett and Erik Sorensen, participated in this research.

References

1. *Reference Manual for the Ada Programming Language.* U.S. Department of Defense, ANSI/MIL-STD-1815A-1983, 1983.

2. Baker, Deborah A. and Sutton, Jr., Stanley M. Some Notes on Exception Flow Analysis. Technical Report CU-CS-320-86, University of Colorado, Department of Computer Science, 1986.

3. Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Principles of Programming Languages IV, ACM, January, 1977, pp. 238-252.

4. Cristian, Flaviu. "Correct and Robust Programs". *IEEE Transactions on Software Engineering SE-10*, 2 (March 1984), 163-174.

5. Goodenough, John B. "Exception Handling Issues and a Proposed Notation". *Communications of the ACM 18*, 12 (December 1975), 683-696.

6. Hecht, Matthew S.. *Flow Analysis of Computer Program.* North-Holland, New York, 1977.

7. Liskov, Barbara H. and Snyder, Alan. "Exception Handling in CLU". *IEEE Transactions on Software Engineering SE-5*, 6 (November 1979), 546-558.

8. Luckham, D. C., and Polak, W. "Ada Exception Handling: An Axiomatic Approach". *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 225-233.

9. Luckham, David, and von Henke, Friedrich W. "An Overview of Anna, a Specification Language for Ada". *IEEE Software 2*, 2 (March 1985), 9-22.

10. Neumann, Peter G. "Letter from the Editor". *Software Engineering Notes 10* (January, April, July, October 1985).

11. Osterweil, Leon J. Using Data Flow Tools in Software Engineering. In *Program Flow Analysis Theory and Applications*, Muchnick, S. S., and Jones, N. D., Ed., Prentice-Hall, 1981.

12. Turba, Thomas N. "The Pascal Exception Handling Proposal". *Sigplan Notices 20*, 8 (August 1985), 93-98.

13. Yemini, S. and Berry, D. M. "A Modular Verifiable Exception-Handling Mechanism". *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 214-243.