Towards the Generation of Efficient Code from Verified Programs

John McHugh

March 1984 Technical Report 40

Institute for Computing Science

2100 Main Building

The University of Texas at Austin

Austin, Texas 78712

(512) 471-1901

Abstract

An investigation was made of the characteristics of computer programming languages intended for the implementation of provably correct programs and of the characteristics of programs written in these languages. It was discovered that potential run time exceptions and the necessity of providing a rigorously correct implementation of exception handlers so dominate the potential control paths of programs written in verifiable languages that the usual code optimization techniques are ineffective. It was further discovered that the call intensive control structures of these programs, necessitated by verification constraints, also thwart optimization and lead to inefficient code. It is shown that theorems can be derived at potential exception sites which, if true, guarantee that the exception condition will never arise permitting removal of the exception path from the program's flow graph. These theorems are proved using the automatic theorem prover which is part of the program verification system. Is is also shown that many of the routine calls contained in verifiable programs may be reduced in expense by converting parameters to global variables or eliminated completely by expanding the called routines at their call sites. Both the exception suppression and call reduction techniques reduce the complexity of the program's call graph and facilitate conventional optimizations. Several examples are presented and the potential improvements in code size resulting from the application of these techniques are discussed.

Acknowledgements

I wish to thank Dr. Donald I. Good of the Institute for Computing Science and Computer Applications (ICSCA) at the University of Texas for suggesting the topic of this dissertation, and for the continuing interest and support which have made the work possible. Dr. J. C. Browne, my advisor, has provided special insight into some of the problems addressed herein and has provided both encouragement and support in the undertaking.

Much of the understanding that I have gained of exceptions and their consequences has been refined in the course of long conversations with colleagues at ICSCA. Richard Cohen has been especially helpful as have been Mike Smith, Larry Smith, and Larry Akers. Ann Siebert provided most of the pathological information flow examples used in connection with the ghost finding process, as well as considerable insight into the dataflow analysis process. Ann is also responsible for the implementation and proof of the flow modulator example.

Bill Wulf of Carnegie-Mellon University and Tartan Laboratories has provided useful discussion and encouragement on several occasions.

The staff of ICSCA, especially Ayshegul Musallam and Sandy Olmstead have generously supported this effort.

The work described in this report was the basis for my Ph.D. dissertation in computer science at The University of Texas at Austin. I wish to thank all the members of my committee: Dr. J. C. Browne, Dr. Donald I. Good, Dr. Gordon Novak, Dr. James Bitner, and Dr. W. W. Bledsoe for their kindness during its defense.

Finally, I wish to thank Ruth, my wife, and Mimi, my daughter, for their patience and support.

Table of Contents

List of Figures

List of Tables

Chapter 1
INTRODUCTION

This dissertation reports the results of an investigation into the relationships between the verification of computer programs and the efficient implementation of those programs. The principle vehicle for the investigation has been the Gypsy [Good 77] language and its associated implementation package, the Gypsy Verification Environment or GVE. Many of the results are directly applicable to Ada [Ada 79]. To a lesser extent, the results are applicable to other languages for which verification is an objective, although the lack of an integrated specification, proof and compilation system makes the application of these techniques more difficult.

We begin with an overview of verification and a discussion of those characteristics of verifiable programs which stand in the way of efficient implementation. This is followed by a review of prior work applicable to the present effort. Gypsy and the GVE are described in order to give the reader an understanding of the language and system on which the experimental work was based.

Chapter 4, discussing exceptions, their treatment and suppression is the crux of the dissertation. The possible presence of exceptions in a program completely dominates its control flow in all but the most trivial cases. That this is the case, may not be obvious to programmers unused to thinking in terms of verification, but as can be seen from the example in figure 1-1, almost every operation can raise some exception condition. The key here is **can** since most operations do not cause exceptions, although most experienced applications programmers can cite examples of operational programs which had been in use for years before failing due to some combination of input data which caused an exception. For program verification to be effective, exceptions and their control paths must be dealt with in a rigorous fashion. In the example, the action to be taken when an exception arises is not specified, but it is apparent that at each exception site the potential flow of control can follow one of two paths:

1. The "Normal" path to the next operation in the program.

2. The "Exception" path to the exception handling mechanism.

If the language definition rigorously defines the program state expected when control reaches an exception handling mechanism, the implementor will have little freedom to restructure the program in the face of these numerous control paths. Accordingly, a through understanding of exceptions and their handling is essential to the efficient implementation of a verifiable program. Although strong typing can aid in showing that many potential exceptions will not, in fact, occur, we also look at the role of the formal specifications in dealing with the exception suppression problem.

Following the discussion of exceptions are discussions of techniques for reducing calling overhead. We observe that verifiable programs tend to be highly modular and, as a consequence, call intensive at run time. Two techniques for reducing this overhead are evaluated. The first involves a global analysis of the program to identify those formal parameters which can be replaced by global variables without altering the semantics of the program. The second attempts to identify routines for which

```
TYPE INDEX = INTEGER[0..MAXINT];
TYPE TABLE = SEQUENCE [ENOUGH] OF ITEMS;
TYPE ITEMS = PENDING;

CONST MAXINT : INTEGER = PENDING;
CONST ENOUGH : INTEGER = PENDING;

FUNCTION FIND_IT (TAB:TABLE; ITM:ITEM):INDEX =
{ Search TAB for an entry and return its index or
  zero if no such entry is found }
BEGIN
VAR I : INDEX :=[1] 1;
RESULT :=[2] 0;
LOOP
  IF TAB[I][3] = ITM
    THEN
      RESULT :=[4] I;
      LEAVE
    ELSE
      I :=[5] I +[6] 1;
      IF I > SIZE(TAB)
        THEN
          LEAVE
        END
    END
END
```

Figure 1-1: A Simple Program Example

calls may be effectively replaced by expansions of the called routine. Again, we find that potential exceptions around call sites may play a significant role in determining the effectiveness of the optimization techniques.

Finally, we note that in Gypsy and other integrated programming and specification languages, code is often added to routines purely for the purpose of facilitating the proof. A technique is described which may be used to detect such code and allow its elimination.

A number of small examples and two substantial real programs, one verified and one unverified, are analyzed and the applicability of each technique is discussed in detail. Estimates are made of the improvement in code size and running speed which can be obtained over naive implementations of the same code.

Extending the work of Wulf on the development of automatically generated high quality compilers, we show how additional constraints could be placed on the PQC [Wulf 80] methodology to

[1] Assignment exception possible

[2] Assignment exception possible

[3] Subscript exception possible

[4] Assignment exception possible

[5] Assignment exception possible

[6] Overflow possible

allow it to deal with the issues which we believe characterize the implementation of verifiable programs.

Chapter 2
A REVIEW OF THE RELEVANT LITERATURE

The present work stands at the juncture of compiler and verification technologies. It is not surprising that there is relatively little prior literature directly applicable to the present effort. There is an extensive literature on program verification *per se*, but it is seldom concerned with implementation issues. At the same time, there is a considerable body of work having an indirect influence on the efficient implementation of a verifiable language. In this chapter, we will take a look at a number of these areas and their contributions to this work.

2.1 Verification

Computer program verification can be traced to the very beginnings of programming. John von Neumann [von Neumann 61] presents proofs of a variety of programs in one of the first manuals on programming. These proofs are particularly interesting since they are for machine language programs, and cover such aspects of program behavior as the absence of overflows, anticipating some of the present work. In the early 1960s, John McCarthy [McCarthy 63] suggested that proof checking programs be used to aid the mathematical proof process. The same paper also suggests that proving that programs possess some desired behavior is a substitute for program testing.

In one of the best known early papers in the field, Floyd [Floyd 67] associated assertions about the state of a program with certain statements in the program and used mathematical proof techniques to prove a final or exit assertion given an input assertion or precondition and axioms for the program constructs. In the same symposium, McCarthy and Painter [McCarthy 67] present a proof of a compiler for a simple arithmetic expression language for a single register machine. This proof avoids exception issues (as do many later programming languages) by assuming that the language being compiled and the target machine share the same semantics for arithmetic operations.

Considerable work has been done in the area of verification since Floyd's classic paper. The area of axiomatic definition of programming languages has received considerable attention. This work has provided a basis for relating the constructs and operators used in computer programming languages to similar concepts in mathematics, permitting the logics used for the proof of theorems in mathematics to be applied to computer programs or to theorems about programs. C. A. R. Hoare's monograph giving an axiomatic definition for the programming language PASCAL is one of the best known works in this area. [Hoare 76]

As interest in verification grew, attempts were made to apply verification to increasingly larger and more realistic programs. In 1970 the algorithms section of the Communications of the Association for Computing Machinery published a proof of a previously published sorting algorithm, [London 70] an act that started a controversy that persists in this day (see below). As attempts were made to prove increasingly lengthy and complex programs, it became evident that limitations on human ability to deal with the growth of proof size would limit hand generated proofs of programs, especially existing programs, to small programs and closed subroutines of some tens or at most hundreds of lines of code.

2.1.1 Constructive Correctness

Removing these practical limitations has followed two related paths. The first is the constructive proof/stepwise refinement methodology which has, in effect, given rise to what is now known as "structured programming" [Hoare 71, Mills 72]. In this approach, a program is first written in a very high level pseudo language and the specifications for each operation in the pseudo language are chosen so that proof of the overall program is relatively easy. As the program is refined, each pseudo operation is expanded in terms of lower level constructs. The expansion is limited so that it is relatively easy to prove that each expansion in fact satisfies its specifications. The process can be repeated through as many levels as is necessary until an actual implementation language or machine is reached. With or without formal verification, the general technique has been moderately successful in providing highly reliable programs to solve a number of real world problems. [Baker 72, McHugh 75].

2.1.2 Mechanical Verification

The second approach to solving the proof explosion is the use of automatic theorem provers as an aid to coping with the immense growth in proof complexity with program size. Much of the original work in automatic theorem proving was directed towards the proof of theorems from mathematics but the applicability of such techniques to the type of theorems that arise in verification has been relatively straightforward.

In practice, the link between a program and a mechanical theorem prover is a verification condition generator or VCG [Moriconi 80]. The VCG accepts as its input a representation of the program to be proved with assertions associated with some of the program's statements. Typically, assertions are supplied at the entry and exit of each routine in the program and at some place within each looping construct. The VCG traces the possible execution paths through the program from assertion to assertion. Based on the semantics of the language in question, the VCG produces an implication for each such path segment. The implications are of the form:

```
  <Assertion at the top of the path segment>
 &
  <State changes due to program statements in the path segment>
=>
  <Assertion at the bottom of the path segment>
```

Additional implications may be produced at intermediate points along the path. For example, it may be necessary to prove the entry assertions of routines called by statements in the path segment, or the language may require that some invariant hold at some or all statement boundaries in the program. The implications produced by the VCG are the input to the mechanical theorem prover.

There are a number of systems in existence that are applicable to real programs and programming languages [Boyer 75, Good 78, Luckham 77]. Limitations of the provers available and the mechanics of the verification condition generation process tend to constrain programs to be proved to be written using numerous relatively short functions and procedures. These may be thought of as analogous to the expansions in the stepwise refinement development scheme. As we shall see in the Gypsy system, this approach has a major and serious impact on the efficiency of the resulting programs.

Verification is not without its detractors. De Millo, Lipton and Perlis [De Millo 79] published a scathing commentary on the concept and practice of program verification, contrasting the process by which programs are verified with the process (largely social) by which mathematics grows through the proof and exchange of mathematical theorems.

2.1.3 The State of the Verification Art

There are indications that verification is on the verge of becoming routinely used as an aid in the development of highly reliable software systems. An increasing number of software development contracts, especially those originating from the Department of Defense and the security agencies are calling for some form of formal specification and verification. There also appears to be interest in the area among such high volume users of microprocessors as the automotive industry. At the same time, published results, i.e. reports of the verifications performed on real, production or production prototype programs are difficult to come by. Nonetheless, there are a number of significant and interesting results for either whole programs or subroutines which could be incorporated into larger systems.

2.1.3-A The Flow Modulator

The Gypsy system has been used to verify a message filter [Good 82] intended to assure that messages containing classified information (as characterized by patterns contained in a table) do not pass from a secure system to an unclassified network. This program is scheduled for actual production and deployment. It also serves as one of the major examples of this dissertation.

2.1.3-B Boyer and Moore

Boyer and Moore have verified a number of FORTRAN programs and subroutines. Most of their programs are relatively small and clearly constructed to illustrate some aspect of the verification process rather than taken from a real application. Their real time control program [Boyer 82] is clearly in this category, but it captures the behavior of a class very real applications. On the other hand, the majority vote [Boyer 81] and string matching [Boyer 80] subroutines, while in themselves short, can be included in larger programs. Much of this work was motivated by the SIFT effort.

2.1.3-C SIFT

NASA has sponsored a long term and ambitious effort to verify a multiprocessor avionics system which is required to have a very low probability of failure in flight. Despite rather optimistic claims which have appeared in the literature [Melliar-Smith 82] the accomplishments of the SIFT effort in the area of real code verification appear to be minimal. If is noted here because it represents an attempt, albeit an unsuccessful one, to extend verification down to the architecture level on one hand [Boyer 83] and to real time [Boyer 82] and numerical algorithms on the other. These areas are generally considered to be beyond the state of the art at the present time, but they represent issues which must be handled if verification is to be extended to a variety of critical control applications.

2.1.3-D Security Policy

As noted above, much recent work in program verification has been motivated by security considerations. A tutorial on security models and a survey examining the applicability of four specification and proof methodologies to a security example appears in the September 1981 issue of Computing Surveys [Landwehr 81, Cheheyl 81]. It is worth noting that of the four systems described, Gypsy, HDM, FDM, and Affirm, only the Gypsy system had a well integrated program verification facility although the others were developing support for code verification.

2.2 Exceptions and Exception Handling

A number of programming languages and language processors deal with exceptions, either by providing mechanisms for transfer of control when exceptions arise or by requiring that exceptions be eliminated from the code. In addition, there is a large body of verification related literature addressing the exception elimination problem in various ways. The following section discusses language features for exceptions. Verification aspects of the issue are covered in a subsequent section.

2.2.1 Exception Classification

Goodenough [Goodenough 75] discusses various exception sources and handling techniques in detail. Both the dynamic semantic errors and the implementation deficiencies of the present work fall into the category which Goodenough calls "Domain Failure", that is the arguments to the operator raising the exception fail to satisfy some entry specification of the operator definition. Transfers of control to exception handlers use Goodenough's "escape" mechanism which forbids resumption of the defective operation.

Goodenough's exception mechanisms cover a much broader range of exception signaling and handling techniques than are dealt with in the present work. Mechanisms for signaling exceptions which do not necessarily indicate an error condition (such as the end of a page during a printing operation), dealing with it (by ejecting the page and printing a new page header), and resuming the interrupted operation are valuable programming constructs but are probably beyond the ability of present specification and verification techniques to deal with adequately.

2.2.2 Languages with Provisions for Exception Handling

A number of languages have dealt with the issue of run time exceptions in one way or another. We will briefly discuss four of these languages in the paragraphs which follow.

2.2.2-A PL/I

PL/I [PL/I 72] is a general purpose programming language originally implemented for the IBM 360 series of computers and subsequently for the Honeywell Multics System and others.

PL/I recognizes a wide variety of exception conditions and provides an elaborate mechanism for handling them when they are raised. The PL/I exceptions include array bounds errors, various types of overflows, and numerous others. Within the handlers, which are dynamically associated with the code in which exceptions may arise, a variety of actions may be taken including abandonment or resumption of the exception raising operation. PL/I is such a complex language that, despite a formal definition, relatively little has been done with the verification of PL/I programs. A number of optimizing compilers have been built, but they have not been concerned with the strict enforcement of the languages semantics.

2.2.2-B Ada

The Department of Defense Standard Language Ada [Ada 79] defines a limited set of exceptional conditions and makes provision for user written handlers for them. The mechanism is similar to that defined for Gypsy *infra* except that condition names may not be passed as routine parameters. At the present time, no production quality Ada compiler exists and little or no work has been done on Ada optimization. There is considerable interest in Ada verification [Young 80, Luckham 80a] and a specification mechanism is being proposed [Anna 80] on which it is presumed that a verification effort will be based. Given that an axiomatic definition of the Ada [Luckham 80b] exception mechanism exists, it is reasonable to assume that Ada verification will address exception issues as well. The difficulties of implementing Ada make it unlikely that techniques proposed here for effective code generation while preserving verifiable semantics will ever find their way into an operational Ada compiler, however applicable they may be.

2.2.2-C Euclid

Euclid [Lampson 77] is a Pascal derivative designed specifically for verifiable systems programming. Euclid has a particularly simple and elegant solution to the exception problem. A legal Euclid program may not have any runtime exceptions. A Euclid compiler must prove that runtime exceptions will not occur. If it is unable to do so, it must create a "legality assertion" [Wortman 79] to check for the exception. If it is not possible to prove the legality assertions, and they are in fact false at runtime, then the program is not a legal Euclid program. It is possible to force a Euclid compiler to produce code for runtime checking of both programmer supplied and legality assertions. If the checking code fails, a runtime error is issued, informing the user that the program has departed from legality. No provision is made for user written code to handle such a fault.

The Euclid project is far behind schedule. At the present time a compiler exists which appears to support most of the language although the degree of implementation of legality assertions is uncertain. A considerable amount of theoretical work on the legality assertion mechanism has been done [Elliot 82, Patkau 79] but there is no indication that a connection between legality assertions and code sequences whose legality is asserted exists. Since the Euclid compiler is intended to produce efficient code [Wortman 81], it would appear that this link is crucial for preserving verifiability and legality.

At the present time, a Euclid verifier is being designed. It is not known when it will be available for use or how closely it will be linked to the legality assertion/code generation mechanisms.

2.2.2-D Pascal

Pascal [Jensen 74] was originally designed as pedagogic tool for teaching structured programming. Because of this, the language is designed to permit rapid one pass compilation. The language is simple enough to place compiler construction within the ability of most advanced undergraduates in computer science. As a result, there are Pascal implementations for most widely used processors and operating systems, but relatively few of these produce efficient code.

Pascal has been the basis for a number of verification efforts [Stanford 79, Melliar-Smith 82]. The proof of the absence of runtime errors in Pascal has also been the subject of several investigations [Eggert 81, German 78]. As in the work on Euclid, no attempt has been made to link the runtime error work directly to code generation and or optimization.

2.2.2-E Gypsy

Gypsy [Good 78] is a Pascal derivative on which much of the present work is based. Gypsy is described in more detail in a subsequent chapter. Let it suffice to note at this point that Gypsy was designed to support program verification and that an integrated verification and compilation system exists for the language. The language semantics take exceptions into account and the language provides for user written exception handlers.

2.2.3 Unimplemented Languages

In addition to the above languages, there have been several language projects which had as their avowed purpose the production of highly efficient language implementations of languages featuring specifications and/or verifiability.

2.2.3-A Alphard

The Alphard language [Wulf 78] was designed at Carnegie-Mellon University in the late 1970s. It had as its avowed aims the implementation of a verifiable language which would produce "better code than is typically produced by assembly language programmers" *ibid* page 2. Had the language been implemented, it is unlikely that assertions would have had an influence on the code generation process as "the language in which the assertions are written, however is not defined by Alphard. The choice of that language is, we believe, a private matter between the programmer and verifier" *ibid* p.5. The preliminary Alphard report did not deal with exceptions although it recognized that a production version of the language would have to do so.

2.2.3-B Zeno

The Zeno language [Ball 79] was designed at the University of Rochester to support research in distributed systems. While verifiability was not a consideration in the design, assertions were to be used as an aid to code generation by supplying additional information about the intended program. Zeno was never implemented [Low 83].

2.2.4 Exceptions, Verification, and Fault Tolerance

Cristian and others at the University of Newcastle have published a number of papers [Cristian 82, Best 81] which link the treatment of runtime exceptions with both formal verification and software fault tolerance. The exception model used in this work is similar to that proposed for Ada, with exception specifications similar to those used in Gypsy. The paradigm for fault tolerance uses the handling of an exception to invoke a recovery block or similar mechanism.

2.3 Optimization and Code Generation

Concern with the efficiency of the code generated by high level language compilers can be traced to the very earliest compilers [Backus 67]. In the early days of high level languages, it was thought that programmers would not use a language for which the code produced did not approach the speed and size of hand coded machine code. Not surprisingly, much of the early optimization work was largely empirical and was based on languages such as FORTRAN which were used primarily for large scale, computer bound, scientific computations. Cocke and Schwartz [Cocke 70] provide an excellent summary of the state of the compiler and optimization arts as of the end of the 1960's.

In the early to middle 1970's parallel advances in graph theory and programming language semantics placed many of the earlier *ad hoc* program optimization techniques on a firmer theoretical basis. Additional advances in machine speeds and address space made implementation of a number of theoretically feasible techniques for global program optimization practical as well.

2.3.1 Flow Analysis and Graph Theory

Aho and Ullman [Aho 77] provide a concise exposition of flow analysis based optimization techniques developed through the mid 1970s. These techniques are based on the identification of basic blocks, which are consecutive sequences of operations which are entered only prior to the first operation and exited only after the last operation in the sequence. By analyzing the behavior of the code contained within the basic blocks and the flows between them a number of transformations can be performed on the intermediate representation which should increase the performance of the program. Typical optimization actions include the identification of common subexpressions, movement of loop invariant code out of loops, dead variable analysis, etc.

The interest in structured programming, i.e., programming using control structures having a single entry and exit, which can be traced to an infamous letter by Dijkstra [Dijkstra 68], has had a major influence on the feasibility of optimization analysis. The flow graphs resulting from structured control constructs are reducible. Reducible graphs may be manipulated with fast (in many cases order (E) where E is the number of edges in the graph) algorithms [Graham 76].

Of particular interest to the present work is the fact that the flow analysis and optimization literature assumes that the operations contained in the program intermediate representations are exception free and that arithmetic operations have integer (or real) semantics as opposed to machine integer or floating point semantics. As we shall subsequently demonstrate, these assumptions are inadequate for verification.

Chapter 3
THE GYPSY LANGUAGE AND VERIFICATION SYSTEM

This chapter provides an overview of the Gypsy language and its implementation and describes the Gypsy Verification Environment, or GVE as it is known.

3.1 The Gypsy Language

The Gypsy language is a Pascal derivative whose features have been constrained to facilitate verification. Figure 3-1 is an example of a simple Gypsy program which illustrates some of the features discussed below. The language is procedure structured with procedures and functions comprising its basic building blocks. Procedures and functions, which are collectively referred to as routines, are grouped into collections called scopes for modularization and information hiding purposes. In addition to routines, scopes may contain type definitions, global constants and lemmas. Collectively, Gypsy routines, types, constants, and lemmas are refered to as units. There are no global variables. Access to a scope or units may be restricted, providing for information hiding both within a scope and between scopes.

Gypsy is strongly typed. User defined types are built up from primitive types (integer, character, rational) by mapping (scalar types), aggregation (records), iteration (arrays), and restriction (subranges). Dynamic types are buffers, sequences, sets and mappings. The elements of dynamic types may not contain a recursive reference to the containing type. The buffer type is provided for inter-process communication. A notion of buffer histories aids in the proof of multiple-process programs. The types of the objects passed through buffers may contain any type except one containing another buffer. An abstract type mechanism is supported in which only specified routines may have access to the internal structure of objects of the abstract type.

Gypsy has most of the usual structured control constructs, *i.e.*, procedure and function reference, loop, if and case. A cobegin statement contains a number of procedure calls that are to be activated concurrently, providing for multiple-process programs. It has one unusual control structure which requires special attention. This is the "when" block, and its associated "signal" statement. Gypsy has an entity called a "condition" which provides a method for handling a large class of potential run time errors. There are a variety of predefined conditions that are associated with the semantics of the various Gypsy operators (*e.g.*, the multiplication operator has the potential for signaling the "multiplyerror" condition.). In addition, the user may declare conditions of his own and may signal those or the predefined ones. A "when" block containing condition handlers may be attached to any statement or group of statements. The scope of the when block is limited to the associated statements, but the structures may be nested. When a condition is signaled, either explicitly or as a result of an operator failure, control is passed to the nearest (in terms of nesting) when block that contains a handler for that condition. Conditions for which no explicit handler is provided are

```
SCOPE factor =
BEGIN

TYPE b1 = BUFFER (1) OF INTEGER;
NAME minint, maxint FROM implementation;
TYPE pdp_11int = INTEGER(minint..maxint);

PROCEDURE main (VAR ttyin:b1; VAR ttyout:b1) =
  BEGIN
    VAR n1:pdp_11int;
    VAR r1:pdp_11int;
    COND toobig, toosmall;
    LOOP
      RECEIVE n1 FROM ttyin;
      r1 := fact (n1) UNLESS (toosmall, toobig);
      SEND r1 TO ttyout;
    WHEN
     IS toobig, toosmall: SEND -1 TO ttyout;
    END;
  END;

FUNCTION fact (n:pdp_11int) : pdp_11int unless (toosmall, toobig)=
  BEGIN
    ENTRY (n ge 0) OTHERWISE toosmall;
    EXIT CASE (
         IS NORMAL: result = factorial(n);
         IS toosmall: (n LT 0);
         IS toobig: factorial(n) > maxint );
    VAR i:pdp_11int := 0;
    COND multiplyerror;
    result := 1;
    LOOP
      ASSERT      (result = factorial(i))
              AND (i IN [0..n]);
      IF i GE n THEN LEAVE END;
      i := i + 1;
      result := result * i;
    WHEN IS multiplyerror: SIGNAL toobig;
    END;
  END;

FUNCTION factorial (m:INTEGER) :INTEGER =
  BEGIN
    ENTRY m GE 0;
    EXIT (ASSUME result = IF m = 0 THEN 1
                                    ELSE m*factorial(m-1)
                          FI);
  END;
END;  {scope factor}
```

Figure 3-1: Factorial - A simple Gypsy program

converted to a universal "routineerror". This conversion takes place on return from any routine that does not handle the raised condition or have it as an explicit formal parameter. Thus "indexerror" raised in a routine, at a location for which no local handler exists, will appear as "routineerror" signaled from its call site in the calling routine.

In Gypsy 2.0 on which this work is based, the predefined condition names are imported into any routine invoking the operations which signal them. There is no method for supplying an actual condition parameter to, say, a subscripting operation as can be done with user written routines.

In addition to the lack of global variables, Gypsy has related features that are intended to simplify the proof process. These include a prohibition of side effects by functions (*i.e.*, no variable parameters), a prohibition against aliasing of actual parameters to the same procedure or to different procedure calls in a cobegin. The exception is the buffers which are immune to the effects of aliasing due to the nature of their operations.

Many of the Gypsy operators can raise one or more predefined conditions, allowing an orderly description of program behavior in the presence of run time errors. In theory, it is necessary to prove a Gypsy program correct in the face of every possible signal (or to prove that such signals will not be made), but in practice, most proofs are based on an assumption that only explicitly signaled conditions must be dealt with. The impact of these assumptions on optimization will be dealt with below.

3.2 Changes in the Gypsy Language

Language design is an art. Design of a language for the writing of verifiable programs is an area in which there is not a lot of experience to draw on. Experience with Gypsy 2.0 has indicated that minor changes are needed in a number of areas and a new version of the language is currently under development. Gypsy 2.1, as it will be known, differs from Gypsy 2.0 in a number of minor ways. The most important of these insofar as the present work is concerned is the condition parameter passing mechanism and routine call mechanism.

As a result of the present work, the condition passage mechanism of Gypsy 2.1 has been altered so that it is possible to pass actual condition parameters to operations such as subscripts. If no actual condition parameter is supplied, "routineerror" is used as the default. In Gypsy 2.1 every routine and/or operator has an additional implicit condition parameter, "spaceerror", which is used to indicate that the computation was unable to proceed due to lack of a memory resource. In Gypsy 2.1, it is also possible to provide actual parameters to the "valueerror" conditions of assignment and parameter passage.

Gypsy 2.1 also specifies the order in which the steps of the call mechanism are performed. This allows us to ensure that, given a call that could raise more than one condition, the condition raised is predictable.

3.3 The Gypsy Verification Environment

Gypsy programs are developed, manipulated and proved in an environment supported by a number of Lisp routines and a database containing the program and information about it. These routines are loosely tied together with a control routine called the "top level" which manipulates the other routines and the database. While conceptually elegant, the system is rather intractable and shows its origins as disjoint programs written by a number of programmers with varying motivations almost all at variance with the concept of producing a well-integrated production quality system.

The following is a summary of the major components of the system.

- Top level - Coordinates the operation of the system. This provides a user interface to the system and supports the incremental development features [Moriconi 77] and utility functions.
- Parser - Operates in several passes to do syntactic and semantic (type consistency) checking on Gypsy units. The output of the parser is a combination of symbol table information and a prefix representation of the program. Both are stored in the database. The syntactic pass of the parser is based on the system described in [Burger 74].
- Infprint - Unparses the program to produce listings. Unfortunately comments are lost in the parse so the utility of the listings is limited.
- Editor - An experimental Gypsy structure editor which manipulates prefix according to Gypsy syntax rules was written by Dwight Hare as a masters thesis [Hare 79]. In addition to the structure editor, there is a link from the GVE to EMACS, a powerful general purpose editor. Either of these editors may be used to manipulate Gypsy text within the context of the GVE. The utility of these editors for the maintenance of production code is limited because the internal forms used in the GVE do not preserve comments from the program text.
- Verification Condition Generator - Generates the theorems to be proved from the user supplied specifications and the program text.
- Prover - A natural deduction type theorem prover used interactively to prove the verification conditions. [Bledsoe 75]
- Expression Evaluator - Evaluates an expression represented as prefix. If the expression is not evaluable (constant) a canonical form is produced.
- Interpreter - Similar to the expression evaluator for program units. Support for some features such as concurrency is lacking.
- Translator - Translates Gypsy into some other form for execution. The current translation targets are Bliss [Smith 80] and Ada [Akers 83].
- Optimizer - The result of this effort. The optimizer communicates with other components via annotations placed in the internal representation of Gypsy programs.

3.4 The Semantics of Gypsy

Any language which is intended to be verified must have a rigorously defined semantics. In Gypsy 2.0 [Good 78], the semantics of the language are not as well defined as one would like. Cohen [Cohen 83] has produced a formal definition of the revised language Gypsy 2.1 which is defined in terms of path tracing rules and functional definitions for Gypsy operations. This is, in effect a formalization of the *de facto* situation in the Gypsy 2.0 GVE in which the semantics of the language are really defined by the verification condition generator and expression evaluator.

In [Good 78], operators are given functional definitions with Gypsy style specification functions. These functions define the programming language version of the operators in terms of ideal, integer (or rational), operations. For example, the Gypsy operator "INTEGER#ADD" ("+" in the Gypsy lexical representation) has its exceptional[7] return defined by the following function

[7]The normal functioning of Gypsy operators is defined, in effect, by the symbolic evaluator, XEVAL. It would be more satisfying to have, as part of the definition, a meta notation suitable for describing both the normal and exceptional functioning of all operators.

```
FUNCTION INTEGER#ADD (X, Y : INTEGER):INTEGER
         UNLESS (ADDERROR) =
           BEGIN
              EXIT CASE (IS ADDERROR:INTEGER#BADADD(X,Y));
           END;
```

In this case the semantics of "adderror" are defined in the (implementation dependent) function "INTEGER#BADADD". For the PDP-11, a 16-bit 2's complement machine, an appropriate definition of "INTEGER#BADADD" would be:

```
FUNCTION INTEGER#BADADD (X, Y : INTEGER)
                 :BOOLEAN =
         BEGIN
             RESULT:=((X+Y)<(-32768)) OR
                     ((X+Y)>(32767));
         END;
```

where the operators +, <, and - are the integer, **not** the Gypsy operators.

The functional notation becomes a bit strained when it is not possible to concisely state the required properties of function within the framework of the Gypsy notation. For example [Good 78] defines the array selection operation as

```
TYPE T=ARRAY[INDEX] OF ELEMENT

FUNCTION ARRAY#SELECT (A:T;I:INDEX):ELEMENT
         UNLESS    (INDEXERROR)=
           BEGIN
             EXIT CASE(
              IS NORMAL:
               RESULT =
                "THE VALUE Z OF THE COMPONENT (I,Z) OF A";
              IS INDEXERROR:
               NOT I IN "INDEXSET");
           END;
```

While we can understand this definition, there is no way to use the definition within GVE to prove properties of individual array accesses. As we shall see in chapter 4, it is possible to restate the Gypsy operation definition functions in such a way as to make it possible to use them within the GVE.

Chapter 4
CONDITIONS AND SIGNALS

4.1 Conditions

Conditions, their associated signaling mechanisms and their handlers provide an orderly method for dealing with abnormal or exceptional events during the execution of a Gypsy Program. Although Gypsy 2.0 served as the vehicle for this investigation, the results are directly applicable to the exception signaling and handling mechanisms of Ada. We also claim that the exception analysis proposed or its equivalent must be performed for any verified program whether exception handling is included in the language or not. This is because we must be able to assure that the execution of the program is consistent with the assumptions under which the verification was performed.

As a result of the analysis performed in connection with this dissertation, major changes in the Gypsy condition mechanisms have been proposed. While the work reported here is based on the existing Gypsy 2.0 implementation, the proposed changes and their consequences are discussed in Section 3.2.

Conditions are divided into two major categories, predefined and user defined. The predefined conditions may be further divided into those which indicate shortcomings in the implementation of the language (Table 4-1) and those which indicate dynamic semantic errors in the program (Table 4-2).

Predefined conditions may be signaled implicitly, as a side effect of the operation with which they are associated, or by any of the explicit signaling mechanisms discussed below. User defined conditions will only be signaled explicitly although they may be reflected into a calling environment by default and appear to be implicit signals in that environment.

Syntactically and semantically, conditions are somewhat different from other named entities in Gypsy. Even the predefined conditions (with the exception of "routineerror") must be declared before they can be mentioned explicitly in a signaling or handling mechanism. This does not prevent the condition from being raised by any of the implicit means. However, the handling of undeclared signals is possible only through the "else" arm of a "when" or through conversion to "routineerror".

"Routineerror" is a special predefined condition which appears as an implicit formal parameter of every Gypsy function and procedure including the main procedure. "Routineerror" in the calling environment is always the implicit actual parameter that matches the formal "routineerror" in the called environment at all call sites. There is no provision in Gypsy 2.0 for supplying an explicit actual condition parameter to match "routineerror". Within a routine, any conditions signaled that are not handled in the routine are converted into "routineerror" in the return to the calling environment. Condition handlers will be described in detail below.

Table 4-1: Conditions Arising from Deficiencies in Implementation

Condition	Operator	Functions	Citation
adderror	+	integer#add	7.7.6[8]
	+	rational#add	7.7.6
divideerror	div	integer#divide	7.7.4
	/	rational#number	7.7.5
	/	rational#divide	7.7.5
minuserror	-(unary)	integer#minus	7.7.6
	-(unary)	rational#minus	7.7.6
multiplyerror	*	integer#multiply	7.7.6
	*	rational#multiply	7.7.6
powererror	**	integer#power	7.7.6
		rational#power	7.7.6
receiveerror	receive	buffer#receive#T	10.2.7
senderror	send	buffer#send#T	10.2.7
spaceerror[9]	adjoin	set#adjoin	7.10.1
	omit	set#omit	7.10.1
	union	set#union	7.10.1
	intersect	set#intersect	7.10.1
	difference	set#difference	7.10.1
spaceerror (cont'd)	union	mapping#union	7.10.2
	intersect	mapping#intersect	7.10.2
	difference	mapping#difference	7.10.2
	with	mapping#delete	7.10.2
	with	mapping#componentassign	7.10.2
	[..]	sequence#subselect	7.10.3
	append	sequence#append	7.10.3
	with	sequence#componentassign	7.10.3
subtracterror	-(binary)	integer#subtract	7.7.6
	-(binary)	rational#subtract	7.7.6

[8]Section in [Good 78].

[9]Given that dynamic types can occur as components of static types, i.e. a sequence field in a record, it is possible for other operators in Gypsy to issue "spaceerror". [Good 78] does not take this into account. See 3.2

Table 4-2: Conditions Arising from Dynamic Semantic Errors

Condition	Operation	Functions	Citation
aliaserror	procedure call	---	8.1
caseerror	case stmt	---	10.3.2
indexerror	[ ]	array#select	7.9.2
	with [ ]	array#componentassign	7.9.2
	[ ]	mapping#select	7.10.2
	with mapomit [ ]	mapping#delete	7.10.2
	[ ]	sequence#select	7.10.3
	[..]	sequence#subselect	7.10.3
	with [ ]	sequence#componentassign	7.10.3
membererror	omit	set#delete	7.10.1
negativeexponent	**	integer#power	7.7.6
	**	rational#power	7.7.6
nonunique	union	mapping#union	7.10.2
	intersect	mapping#intersect	7.10.2
nopred	pred	s#pred	7.7.1
nosucc	succ	s#succ	7.7.1
overscale	scale	s#scale	7.7.1
powerindeterminant	**	integer#power	7.7.6
	**	rational#power	7.7.6
routineerror	call	---	10.5
	call	---	8.1
sumerror	difference	mapping#difference	7.10.2
underscale	scale	s#scale	7.7.1
valueerror	assignment	---	4.7
	assignment	---	5
	call	---	8.1
	assignment	s#assign	10.2.2
varerror	call	---	8.1
zerodivide	div	integer#divide	7.7.4
	/	rational#number	7.7.5
	/	rational#divide	7.7.5

4.2 Signals

Gypsy provides a variety of mechanisms for explicitly signaling a condition. The most obvious of these is the signal statement. When executed, the signal statement causes control to be transferred to the most tightly enclosing handler for the condition named in the statement. The otherwise clause of an assertion also signals a named condition and has the same effect at run time as an if statement containing the negated assertion as its Boolean expression and the signal statement as its then part. Thus

```
    Assert foo otherwise fooerror;
```
is equivalent, at run time, to

```
    If not foo
       then
          signal fooerror
       end;
```

Function and procedure calls can also signal conditions. Any user defined function or procedure can signal the predefined condition "routineerror". This is built into the call/return mechanism and it is not possible to match an actual condition parameter to the implicit formal condition parameter "routineerror". User defined functions and procedures may have explicit formal condition parameters as well, and the names of these parameters may come from the list of predefined conditions or they may be the names of user defined conditions.

When a function or procedure is invoked, the invocation may contain actual condition parameters which match the explicit formal condition parameters of the routine. In this case, signaling an unhandled formal condition parameter within the routine has the same effect as signaling the corresponding actual at the call site. Since functions have no side effects, the order of expression evaluation is not material and no damage can be done in the calling environment by aborting an expression evaluation with a signal from a function invocation. This is not the case with procedures and, in general, the state of the procedure's var parameters is unknowable when the procedure returns a signal. As an aside, it is interesting to note that although "routineerror" was intended as a method of hiding routine implementations from the caller, it may be possible to discover a great deal about the implementation of a routine which signals "routineerror" by examining its var parameters. Parameter passage by value/result with the result being supplied only on normal return would eliminate this possibility.

Actual condition parameters are optional in Gypsy 2.0. If no actual condition parameters are provided, the call is interpreted as though actual condition parameters with the same names as the formals had been provided by the caller. If any of these names are known in the calling environment (by declaration as local conditions or formal condition parameters) the signals raised by these implicit actuals may be handled by any of the mechanisms discussed below. Names not known in the calling environment can be handled only by conversion to "routineerror" at the next calling level or by "else" parts of "when" clauses. This injection of condition names into the calling environment gives rise to a class of subtle programming errors which will be eliminated in Gypsy 2.1.

The signaling of conditions from omitted actual condition parameters may be viewed as an implicit signaling mechanism. Most Gypsy operators are able to signal one or more of the predefined conditions. In most cases, it is sufficient to think of the operator notation as shorthand for a call on a function which performs the indicated operation. This function has the appropriate formal condition parameters and signals them when necessary. Because the shorthand notation does not provide for actual condition parameters to be passed to the operator implementation functions, the implicit condition parameter mechanism applies to Gypsy operators as well.

The model of a Gypsy operator as a function with possible condition parameters is satisfactory for the predefined conditions which represent implementation deficiencies. It is also satisfactory for most of those which represent dynamic semantic errors. There are three notable exceptions, "aliaserror", "valueerror", and "varerror". "Aliaserror" occurs when two or more actual parameters in the same call operation refer to the same object and at least one of the parameters is being passed as a variable. Figure 4-1 shows a program fragment which admits several "aliaserror" occurrences.

```
TYPE int = INTEGER[minint..maxint];
TYPE index = INTEGER[0..37];
TYPE intarray = ARRAY [index] of int;

PROCEDURE arm1 (VAR i : int; VAR ia : intarray;
                                  j,k : index) = PENDING;
PROCEDURE arm2 (VAR i : int) = PENDING;
PROCEDURE arm3 (VAR ia : intarray; VAR j : int; k : int) = PENDING;

PROCEDURE alias( ... ) =
BEGIN
  VAR i : int;
  VAR iia, jia : intarray;
  .
  COBEGIN arm1 (i1*, iia2, 3, i1);
          arm2 (iia[12]2);
          arm3 (jia2, jia[mod(i,38)]2,3, jia[13]3);
  END;
  .
 END;
* Superscripts refer to enumerated discussion of "aliaserror" varieties in
the text
```

Figure 4-1: "Aliaserror" examples

The superscripts in the figure refer to the types of "aliaserror" as follows:

1. Simple "aliaserrors" occur when exactly the same object is passed to two or more formal parameters of a routine and at least one is a formal variable. This situation can always be detected at parse time during the static semantic checking phase and is on no further interest here. While the example shows a simple "aliaserror" within a single procedure call, one could just as well exist between separate arms of the COBEGIN.

2. Structure-element "aliaserrors" occur when a structured object, such as an array, is passed as a parameter along with a reference to one or more of its elements and at least one of the references is a formal variable. The examples show both intraprocedural and interprocedural "aliaserrors" of this type. Again, errors of this type are parse time detectable and of no interest as far as optimization is concerned.

3. Element-element "aliaserrors" occur when two elements of the same structured object appear as parameters in the same calling context and at least one of them is a formal variable. If all the element selectors are not parse time computable, then it is not possible to determine at parse time whether or not an "aliaserror" will occur since the identity of the objects passed is not known until the selectors are evaluated. In the example, "aliaserror" should be signaled if
   ```
   MOD(i,38) = 13
   ```
   is ever TRUE.

Of the three cases discussed above, only 3 need have any impact on implementation. As is the case with other exceptions, proving that the "aliaserror" circumstances will never occur can be a substitute for run time code to compare the selector values. While we can easily define the proposition to be proved for a given "aliaserror" situation, there appears to be no easy way to capture the semantics of "aliaserror" in general and encapsulate it in a function definition as we have done with other operators in Appendix A.

"Valueerror" occurs when an attempt is made to assign a value to an object which is outside the value set of the type of the object. There are several circumstances which can give rise to a "valueerror". The two most common of these are the assignment statement and parameter passage to functions and procedures. Figure 4-2 shows some fragments of code which give rise to potential "valueerrors".

```
    CONST minint = -32768;
    CONST maxint = 32767;
    TYPE int = INTEGER[minint..maxint];
    TYPE int1 = INTEGER[0..maxint];
    TYPE b = BUFFER OF int;
    TYPE b1 = BUFFER OF int1;

    FUNCTION foo (x,y : int1) = PENDING;
    PROCEDURE proc (bf : b1; x,y : int1) = PENDING;
    PROCEDURE bar (...) =
    BEGIN
      VAR i,j: int;
      VAR i1,j1: int1;
      VAR bfr : b;
      VAR bfr1 : b1;
           .
           .
      i1 :=* i;
      ... foo(i*,j1) ...
      proc (b1, i1, j*);
      SEND i to bfr1*;
      RECEIVE i1* from bfr;
           .
           .
    END;
* Possible "Valueerror" site
```

Figure 4-2: "Valueerror" examples

In the examples considered all the potential "valueerrors" arrise when an attempt is made to assign from an object of type "int" having a range including both negative and positive numbers to an object of type "int1" which covers only zero and positive numbers. Both direct assignments and actual-formal bindings are considered. The "valueerrors" from the buffer statements SEND and RECEIVE can be considered as having the effect of a direct assignment. In addition to the constructs shown in Figure 4-2, the Gypsy GIVE and MOVE statements can also raise "valueerror". The semantics of these statements can be expressed in terms of SEND and assignment operations and need not be considered further.

As was the case with "aliaserror", it is not possible to provide a functional representation which captures the semantics of "valueerror" for all pairs of Gypsy assigning and assigned objects. However, for any potential "valueerror" site, it is possible to express the conditions under which the operation will be exception free. For example, the assignment statement in figure 4-2 will not signal "valueerror" if

```
    i IN [0..maxint].
```

"Varerror" occurs at a parameter passage site for variable parameters if the types of the actual and formals are such that it would be possible for an assignment to the formal within the called routine to violate the type constraints of the actual. Figure 4-3 shows two potential "varerror" sites. The first site will always signal "varerror", a fact that can be determined during semantic analysis and diagnosed as a semantic error in the program. The second is more subtle. Gypsy 2.0 allows run time parameterized subrange types. The type of the actual in the call to "var2" depends on parameters of the calling routine. Whether a "varerror" occurs depends dynamically on the type of i2 which depends on the values of minv and maxv.

```
    CONST minint = -32768;
    CONST maxint = 32767;
    TYPE int = INTEGER[minint..maxint];
    TYPE int1 = INTEGER[0..maxint];

    PROCEDURE var1 (VAR x : int) = PENDING;
    PROCEDURE var2 (VAR y : int1) = PENDING;

    PROCEDURE vare ( minv, maxv : int) =
    BEGIN
      VAR i1 : int1;
      VAR i2 : INTEGER[minv..maxv];
          .
          .
      var1 (i1#);
      var2 (i2*;
          .
          .
     END;
# Certain  "Varerror" site
* Possible "Varerror" site
```

Figure 4-3: "Varerror" examples

Note that "varerror" depends only on a potential type violation, not on an actual type violation, thus we are not concerned at all with the assignment operations within the routines "var1" and "var2". This concern for a potential violation contrasts with the semantics of "aliaserror" where the condition is not signaled unless two formal parameters really do have overlaping or identical objects as actuals[10]. Gypsy 2.1 eliminates dynamic subrange types making "varerror" completely determinable at parse time.

As noted above, all the signals associated with Gypsy operators and operations are of the predefined variety. None of these implicit signals can be explicitly handled unless the condition name is made known by a local condition or formal condition parameter declaration.

[10]Requiring a type violation within the called routine as a precondition for signaling "varerror" requires passage of the type of the actual as well as its location and greatly complicates the assignment operations

4.3 Condition Handlers

Gypsy provides both run time and proof time mechanisms for handling signaled conditions. The runtime mechanism is the "when part" of a statement. The "when part" appears at the end of a compound or control statement and is similar to a "case" statement whose selectors are conditions which might be raised within the statement to which the "when part" is attached. An "else" arm may be provided to handle any conditions not specifically handled by named arms. This allows handling of conditions whose names are not known in the environment in which they are raised. Figure 4-4 is a code fragment containing several statements with "when" parts. Since the statement in question may be a compound structure containing statements which have "when parts", a dynamic block structure for signals and conditions is established[11]. In the example, handlers for the condition, "singular" can be seen on both the BEGIN and IF statements.

```
...
BEGIN
  invert (matrix1) UNLESS (singular);
  multiply (matrix1, matrix2);
  IF diagonal (matrix1)
    THEN
      invert (matrix2) UNLESS (singular);
    ELSE
      SEND "Too bad" TO console;
  WHEN
    IS singular:
      SEND "Singular product matrix" TO console;
      SIGNAL escape;
    ELSE
      SEND "Unexpected error processing product" TO console;
      SIGNAL escape;
  END;
WHEN
  IS singular :
    SEND "Singular input matrix" TO console;
    SIGNAL routineerror;
  IS escape :
    SIGNAL routineerror;
  ELSE
    SEND "Unexpected error processing input" TO console;
    SIGNAL routineerror;
END;
...
```

Figure 4-4: Statements with "when" parts

When a condition is signaled, control is transferred to the most enclosing handler for that condition. This will be either a named "when arm" for the condition or an "else arm". After passing through the "when part", control passes to the statement following the one containing the "when part"; unless, of course, the "when" or "else arm" itself signaled a condition in which case control is passed to the most enclosing handler for that condition. In Figure 4-4 "singular" signaled from the call to "invert" within the IF statement causes control to pass to the "singular" arm of the IF statement's WHEN part. This arm ends with a SIGNAL statement which transfers control to the

[11]Since "when arms" can contain arbitrary statements, they too may contain "when parts". This does not alter the basically block structured scoping of the handlers.

"escape" arm of the BEGIN statement's WHEN part which in turn signals "routineerror".

If there is no handler for a signaled condition, one of two things will happen:

1. If the signaled condition has the same name as a formal condition parameter, the corresponding actual (possibly the implicitly injected formal name) will be signaled into the calling environment and the search for a handler will proceed from the call site.

2. If the signaled condition is either a declared local condition or an undeclared predefined or injected condition, it will be converted to "routineerror" and "routineerror" will be signaled into the calling environment and a handler for "routineerror" will be sought. If no such handler exists all the way back to the main routine, the program will terminate abnormally.

In programs containing multiple processes, the search for handlers can be blocked at COBEGIN statements since all arms of a COBEGIN must terminate before the conditions signaled by the various processes can be handled. A COBEGIN can signal multiple conditions as each process can return a condition. If this occurs, only an ELSE arm can handle the multiple conditions and it is not possible to determine from the conditions alone which tasks terminated abnormally or what conditions were signaled. Figure 4-5 shows a cobegin with a when part. Both arms of the cobegin have an explicit condition parameter, "fooerror" as well as the implicit parameter "routineerror". Table 4-3 shows where control passes for each combination of conditions returned by the arms of the COBEGIN.

```
COBEGIN
  proc1 (param1) UNLESS (fooerror);
  proc2 (param2) UNLESS (fooerror);
WHEN
  IS fooerror: .... ;
  IS routineerror: ....;
  ELSE .... ;
END;
...
```

Figure 4-5: A COBEGIN statement with WHEN part

Table 4-3: Handler sites for signals in Figure 4-5

		proc1 return	
	NORMAL	**fooerror**	**routineerror**
proc2 return			
NORMAL	...	fooerror	routineerror
fooerror	fooerror	else	else
routineerror	routineerror	else	else

4.4 Exit Specifications

It is possible to specify the results of a routine that returns a condition with the exit case construct. This is similar to the when construct except that the arms are specification expressions. The allowable selectors are:

1. "routineerror"

2. the explicit formal condition parameters

3. NORMAL indicating a non-condition return from the routine.

```
CONST maxint = 32767;
CONST minint = -32768;
TYPE int = INTEGER[minint..maxint]

FUNCTION ifact (i:int) : int UNLESS (toosmall, toobig) =
BEGIN
  ENTRY i GE 0 OTHERWISE toosmall
  EXIT (CASE
    IS NORMAL: result = factorial(i);
    IS toosmall: i < 0;
    IS toobig: factorial(i) > maxint;
    IS routineerror: FALSE);
  ...
END;
```

Figure 4-6: Exit Case Specifications

Figure 4-6 shows an example of an exit case specification. If the exit case construct does not cover the possible returns from the routine, omitted cases are assumed to have the specification TRUE. The non case exit specification is, in effect, a shorthand notation for an exit case construct with all cases except NORMAL given the specification TRUE and NORMAL given the exit specification. The lack of any exit specification covers all return cases with the specification TRUE.

Exit case constructs serve as proof time handlers for signaled condition parameters. Only the NORMAL arm can be validated i.e. can contain an OTHERWISE clause leading to a signal at run time. This means that it is possible for the NORMAL exit to signal any of the condition parameters but that unhandled condition parameters signaled within the routine body cannot be transformed by the exit case specification. The exit case arms serve to anchor the return paths from a routine and indicate what must be proven for each condition return from the routine.

4.5 Conditions and Verification Condition Generation

The purpose of verification condition generation is to produce a set of theorems about a program which are sufficient verify the program. One verification condition or VC is generated for each control path through a routine where a path runs from an assertion or entry specification to an assertion or exit specification. A VC is also generated for the entry specification of any routine called from the path and at any point where process blockage might occur due to buffer operations. Figure 4-7 shows a routine for the computation of factorial its VCs and the reasons for their generation. Any control mechanism offering a choice of paths through the code gives rise to multiple paths and multiple VCs. Signals give rise to paths leading from the source of the signal to its handler, if there is one; or to its real or implied "exit case" if there is not. Unhandled local conditions are converted to "routineerror" for exit purposes. In the example, exception paths are not traced because the default "routineerror" exit specification of TRUE is assumed and any VCs resulting from tracing paths to this exit would be trivial.

4.6 Condition Handling and Code Generation

When all the possible control paths arising from implicit signals are taken into account, it can be seen that executable Gypsy programs are dominated by signals and signal handlers. The fact that most of these paths will never be traversed is of little comfort since, in principle, any one might be. The presence of such a large number of possible control paths effectively thwarts many of the normal strategies for program analysis and code generation. If we take a naive view of code generation in the face of such control flows, we are left with several possibilities depending on constraints such as

```
SCOPE factor=
BEGIN

NAME maxint FROM implementation;
TYPE posint = INTEGER(0..maxint);

FUNCTION fact (n:posint) : posint =
  BEGIN
    EXIT  RESULT = factorial(n);
    VAR i:posint := 0;
    RESULT := 1;
    LOOP
      ASSERT RESULT = factorial(i);
      IF i = n THEN LEAVE END;
      i := i + 1;
      RESULT := RESULT * i
    END;
  END;

 FUNCTION factorial (m:INTEGER) :INTEGER =
   BEGIN
     ENTRY m GE 0;
     EXIT (ASSUME RESULT = IF m = 0 THEN 1
                                        ELSE m*factorial(m-1)
                           FI);
   END;
 END;  {SCOPE factor}

Verification condition FACT#1
    ASSERT (Continuing in LOOP)
       H1: RESULT = FACTORIAL (I)
       H2: I ne N
      -->
       C1: RESULT + I * RESULT = FACTORIAL (I + 1)

Verification condition FACT#2
    NORMAL exit specification for unit FACT::FACTOR
       H1: RESULT = FACTORIAL (I)
       H2: I = N
      -->
       C1: RESULT = FACTORIAL (N)

Verification condition FACT#3 has a True conclusion.
    FACTORIAL::FACTOR entry

Verification condition FACT#4
    ASSERT (Entering LOOP)
      1 = FACTORIAL (0)
```

Figure 4-7: Code for computing factorial and its VCs

program size and performance. When control reaches a condition handler, the program state must be the same as it was at the signaling site. Since, in general, there are many paths leading into each handler, it is unlikely that such strategies as temporary assignment of variables to registers, etc. can be used unless values are restored to their memory locations along each path to the handler. If space is no problem, execution along the normal paths can be made more efficient at the expense of additional run time and code along the signal paths. This code would assure that each handler has the appropriate program state when it is entered no matter what register assignments, etc. were in effect at the signal site. Note that in a paged machine, it may be possible to place the signal processing and handling code in one set of pages and the normal path code in another, reducing the working set size of the program significantly. If space is a problem, as it has been in Gypsy implementations to date, signal processing is not likely to be possible, and the implementation must preserve the appropriate program state at all times. In addition to assuring that variables in memory are kept updated to their current values at all times, we have the additional constraint that destructive or in-place modifications of variables are not permitted if such a modification could result in a signal. For example consider

```
    var   K:Integer;
    var   I:[1..10];
    .
    .
    .
    I := I + K;
```

Naively, on a two address machine such as a PDP-11, one might implement the addition as

```
    ADD   K,I        ; K+I -> I
    BCS   $ADDER     ; Branch if overflow ("adderror")
    BLE   $VALER     ; Branch if result < 0 ("valueerror")
    CMP   #10,I      ; Compare I to 10
    BLT   $VALER     ; Branch if result > 10 ("valueerror")
```

where $ADDER is the "Adderror" handler and $VALER is the "valueerror" handler. Unfortunately, Gypsy semantics require I to have its old value at entry to the handler and either the signals must be processed by undoing the addition along this path or the code must be rewritten as

```
    MOV   I,R0       ; I -> temp
    ADD   K,R0       ; K+I -> temp
    BCS   $ADDER     ; Branch if overflow ("adderror")
    BLE   $VALER     ; Branch if result < 0 ("valueerror")
    CMP   #10,R0     ; Compare I to 10
    BLT   $VALER     ; Branch if result > 10 ("valueerror")
    MOV   R0,I       ; temp (K+I) -> I safely
```

This requires seven instructions and eleven words as opposed to five instructions and nine words for the incorrect code. If the handlers are not within branch range, it may be necessary to reverse the sense of tests and add jump instructions at the additional cost of two words each. Note that if we did not need to do either check, the code for the statement would be

```
    ADD   K,I        ; K+I -> I
```

which requires a single instruction and at most three words when both operands are in memory.

The large numbers of control paths generated by implicit signals so dominate Gypsy code generation under our naive view that only relatively simple minded code generation techniques suffice for the working parts of the program. It is clear that reducing the number of such paths would allow more sophisticated code generation techniques and optimizations. Often local context or type information is sufficient to show that a given Gypsy operation will not raise exceptions. These cases will be covered here. Cases which require global information and/or the use of the proof system will be covered in the section on optimization below.

Dynamic semantic violations usually involve violations of Gypsy type restrictions. The most common are "valueerror" and "indexerror". "Valueerror" occurs when an attempt is made to assign to a variable a value outside the value set associated with its type. Clearly, if the types of the objects are identical, no error is possible. That this is the case can often be determined directly from the types of the objects involved. If the value being assigned is the result of an expression evaluation, this will not generally be the case although it may be possible to prove that the assignment is "valueerror" free.

Most Gypsy programs define an integer subrange type which has the same value set as "integers" on the hardware for which the program is written. All implementable scalar types are subranges or mappings to subranges of this type[12] which we shall refer to as "int". Since run time expression evaluation is performed in terms of "int" operations, expression results will be of this type and assignments to "int" objects free of "valueerror". The price paid is that of requiring checks for the implementation deficiency conditions which may be relatively inexpensive.

Implementation deficiency condition checks can often be suppressed by the use of suitably restricted types. Gypsy semantics define the type of the result of an operation in terms of the base type of its operands, where the base type has all range restrictions removed, i.e., if A and B are of type "[1..10]", the expression (A+B) is of type "integer". Clearly the value of A+B is in the type "[2..20]" and no "adderror" is possible on a machine where addition functions properly over the range $-32768 < A+B < 32767$. Extending this analysis to expressions of arbitrary complexity is straightforward but the resulting ranges can easily exceed the implementation range especially when multiplication or exponentiation are involved. In general, it appears that proof techniques making use of additional program state information are more likely to produce satisfactory results than simple interval calculations. In naively emitting either hard code or interpretive code as discussed below, either type of exception check is possible provided that the architecture makes the necessary information available. If Gypsy is translated into another high level language, it may be difficult or impossible to gain control of the code generation process in such a way as to detect implementation deficiency exceptions.

4.6.1 Threaded Code

The necessity for extensive checking in a naive implementation of Gypsy results in excessive code for individual operations with large program sizes a direct consequence. Because simple operations in Gypsy cannot be handled with single machine instructions, it may pay to rely heavily on subroutines for even simple operations. If the calling sequence is simplified, a net savings in space can be achieved at the expense of modest increases in run time. One such scheme is the use of threaded code. [Bell 73] In a threaded code scheme, all code is represented as a list of subroutine addresses. A pointer to the current entry in the list is maintained and each routine transfers control directly to it successor. Arguments are passed between routines via a stack. On the PDP-11, threaded code for our example might look like this:

```
$PUSHK  ; Put K on the stack
$PUSHI  ; Put I on the stack
$ADD    ; Add top two stack elements
        ; Leave result on stack top
        ; Transfer to "adderror" handler on overflow
$VAL10  ; Check stack top for [1..10]
        ; Transfer to "valueerror" handler if not
$POPI   ; Store stack top in I
```

The push and pop routines are created for each variable in the routine and would typically be of the form

```
$PUSHK:  MOV K, -(SP)    ; Push K onto the stack
         JMP @(R4)+      ; R4 is the "thread" linkage
```

for scalar items requiring a single word. These routines are reusable, i.e., only one copy of this code exists no matter how many loads of K are required. $ADD adds the top two items on the stack leaving the result there or signaling "adderror" if one occurs. This signal could take the form of a call

[12]Is is, of course, possible to implement any sort of arbitrary precision arithmetic for Gypsy, but we restrict ourselves to implementations based on hardware integer arithmetic for the purposes of this investigation

to a routine that would determine the location in the thread of the handler for "adderror" and transfer control there, adjusting the argument stack as necessary. Only one copy of this routine is necessary for all adds in the program. $VAL10 is a value error checking routine for the type [1..10]. It will be called any time value error checking of such a type is required. In the example above, the thread occupies five words. Each push and pop routine is three words. If we assume that the signaling is done with a general purpose routine, the add can be done in six words (Figure 4-8) and the value error check in nine words (Figure 4-9).

```
$ADD:
        ADD(SP)+,(SP)      ; Add top two stack elements
                           ; Leave result on stack top
        BCS $ADD1          ; Branch on overflow
        JMP @(R4)+        ; Thread link
$ADD1:  JSR PC,$SIGNL      ; Call signal handler
        $ADDER             ;  with "adderror" parameter
```

Figure 4-8: Threaded Code for Addition of two INTs

```
$VAL10:
        MOV (SP),R0        ; value -> temp
        BLE 1$             ; Branch if value le 0
        CMP #10.,R0        ; See if value > 10
        BLT 1$             ;  and branch if it is
        JMP @(R4)+        ; Value in [1..10]
                           ;  so take thread exit
1$:     JSR PC,$SIGNL      ; Call signal handler
        $VALER             ;  with "valueerror" parameter
```

Figure 4-9: Threaded Code for Value Error Check of Type [0..10]

This results in a total of thirty words of thread and code for the statement, however all the routines are reusable and a second instance of the same statement would only cost the five additional words of thread. The break even point for threaded code is not immediately obvious. For routines which are only a few words long, the threading overhead is substantial. For routines performing complex operations such as copying complicated structures, out of line code offers large space savings at minor expense.

In both the inline and threaded cases, signals are processed by transferring control to the handler for the signaled condition. In the inline case, we have assumed that the handler is known and directly or indirectly reachable. Assuring this is simply a matter of bookkeeping during code generation and/or run time. With threaded code, signals arise within the operational routines and the location of the handler is not directly known. One reasonable solution to this problem is to link the handlers within the thread so that the signal mechanism can search for a matching handler. Noting that handlers are relatively rare (except that there is an implied handler for the "routineerror" return mechanism in every routine) we can afford some thread space overhead for those which do appear. It appears that about four words will suffice for each condition handled with a like amount for each WHEN part and the usual condition return with no explicit formal condition parameters.

4.6.2 Translation of Gypsy to BLISS

An aside in [Loveman 77] suggests translating high level languages into systems implementation languages such as BLISS (*ibid*, page 137) to take advantage of their low level optimizations. BLISS [Wulf 71] is a family of "high level" systems programming languages. Implemented first for the PDP-10 and then, with considerable variants, for the PDP-11, it has recently been standardized into a language called Common BLISS with (more or less) compatible dialects for the PDP-11, VAX, and

PDP-10 machines referred to as BLISS-16, BLISS-32 and BLISS-36 respectively. The BLISS translators have a reputation for producing high quality machine code.

After an earlier unsatisfactory effort to translate the Gypsy internal form directly into assembly code for the PDP-11, it was decided to attempt translation into BLISS. It was hoped that an efficient Gypsy implementation could be achieved with a minimal effort. Using Common BLISS, VAX and PDP-10 implementations should also be possible with little additional effort.

Gypsy conditions cause problems in BLISS. In some cases, it is difficult or impossible to detect a Gypsy exception in the corresponding BLISS code. Implementing an efficient transfer of control to the handler is also difficult in BLISS.

The detection of Gypsy dynamic semantic violations in BLISS offers no particular problems. Just as it is possible to recast implicit checks and signals for conditions such as "indexerror" and "valueerror" in Gypsy into explicit checks and signals, it is possible to do the same in BLISS. Thus the Gypsy

```
    VAR  A : [1..10];
    VAR  B : [5..15];
           :
           :
    A := B  ;
```

becomes

```
    IF B < 11
       THEN
           A := B
       ELSE
           SIGNAL VALUEERROR
       END;
```

when the test and signal are made explicit. Similarly, in BLISS, the assignment could be represented as:

```
    A = (IF .B < 11
          THEN
           .B
          ELSE
           % CODE TO SIGNAL "VALUEERROR" % );
```

Implementation deficiency conditions are another matter. BLISS operations have the same semantics as the underlying machine instructions and no checks for overflow, etc. are available in the language. Common BLISS makes no provision for the insertion of machine code and access to the PDP-11 condition codes is not possible in BLISS-16. The inability to detect implementation deficiencies in BLISS is compounded by BLISS optimizations which reorder code and often change the operations actually performed. For example,

```
    I := I + 1
```

will be implemented on the PDP-11 as

```
    INC I    ; I+1 -> I
```

which does not set a condition code bit even if the result overflows. Under these circumstances, the only safe way to implement Gypsy operators that have associated implementation deficiencies is via hand coded machine language functions with explicit formal condition parameters[13]. As this will probably thwart most of BLISS's optimizing capability, much of the potential benefit of the Gypsy to BLISS translation will be lost. It must be emphasized here that unless it is known at compile time that no implementation deficiency is possible from a given instance of a Gypsy operator, it is not

[13]In many cases, it is possible to generate a Euclid style leagalty assertion to guard the operation. While it is possible to create such guards in such a way as to avoid exceptions within the guard, there is no way to ensure that the corresponding code generated by BLISS will not be reordered to produce exceptions.

permissible to translate it into the corresponding BLISS operator. Even the optimization condition approach outlined in the next chapter cannot prevent BLISS from so altering the order of evaluation of expressions that Gypsy operations proved safe are converted into exception causing code.

The Gypsy signal is a direct transfer of control from within a structure to an appendage of that structure which would not normally be executed. After execution of the appendage, control passes to the normal successor of the exited structure. BLISS is a "Go to" free language and does not permit arbitrary transfer of control. None of the Common BLISS control structures exactly match the Gypsy SIGNAL..., WHEN IS... pair.

Common BLISS has an enable-signal construct which is quite unlike the Gypsy mechanisms. This mechanism is not suitable for implementing Gypsy signals. Common BLISS does not make any provision for abnormal returns from routines so the signaling of local and parameter conditions must be treated separately. At the present time, insufficient experience with Gypsy makes it difficult to decide which of the many possible BLISS implementations of Gypsy signals is optimum. The inability to deal directly with machine deficiency errors in common BLISS may defeat many of its optimizations, a primary reason for its consideration.

4.7 Conditions and Optimization

Conditions affect optimization in two ways. As we have noted before, checking for situations which could give rise to signals can take a significant amount of time and code space. Most of these checks are unnecessary as the circumstances resulting in the signal will never occur. A significant savings in time and space can be achieved by identifying and eliminating the unnecessary checks. Most of the experimental results reported in chapter 9 involve the elimination of unnecessary checking code. A secondary effect of showing that a potential exception site will not raise an exception is a reduction in the number of possible control paths of the program. Traditional optimizations depend on being able to rearrange the code written by the programmer producing a more efficient but equivalent computation. In general, the equivalence is required only for successful computations, i.e. those that do not signal exception conditions from either the optimized or unoptimized code. Languages such as FORTRAN leave the state of the computation undefined when an exception occurs. From a practical standpoint, the situations where either the optimized or unoptimized versions of a code raise an exception are ignored.

The fact that the Gypsy programmer can chose to handle exceptions constrains the types of reorganization that can be performed. This is because control must arrive at the condition handler with a program state consistent with that expected by the handler. For conditions signaled and handled within a given routine, this means that:

1. Code causing a signal cannot be moved past code which alters a component of the program state which is used in the handler for the signal or in any path from the exit of the handler to the routine exit. Thus in figure 4-10, the common expressions in the arms of the CASE statement, namely the address calculation for "updated_master[new_record]", cannot be moved outside the CASE because the handler for "indexerror" uses the values of "new_record" which may be different in each arm.

2. Code not causing a signal cannot be moved past code causing a signal if it alters a component of the program state which is used in the handler for the signal or in any path from the handler exit to the routine exit.

For signals converted into "routineerror" at the routine boundary or condition parameter signals not handled within the routine, we can assume that only code affecting var parameters of the routine need be considered as contributing to the expected state at the handler. A first approximation of the handler associated with the "routinerror" exit from a routine assumes that the path from the

```
read_transaction(input_buffer, operation, new_record);
CASE operation
  is enter:
     normalize (new_record);
     updated_master[next_record] := new_record;
  is update:
     update_record(new_record , old_master);
     updated_master[next_record] := new_record;
  is create:
     updated_master[next_record] := dummy_record
                                    with .name := new_record.name;
  when
    is indexerror:
       log_error(log, operation, next_record, new_record);
end;
```

Figure 4-10: Code movement blocked by handler state constraints

"routinerror" return in the calling environment makes use of all the variable parameters to the routine. If global analysis of the actual parameters matching "routineerror" shows that this condition is not handled along any program execution path calling the routine, then code motion within the routine need not be constrained by "routineerror" path considerations.

As we have noted earlier, signals of predefined conditions are seldom issued and handlers for them are relatively uncommon. This leads us to suspect that most potential signal sites will never emit signals. If we can prove this at a given site, we can omit the condition checking and signaling code at the site. If we are successful in producing large regions of signal free code, substantial code reorganization may be possible in spite of the constraints noted above. Thus, showing the absence of signals of predefined conditions appears to be the key to optimization.

The first task confronting the optimization effort is to show that most potential signal sites will not signal. To do this, the GVE verification condition generator was modified as discussed in chapter 5. This modification allows the verification condition generator to produce an additional set of theorems which will be referred to as optimization conditions or OCs.

This phase of processing may give rise to a large number of OCs. Many of these occur at "valueerror" sites, and, in strongly typed programs, can be shown to be trivially true at the time they are generated. Nontrivially true OCs require proof using the theorem prover in the verification system. Most of these proofs appear to be straightforward but depend more on language related information than do most correctness proofs.

Conditions arising from implementation deficiencies are difficult to deal with. While their sources can be guarded in the same manner as the sites producing dynamic semantic error conditions, this is not appropriate unless extremely straightforward code generation techniques are contemplated. We should like to be able to reorder expressions and factor out common subexpressions in the process of code generation. Even when we have shown that it is possible to move a subexpression such as an address or index calculation, we must consider its evaluation order as a separate issue. The OCs we generate concerning expression evaluation are based on the order in which the expression appears in the prefix or internal representation of the program. For example, figure 4-11 shows an expression, its prefix representation, and the conclusions of the OCs which must be proved if the expression is to be coded without exception checking. Even if the OCs are proved, we know nothing about exceptions which might be raised with other orders of evaluation.

Basically, we have two choices:

- Expression:
```
      a * b + (b - 5) * c
```
- Prefix:
```
      (PLUS (TIMES A B) (TIMES (MINUS B 5) C))
```
- OC Conclusions:
```
      not badmultiply(a, b)
      not badsubtract(b, 5)
      not badmultiply(b-5, c)
      not badadd(a*b, (b-5)*c)
```

Figure 4-11: Optimization Conditions for an Expression

1. We can define a function say "CAN_EVALUATE(exp)" which will be true only if all orders of evaluation of "exp" are safe and attempt to prove that the all paths leading to the evaluation of "exp" imply the truth of CAN_EVALUATE(exp). In considering this approach we will ignore the fact that such a function is not easily expressable in the GVE.

2. We can defer OC generation for implementation deficiencies to code generation time and combine the generation of the necessary OCs with the generation of the code. This requires close interaction between the code generation and proof processes as the failure to prove a given OC will affect the code being generated.

The first choice preserves the independence of the optimization and code generation efforts but has several drawbacks. The first is that the proof of CAN_EVALUATE(exp) may in general be quite difficult. Failure to prove this property merely indicates that some order of evaluation of "exp" may fail. Unless we can use this information to identify unsafe evaluations and assure that the code generator will not use an unsafe evaluation we must evaluate the expression using guarded instructions even though the particular evaluation used by the code generator does not require this. In addition, proving CAN_EVALUATE requires extensive knowledge of what the code generator may do. For example, consider the following situation:
```
CONST maxint = 32767;
CONST minint = -32768;

TYPE int = INTEGER[minint...maxint];
TYPE medint = int[minint/3+1...maxint/3-1];
            .
            .
  VAR a,b,c:  medint;
  VAR d,e,f:  int;

  d := a + f;
  e := c-f;
  .
  ... a + b + c
  c := 0;
  .
```
where "int" represents the hardware integer type. Clearly CAN_EVALUATE(a+b+c) is TRUE since no order of evaluation or assignment of values to the variables a, b, and c can produce a result outside the "int" or hardware range. However, a clever code generator, noting that it has in its registers, for example
```
Ri = a + f
Rj = b - f
Rk = c
```

and knowing that c is dead may well emit code such as

```
      Rk <- Rk + Rj
      Rk <- Rk + Ri
```

about which we have proven nothing. This computation will overflow given that either

```
      (a+f) + c  > maxint
```

or

```
      (a+f) + c  < minint
```

Under these circumstances, the proof of CAN_EVALUATE(exp) must include all evaluations of "exp" that might be emitted by the code generator. In practice this requires mimicking the behavior of the code generator and would involve an excessive amount of effort.

If we view optimization and code generation as a continuous process transforming a high level machine independent representation of a program into hard code through a series of transformations and decompositions which successively introduce more machine dependence into the representation as the higher level constructs are decomposed, we can introduce the OCs for the suppression of hardware exception tests at the point where hardware instruction sequences are introduced into the representation. To be practical, we must be able to appeal to a prover for proof of these OCs at the time they are emitted since the code sequence to be emitted depends on whether or not the OC can be proved. Viewing the transformation as a continuous process, we wish to avoid the branching that would occur if every OC resulted in two versions of the representation, one with and one without checking code. A code generation process using this technique is described in chapter 10.

4.7.1 Relaxing Condition Semantics

Because verifiable programs are bound by rigidly defined semantics even when exceptions occur, and because exceptions can arise anywhere, the Gypsy implementor has relatively little freedom to perform the classical sort of code reordering optimizations. As we have seen above, showing that most possible exception sites will not emit signals is a key to optimization. If it is not possible to prove a routine to be condition free, we may not be able to perform significant optimizations on it. Often the signals of which we are unable to show the absence are not handled and are allowed to signal "routineerror" at the call site of the routine in which they occur. In many cases, such signals are used to allow the user to distinguish between error and normal termination of the program and the state of the program when the signal is given is not material. In other cases, recovery is attempted and preservation of the program state expected at the handler is necessary.

A method of relaxing the rigid semantics as a form of negotiation between the programmer and the optimizer could be provided. It may be argued that the exit specifications of the routine serve this purpose and that the only obligation of the optimizer is to preserve the appropriate EXIT CASE arms in the event of condition returns. Given that partial specifications and mixtures of specified and unspecified code are common practice, this is probably too weak a constraint to be useful. With the possible exception of signals which always pass directly to an abnormal termination of the entire program, some care needs to be taken to assure that a the exit state of the variable parameters of a routine is such as to assure that the computations following the return from the routine will be correct.

A combination of the global actual/formal parameter analysis discussed in chapter 6 with conventional data flow analysis along the exception paths from every call site of the routine could provide the constraints necessary. As an alternative, optimization directives or "pragmas" could be added to a language to achieve this result. For example each WHEN arm and EXIT CASE arm could be annotated with a PRESERVE clause of the form

```
      PRESERVE  v1, v2, ... vm;
```

where each vi is an optimization time determinable item identifier such as a variable name, record field selector, subarray, etc. Defaults of

```
      PRESERVE ALL;
```

or

```
    PRESERVE NONE;
```

could be established for each routine or for a whole program by interaction with the top level of the optimizer.

The scope of a PRESERVE clause would extend to all code that could signal its containing WHEN arm or EXIT CASE arm, thus the effects for nested handlers for the same condition would be cumulative and most restrictive for the innermost code. The appearance of an identifier on a PRESERVE list would restrict code motion so that code that might signal the associated condition could not move past code that alters a preserved item.

4.8 Architectural Support For Exception Checking

Much of this chapter has been devoted to discussions of how to avoid the emission of the somewhat lengthy code sequences required to implement exception checking and condition signaling. It is reasonable to ask if it might be possible to perform such checks in the hardware of the machine on which the language is implemented. Some advantage would be obtained with such an approach. If checks could be performed in parallel with the primary computation, i.e., "valueerror" checks made in parallel with the address calculations for a store instruction used as part of an assignment operation, a savings in both time and space would result. In addition, proof time effort would be minimized since the run time checks would incur little or no penalty. Many architectures provide parallel checks and signals for implementation deficiencies. In these architectures, an overflow causes an interrupt and transfer of control to an appropriate handler. Unlike condition code machines such as the PDP-11 used in the code examples above, checking for an exception on such a machine incurrs no expense although its handling may still be quite expensive.

Unfortunately, inexpensive checking for exceptions is not in itself sufficient to solve all the problems associated with exceptions. As we have seen, the presence of potential signal paths is the main bar to effective optimization. Efficient run time checks do nothing to remove this barrier. Although the makers of one high level architecture claim hat they can achieve effective performance without optimization [INTEL 81] there is substantial evidence in the case of code generation for stack machines that optimization can provide significant benefits [Carter 74]. Were this to be the case for a high level condition checking architecture, the proof time work to permit optimization would still be necessary.

Note that such an architecture would provide an additional degree of fault detection as it would be reasonable to check for even those errors which had been proven not to exist. In the unlikely event that such an error was detected, it is doubtful that meaningful handling for recovery could be provided, however valuable *post mortem* information could be obtained.

4.9 Summary

In this chapter, we have attempted to deal with signals and conditions in Gypsy and to some extent in similar languages such as Ada. The impact of conditions on various components of the verification environment has been considered in detail. As potential conditions appear to dominate verifiable programs, the potential effects on run time program images have been treated in detail. We note that most potential signal sites never emit a signal and conjecture that proving this is a prerequisite for the generation of efficient code for Gypsy and similar languages.

Conditions associated with implementation deficiencies or hardware errors appear to be the most difficult to deal with since their issuance and detection is intimately associated with the emission of code sequences for a particular architecture. If the implementation is allowed to evaluate expressions in an arbitrary fashion either exception checking code must be included in the emitted sequences, or proof that it is unnecessary must accompany code generation.

More attention needs to be given to condition signaling and handling in verifiable languages. If implicit signals are to be used for flow control or for recovery from run time faults, the paths of concern need to be explicitly identified if for no other reason than to make verification more tractable. Whether meaningful recovery from implementation deficiencies manifesting themselves as hardware errors is possible or useful deserves further study. It may be necessary to severely constrain code generation within the scope of a handler for such a signal in order to assure a meaningful program state at the handler. The merits of such an approach need further evaluation.

Chapter 5
AN IMPLEMENTATION OF EXCEPTION SUPPRESSION

The verification condition generator (VCG) of the Gypsy Verification Environment (GVE) has been modified to issue Optimization Conditions (OCs) which, if proved, are sufficient to show that a given Gypsy operator will not signal its exception condition(s) at run time. Elimination of exception checking code can be predicated on the proof of the appropriate OCs. This chapter describes the mechanisms provided for this purpose.

5.1 Extended Routine Bodies

The Gypsy parser produces an intermediate representation of a Gypsy routine, known as prefix, which is similar in structure to the Gypsy text supplied by the user. This form leaves much to be desired for code and VC generation purposes as many aspects of the language are not explicitly represented . These include

- initialization of local variables,
- binding of initial values of var parameters,
- the relationship between concrete and abstract specifications,
- the flow of control through exit case specifications and the associated condition return mechanism,
- allocation and deallocation of local storage.

In addition, the usual prefix representation of a routine does not provide a coordinate system making it impossible to refer to a particular instance of an operator in the prefix.

To simplify VC and code generation, and to permit communication between the two functions, an extended routine body has been defined. This consists of a single compound statement which accurately characterizes both the VCG and runtime behavior of the routine. Figure 5-1 shows the most general form of the extended body. The allocation and/or concrete specification levels of the structure may be omitted if the routine has no local symbols or concrete specifications.

Statements and expressions in the extended body are annotated to provide a coordinate mechanism. Each statement is given a statement number and each expression within a statement is given an operator number. This, combined with positional information for function and routine arguments, record field identifications, and condition names allows each exception site to be uniquely identified.

```
(BEGIN
   <keep off stmt>
   (BEGIN
      <primed var param bindings>
      <assertions from entry specs>
      <assumption of 'ed holdspec>
      <assertions from centry specs>
      (BEGIN
         <allocate locals>
         <initialize locals>
         <keep on stmt>
         <routine body>
         <deallocate locals>
       WHEN
         <is condparam:
                <deallocate locals>
                <signal condparam>>
         <else: <deallocate locals> <signal routineerror>>
       END)
       <assertions from normal cexit specs>
       <assumption of 'ed holdspec and 'ed centry>
    WHEN
      <is condparam:
             <assertions from condparam cexit specs>
             <assumption of 'ed holdspec and 'ed centry>
             <signal condparam>>
      <else: <assertions from routineerror cexit specs>
             <assumption of 'ed holdspec and 'ed centry>
             <signal routineerror>>
    END)
   <assertion of normal exit spec and holdspec>
   <return normal>
  WHEN
   <is condparam:
          <assertion of condparam exit spec and holdspec>
          <return condparam>>
   <else: <assertion of routineerror exit spec and holdspec>
          <return routineerror>
  END)
```

Figure 5-1: Extended Routine Body for Gypsy

5.2 Operator Definition Functions

[Good 78] provides functional definitions for most of the Gypsy operators. Most of the definitions given contain too much informal verbiage to be used to prove anything about the operators in a Gypsy program. Many of the definitions are also parameterized on type, a feature unsupported by Gypsy.

It would have been possible to encode the specifications of the operator definition functions into the VC generator, but this was rejected for several reasons

- a large amount of special case code would have been required,

- changing the machine dependent aspects of the specifications would have been complicated,
- it would not have been possible to use the full expansion and instantiation capabilities of the prover,
- experimentation with different formulations of the operator specifications would have been difficult.

After considerable experimentation, an almost suitable formulation of the operator definition functions was developed. These appear in appendix A. These are **almost** acceptable because a few of the functions require type names as parameters. Some predefined Gypsy functions have type names as parameters, but user defined routines are not permitted this feature. The definition routines are written so as to be acceptable to the Gypsy parser and the resulting database is edited to provide proper instantiation of the required type name parameters.

5.3 Valueerror

Even with the type name parameter kludge described in the previous section, it is not possible to formulate a single general assertion, acceptable to the Gypsy expression evaluator and prover which captures

"<EXP> IN THE_VALUESET_OF_TYPE_T"

for an arbitrary type T. This means that we cannot specify a value checking function in the same way we could an index checking one[14] .

Fortunately, for any given type, we can develop an expression, dependent on the definition of the type which will serve to define the "valueerror" condition for a given assignment or parameter passage.

5.4 Generation of OCs

When the VC generator encounters an operation from which an exception might arise, it must trace both the normal path and the exception path[15] from the operation. If we are generating optimization conditions, it must also generate a theorem (the OC) which asserts that the circumstances which would cause the exception path to be taken cannot occur. We associate the coordinates of the location in the code with which it is associated with each OC. In general, the coordinate is a five-tuple of the form:

```
(<STATEMENT> <OPERATOR> <ARGUMENT> <FIELD> <CONDITION>)
```

where:

- <STATEMENT> is the number of the statement involved
- <OPERATOR> is the number of the operator. It will be NIL if the OC is for "valueerror" on an assignment or procedure call.
- <ARGUMENT> is NIL except for procedure or function calls in which case it indicates

[14]With index operations, we know that the index type is a scalar with a defined upper and lower bound which are accessible via the Gypsy UPPER and LOWER functions.

[15]To avoid the effort of tracing paths which lead to a trivial assertion of TRUE, we note whether a handler (or exit case) for the condition exists and suppress the path if there is none.

the position in the data argument list for a "valueerror" or "varerror" OC.

- <FIELD> as above, identifying a record field in a structured argument.
- <CONDITION> the name of the condition for which check code can be suppressed if the OC is proved.

The OCs are similar to ordinary VCs. They are implications of the form:

```
    <INITIAL_PATH_ASSUMPTION>
  & <CUMULATIVE_STATE_ALONG_PATH>

  =>
     NOT <EXIT_CASE_OF_OPERATOR_FUNCTION>
```

for operators defined with functions or

```
    <INITIAL_PATH_ASSUMPTION>
  & <CUMULATIVE_STATE_ALONG_PATH>

 =>
    <EXP> IN <VALUE SET>
```

for "valueerror" OCs.

In the case of operators, the operator name is associated with its definition function via a table, and the VC generator evaluates the corresponding function call in its usual fashion. If OCs are being generated, the abnormal exit specifications of the function are instantiated using the normal GVE mechanisms. For "valueerrors" the conclusion of the OC is produced from symbol table information associated with the type of the receiving object.

In both cases, the conclusion is evaluated to see if it is TRUE independently of hypotheses. Such OCs are called UNCONDITIONALLY TRUE and, as we shall see, permit unconditional elimination of checking code.

5.5 System Interface

Support for optimization conditions has been provided in the top or executive level of the GVE. The VC generator is told to generate OCs in one of two ways:

1. The optimizer command can cause the generation of OCs and VCs for an entire program;
2. A flag can be set to cause the generation of OCs whenever an executable function or procedure is given to the VC generator.

The user can examine, prove, determine the proof status of, and the reasons for the generation of OCs in the same way as VCs. It is also possible to obtain detailed summaries of OC status via an analysis package. Samples of these summaries are shown in Chapter 9.

An internal interface between OCs and code generation is also available. A Lisp function, TOP:OPTIMIZE-OPERATION?, allows the code generator to determine the proof status for a given location in a specified routine. This routine computes a function of the proof status of the OC associated with the location and the optimization confidence level. The optimization confidence level is set by the user and allows the user to force exception checking at all sites, suppress it at all sites or make suppression depend on some combination of the proof status of the OC and the program in which it appears. The relationship between the OC proof status and the proof status of the program is discussed in detail below.

5.6 Proof of OCs

Proof of OCs is no different from proof of VCs. OCs depend on proof VCs however in a way which may not be immediately obvious. If we consider the path tracing which goes on during VC and OC generation, we note that each path starts with an assumption and ends with a conclusion. The assumption is either an entry specification or user supplied path breaking assertion. In the former case, proof of the entire program requires that the entry specification be proved at each site where the routine in question is called. In the latter case, the assertion will be the conclusion of one or more VCs resulting from paths leading to the assertion and, again, proof of the entire program requires these to be proved. Proving the entry and path breaking assertions justifies their assumption in other paths and their inclusion as hypotheses in the VCs and OCs of those paths.

Now a program is either proved or it is not[16] and not much useful can be said about the correctness of a program for which only a portion of the VCs have been proven. In the case of OCs we may want to do as much optimization as is safe without doing the correctness proofs for the entire program.

How much can we accomplish without doing the correctness proofs? The answer is that we can often do a great deal. If we could analyze our OC proofs to determine whether the proofs depend on user supplied specifications or simply on type information and program states, we could optimize the latter cases without consideration of program verification status.

Noting that run time validated specifications assure the truth of the assumed specification on paths from them, we can further expand the set of trustable OCs. Finally, we note that many routines use a default entry specification of TRUE which is automatically satisfied at all call sites reducing the correctness of user assertions within the routine to a local problem in the same way as validated entry specifications.

Tracking the sources and uses of hypotheses through the proof process is not practical in the Gypsy proof system. We do find that a large percentage of the optimization conditions generated have TRUE conclusions and are thus independent of any hypotheses[17] . In a number of other cases, OC conclusions reduce to TRUE when they are evaluated in a state which uses the values of global constants from the program. In the implementation, we provide the following criteria for determining whether to permit omission of exception code:

- NEVER - force exception checking of all operations
- TRANSLATOR - Allow the translator to eliminate exception checking in those cases in which it can determine that exceptions will not occur[18].
- SAFE - force exception checking of all operations except those whose OCs have true conclusions
- PROGRAM - same as "SAFE" if the program is not completely proved. If it is proved, suppress checking for all operations with proven OCs
- UNIT - if the unit being processed is proved and has a TRUE or validated entry specification, suppress checking for all operations with proven OCs.
- OC - suppress checking for any operation whose OCs are proved

[16]We can consider a verified package for use in an unverified environment as equivalent to a verified program if all its entry points are suitably guarded with runtime checked or trivially true entry specifications.

[17]"Valueerror" OCs for proper type containment have TRUE conclusions.

[18]This is a subset of the exception sites which are detected as unconditionally safe during the OC generation process.

- ALL - suppress all exception checking.

The first five criteria are "sound" in the sense that they should not suppress exception checking in any case where it could arise assuming that the code generated correctly follows the language semantics and that the proof are valid. Depending on the OC proof status alone, is unsound, but might be used to determine the potential program size under the assumption that the program could be proven. Similarly, suppressing all exception checks can demonstrate the maximum potential savings from these techniques but may result in incorrect code for programs which depend on exceptions for flow control.

Chapter 6
CALL OVERHEAD AND PARAMETER GLOBALIZATION

6.1 The Routine Problem

Procedure and function calls are the most frequent operations in most Gypsy programs. This is especially true of programs which are intended to be verified. The primary reason for using routine calls so heavily is to reduce the path explosion which occurs when verification conditions are generated. As noted in the overview of verification above, a verification condition is generated for each possible control path between pairs of assertions. As a practical matter, we start each path assuming only its initial assertion, and attempt to prove its ending assertion using the operations along the path. Each branching operation in the code between assertions causes a path split. Sequential splits have a multiplicative effect. For example:

```
ASSERT A1
IF P1
  THEN
      I1;
  ELSE
      I2;
  END;
...
PROVE A2
```

gives rise to two paths i.e.

```
PATH1: ASSUME A1
       ASSUME P1
       I1
       ...
       PROVE A2
PATH2: ASSUME A1
       ASSUME NOT P1
       I2
       ...
       PROVE A2
```

similarly a CASE statement such as

```
ASSERT A1;
...
CASE T
  IS 1: C1;
  IS 2: C2;
  IS 3: C3;
  IS 4: C4;
  ELSE: CE
ASSERT A2;
```

gives rise to a five path split in a similar fashion. If we combine the two statements as in

```
ASSERT A1
IF ...
CASE ...
ASSERT A2
```

we obtain ten paths. It is not uncommon for routines containing fifty or so executable statements in a suitably complex control structure to generate hundreds of paths. Each path gives rise to one or more verification conditions to be proved. Normally, we don't try to verify code in this form, but rather attempt to reduce its complexity by restructuring the program.

There are two ways to reduce the path explosion. The first is to add additional assertions to the code. For example, adding an assertion, say A12, between the IF and CASE statements above would reduce the number of paths to seven. The only drawback is that this extra assertion must be very carefully constructed as it must contain both the portions of A1 which are still true after the IF and which are necessary to prove A2 through the CASE as well as characterize the actions of the IF necessary to prove A2. This is because we treat paths independently and ignore the state information leading up to the entry assertion in previous paths for which it was the exit assertion.[19]

The second technique, and one which is most commonly used in practice, is to control the path explosion by encapsulating small blocks of code into procedures. This allows control of the path explosion while allowing the effect of the code to be inserted in the calling path by assuming the exit specification of the procedure.

Division of the code into numerous small procedures makes verification tractable, but it imposes a significant burden on program implementation. Routine calls affect program performance in two ways. Firstly, the time and space required to construct the calling sequence is usually significant. Secondly, the discontinuity in the control path introduced by the call can disrupt register allocations and block or hide possible optimizations such as common expression identification.

In this chapter, we describe techniques for reducing the calling overhead, in chapter 7, we discuss elimination of calls altogether.

6.2 The Introduction of Global Variables

Proof rules for global variables are difficult, but not impossible to work with [Boyer 80]. Gypsy has excluded them from the language in order to simplify the proof process. At the same time, analysis of Gypsy programs indicates that, even for routines called from numerous sites, the set of actual parameters corresponding to a given formal of a given routine often consists of a single object when the actuals are traced back through a formal/actual chain to their ultimate declaration sites. Furthermore, such objects are often either parameters to the main routine of a program or are local variables within the main routine. In a way, this is not unexpected, since Gypsy programs communicate with their environment through the parameters of the main routine which are often communications buffers, and passing these all over the program is a reasonable way to proceed. Similarly, the main routine is a suitable home for data objects which must be available to a number of lower level routines within the program.

Although syntactic access to such objects is via the parameter passage mechanism, the same effect can be achieved by substituting global access in the implementation. This can be done under the following constraints:

1. The semantics of the parameter passage mechanisms must require that the results of partial computations, i.e. exception returns, be reflected in the return state of the var actuals. Call by reference as used in Gypsy satisfies this criteria.

[19] In theory, we are entitled to assume the intersection of the state information obtained at the end of each path terminated by the assertion; in practice, we settle for the assertion.

2. If type mismatches between formal and actual parameters exist, the necessary checks to assure that the value of the actual is compatible with the type of the formal must be made at the call state even though no actual is passed.

3. Within routines using global accesses, all type and exception checks must be made under the assumption that the global object has the type of the formal it replaces if this is different from the type of the global object.

6.3 Implementation

Finding the homes of the actuals corresponding to the formal parameters of a unit is done in two passes over the database for a program. The first pass starts with the main routine of the program and examines each call site in the call tree reachable from the main program. At each site, the actuals are recorded in the called routine, along with the location of the calling site. The actuals for the main routine are considered to be locals in higher level routine, *IMPLEMENTATION , which invokes the main routine. The result of this pass is a list for each formal parameter of each routine which contains the immediate actuals for each call on the routine. For local objects and parameters to the calling routine, an entry gives the name of the actual within that routine. Literals, global constraints and expressions are identified as such. Each expression is given a unique identifier and no attempt is made to combine separate instances of the same expression as considerable data flow and state analysis would be necessary to ensure that syntactically identical expressions represented the same object. Note that a conventional common expression optimization used prior to this analysis would subsume the problem.

Actual condition parameters are identified by condition name and handler location where the location is the statement number of the statement containing a when arm for the condition. By convention, statement number zero is used for conditions handled by a condition return from the calling routine, i.e. its formal condition parameters.

Pass 2 tracks the immediate condition parameters back to their "home" sites. This is done by examining the list created for each formal parameter in pass 1 and finding those items which are parameters to a calling routine. In this case, the actual list for that parameter is substituted for the parameter in the list being processed and the process is repeated until the list consists solely of parameters to the main routine, locals of some routine, global constants, literals and expressions. If the parameter being processed appears in the list, as a result of recursion, it is eliminated as it must be grounded elsewhere.

If the list of actuals for a given formal contains only a single entry which is a local in some routine or a parameter to the main routine, a potential global has been identified, and a unique name is associated with the actual for use as its global name.

6.4 Analysis

To permit evaluation of the globalization analysis, a report generator has been provided. This requires a detailed routine by routine description of the results of the analysis as well as summary and statistical reports by routine and for the entire program.

From an application standpoint, the maximum benefits for the least effort can be obtained by using global accesses for reference to parameters whose actuals are parameters to the main routine or locals within the main routine. Allocating such variables statically incurs no space overhead since the main routine is always active and saves time since they may be initialized at compile rather than run time. The analysis phase identifies these cases as well as the more general ones. The results of parameter globalization for several programs are discussed in detail in chapter 9.

6.5 Potential Program Improvement

The exact savings that can be obtained by globalization are highly dependent on the implementation of the calling mechanism. The current Gypsy implementation for the PDP-11 uses a BLISS calling sequence which pushes the address of each argument onto a stack. This consists of one instruction of one or two words per argument. In addition, additional instructions may be necessary to compute the actual address. Access to the formal within the called routine involves an indirection which may or may not increase the code size and execution time. In a program with several hundred call sites and an average of four parameters per site, a reduction of 30% or 40% in call overhead can save on the order of a thousand words or ~10% of the code size.

For less dynamic, e.g. display based, calling sequences, the savings may be less, but they are likely to be substantial.

Chapter 7
REDUCTION OF CALLING OVERHEAD BY DEPROCEDURING

As noted in Chapter 6, procedure and function calls are among the most common operators in Gypsy. There is evidence that this is also the case in well structured programs written in non-verifiable languages [Zelkowitz 74]. We have seen, the portion of the call overhead associated with parameter passage can be substantially reduced by converting some parameters to global variables. In this chapter, we look at another technique for the reduction of call overhead, the expansion of routine bodies in line at their call sites.

The suggestion for such a procedure can be traced to pages 215-235 of [von Neumann 61] although it is clear that the motivation here is the reuse of code rather than increasing the efficiency of the call mechanism since von Neumann's methodology does not appear to contain routine calls per se, and can more properly be said to anticipate macro expansion. Open subroutine instantiation appears to have been the rule in early programming systems [Backus 67H] although by the late 1950's, the separately compiled subroutine and closed linkage had appeared in the Fortran II system(*ibid*).

Just as the additional control paths introduced by exceptions thwart conventional optimizations, so do the discontinuities in control introduced by routine calls implemented with closed linkages. Clever design of calling sequences can minimize the disruptions in some cases, but this may be at the expense of introducing considerable machine dependent information into early, machine independent phases of optimization and code generation.

Expansion of routines at their call sites should invariably increase the speed of the resulting code by eliminating the linkage overhead. Depending on the relationship between the size of the linkage and the size of the routine body, it may either reduce or increase the size of the program. It appears clear that all instances of routine calls in which the calling sequence is larger than the body of the called routine should be implemented by replacing the routine call with the routine body. At the other extreme, it is equally clear that any routine which is called only once in a program can be instantiated effectively at that call site. Between these two extremes lies a very hard and, in general, intractable problem. There is evidence that the main advantage to be derived from routine expansion is the increased applicability of other optimizations [Scheifler 77, Carter 77]. In this case, a substantial gain in both time and space may be obtained from the expansion of routines which are larger than their calling sequences.

7.1 Identifying Expandable Routines

The requirements of verifiability only add to the tendencies, inspired by a desire for modularity and abstraction, of modern programs to be composed of many small functions and procedures which can be effectively instantiated at their call sites. For the purposes of this investigation, we have identified two classes of routines which we suspect will benefit significantly from such expansion.

The first is the class of procedures for which only a single call site exists. This class is actually identified during the globalization phase of the analysis and no implementation aid beyond this identification has been provided. This is largely because the internal representation used in the GVE does not contain adequate constructs for the representation of an arbitrary routine body as a statement. On the other hand, it appears that the translator, which produces BLISS "object" code from the internal representation is capable of producing such expansion during BLISS generation with relatively minor modifications. Lack of manpower on the Gypsy project and the relatively small potential benefits to be expected from the expansion of one time calls are the primary reasons why this has not yet been implemented.

The second class of routines identified as suitable for inline expansion consists of those which have exactly one variable parameter and for which an equivalent routine can be found consisting of a single assignment to that variable parameter. This class contains functions (for which the implicit RESULT parameter is the only variable) as well as procedures. It appears that many routines associated with Gypsy abstract types will satisfy these criteria although, until recently, incomplete support for Gypsy abstract types has limited experience in this area.

7.1.1 Interaction with Exceptions

As we noted above, a major gain is to be expected from eliminating the control flow discontinuities caused by routine calls. If the call or expanded code is a source of signal paths, much of this advantage may be lost. Exception sites within a routine may also complicate finding its single assignment form. Consider the following:

```
Function Foo (I,J,K:Int):[-10..10]=
  Begin
    Entry(I GE 0) and ABS(J-K) LE 10*(I+1)
    Var II:[1..Maxint];
    II:=I+1;
    RESULT:=(J-K)/II;
  End;
```

If this function returns normally, it is equivalent to (J-K)/I. This form admits three exceptions, "subtracterror", "divideerrror", and "zerodivide". The function definition given above admits two additional exceptions; "valueerror" can occur at each assignment. The first term of the entry specification is sufficient to prove that the potential "valueerror" exception for the assignment to II will not occur. Similarly, the second term can prove the absence of the exception for the assignment to RESULT. In order to expand a call to FOO, we must prove the entry specifications of FOO at the call site.

We noted above that the expression form for FOO retained three potential exceptions from the body of FOO. If any of these are raised during an invocation of FOO, "routineerror" would be signaled at the call site. The exception checks implemented in connection with the expansion of FOO must signal "routineerror" rather than their usual conditions.

7.2 Implementation

Starting with the main routine for a program, all units in the calling tree are checked for suitability for expansion. The criteria used are:

1. The routine is either a function or has a single VAR parameter.
2. The routine has no validated entry or exit specifications.
3. The routine has no WHEN part on its statement list.
4. The routine body consists only of assignment statements.

The last criterion assures that the routine body can be represented effectively by a single expression which calculates the final value of its VAR parameter or RESULT. It is a stronger criterion than necessary, but probably covers a majority of the cases where expansion would be effective.

If a routine meets the criteria, its statement list is evaluated to determine the equivalent expression. This evaluation is performed using the symbolic evaluator XEVAL which is part of the GVE. Its name is added to the list of expandable routines kept under the property OPT:EXPANDABLE_ROUTINES under the main program in the database. The processed expansion is entered in the routine's symbol table under the property OPT:EXPANSION. The expansion consists of the following items:

- EXP-NAME - the name of the routine
- EXP-FORMALS - a list of the routine's formal parameters
- EXP-VAR - the name of the single variable parameter of the routine
- EXP-BODY - the expression formed by processing the routine body, annotated so that condition parameters can be properly instantiated and optimization conditions checked
- EXP-OCS - a list of statement numbers within the routine for which "valueerror" OCs must be proved before expansion is permitted

In order to perform an expansion, it is necessary to know that routine's entry conditions have been proved at the call site, and that the internal "valueerror" OCs from the routine have likewise been proved. Once this is done, the EXP-BODY component of the expansion can be instantiated with the actuals from the call site. The instantiated expression will directly replace a function call or used to form an assignment statement with the instantiation of EXP-VAR as its left side to replace a procedure call.

7.3 Analysis

As with the other phases of the optimizer, an analysis and report generation facility is included. The expansion analyzer generates a report which shows the expansion of each expandable routine. A second report summarizes the sites at which each routine is called, giving the potential expansions. Chapter 9 contains fragments of the reports generated by the expandability analyzer.

Chapter 8
ELIMINATION OF "PROOF ONLY" VARIABLES

An integrated programming and specification language such as Gypsy or Euclid (or Ada with the Anna [Anna 80] annotation extension) may contain variables and code which serve only to facilitate proofs. Such variables, and any code which references them, may be eliminated without altering the effect of the program. The problem of identifying specification-only variables which we will call "ghost" variables is the dual of the information flow problem of computer security.

8.1 The Information Flow Problem

Briefly stated, the information flow problem [Reitman 79, Denning 77, Landwehr 81] is:

- Given a program and sets of input and output variables to the program, determine for each output variable, the subset of the input variable set about which it might possibly contain information after execution of the program.

Security information flow analysis is, of course, concerned with the relative classifications of the input and output variables and with insuring that the program behavior conforms with some security policy, say that information does not flow from a variable of security level N to one of security level P if P<N.

Ghost analysis is concerned not only with information flow but with the vehicles for the flow. Thus we may characterize the ghost finding problem as

- Given an information flow characterization of a program plus information about the internal program variables which facilitated the flows, determine the set of internal and input variables which do not contribute to the information flow to any output variable.

8.2 An Algorithm for Finding "Ghosts"

We present a two pass algorithm for performing the information flow analysis necessary to find ghosts in a program. We assume that the information flow characteristics of any predefined or library routines are known. The information flow rules for Gypsy statements are discussed in section 8.3 below.

1. Compute a calling tree from the main routine of the program to be analyzed.
2. Order the routines in the calling tree such that each routine precedes all routine which call it[20] .

[20]This is not possible if recursive call cycles exist in the program. We ignore this possibility for now.

3. For each routine in the list, record the information flows to its VAR parameters.

4. Reorder the routines in the calling tree such that each routine follows any routines which call it.

5. For each routine calculate the set of local ghosts as follows:

    a. Remove from the formal parameter objects of the routine any objects[21] all of whose corresponding actuals at all call sites are ghosts.

    b. From the information flow for the remaining parameters, determine all objects (parameter and local) which act as sources or vehicles for information flow to the remaining parameters. Record these as the non-ghosts of the routine.

    c. Subtract the set of non-ghosts determined above from the set of all objects, parameter and local, known in the routine. The remainder, if any, is the set of ghosts for the routine.

8.3 Information Flow Rules For Gypsy

For most Gypsy statements, the information rules are straightforward and are similar to those given in [Reitman 79]. Ignoring concurrency and exception flow, Gypsy can be reduced to nine information flow statements:

- (T-ASSIGN LHS-OBJECT AFFECTORS) reflecting a total assignment of an object in which its state is completely replaced by the states of the affectors.

- (P-ASSIGN LHS-OBJECT AFFECTORS) reflecting a partial assignment in which the states of the affectors are added to the state of the object.

- (CONTROL AFFECTORS) in which the affectors are pushed into the control state which affects all assignments.

- (DECONTROL) which pops the control component of the state.

- (PARALLEL SL ... SL) The statement lists are evaluated in parallel starting from the same state, and resulting in a merged state at the end.

- (LOOP SL) A loop body on which closure must be performed before evaluating the path through the loop.

- (RSIGNAL COND ... COND) Merge the present state with the exit state associated with each COND.

- (PATH S ... S) A branch path to the end of the routine. Path evaluation starts in the current state, and the final state of the path is merged into the NORMAL exit state of the routine. Paths result from handled signals and loop exits.

- (ABORT) Return the null state.

Pass 1 of the information flow analysis reduces the extended body of a Gypsy routine to a tree of these information flow statements. For example:

```
A := B + C
```

becomes

[21] An object is a variable, a constant, or any component such as a record field of a variable or constant.

```
        (T-ASSIGN A (B C)),
```

while

```
        A[I] := B + C
```

becomes

```
        (P-ASSIGN A (B C I)).
```

The partial assignment indicates that prior state of A still affects A after the assignment to an element of A. Procedure calls become a parallel series of assignments to the VAR actuals of the call using the information flows from the previously completed analysis of the called procedure instantiated on the actual parameters. Normal control constructs such as IF and CASE statements also create parallel structures. The parallel structures are bracketed with a CONTROL - DECONTROL pair to show the domain in which information flows from the control expression. This avoids the necessity of dealing with changes in the states of the control objects within the scope of their influence. Loops require special treatment because we need to perform a transitive closure of the information flow within the loop body before considering flows which leave the loop.

8.3.1 Concurrency

Concurrency introduces another complication. Buffers are shared among the arms of a COBEGIN and we must take into account the possibility of information flow from parameters of one routine to parameters of another routine via buffers. Is is sufficient to combine the buffer assignments from the parallel assignments resulting from the evaluations of each call in the COBEGIN and perform a transitive closure on them prior to evaluating the parallel structure of the COBEGIN itself. This is done by combining the buffer assignment into an information flow LOOP statement.

Information can also flow from the blockage behavior of concurrent programs. Gypsy send and receive statements block when the buffer addressed is full or empty respectively. The blockage or non-blockage of a buffer operation can transfer information in much the same way as the control expression of an IF statement. We introduce a BUFFER statement into our information flow model to account for this. There is no DEBUFFER statement because the influence of the buffer action continues throughout the remainder of the program.

8.3.2 Exceptions

Exceptions transfer information about their source in a manner similar to control expressions. If a handler for the signaled exception exists, the information flow persists until the exception and non-exception paths merge after the handler. If no handler exists, the flow persists to the respective normal and exception EXITS of the routine and is instantiated with actual parameters at every call site of the routine. Given that potential exceptions dominate the control flow of Gypsy programs, it is not surprising that they also dominate the information flow. The net effect of considering all potential signal paths in a routine is potential information flow from all objects involved in exception raising operations to all objects normally operated upon in paths from the exception site to the routine's exits. This noise, as we may call it, obscures the non-transfers we hope to identify as ghosts.

Just as we can avoid exception checking code for situations where we can show that the exceptions cannot occur, we can avoid tracing the paths from such signals, and our flow analyzed incorporates a mechanism similar to the one available to the code generator for this purpose.

For those exceptions we do trace, we introduce an additional information flow statement, SIGNAL, which records the condition which will cover the subsequent path split for the normal and exceptional cases. To note the scopes of condition handlers within the information flow paths, we also add an S_PUSH statement to indicate entry to a handler scope and an S_POP to indicate exit from a handler.

8.4 Symbolic Evaluation of Information Flow

Pass 2 of the information flow analysis performs a symbolic evaluation of the tree constructed during Pass 1. The evaluation builds a set of final states, one for each condition return from the routine. The evaluation is performed with respect to a state consisting of entries for each object in the routine plus three special entries. Objects named in these special components transfer information to all objects receiving information as a result of T or P-ASSIGN statements.

- BUFFER* contains the names of all buffer objects on which blockage operations have been performed during path evaluation.
- CONTROL* contains the names of all objects used in control expressions which have not yet terminated. It is treated as a stack and operated upon by CONTROL and DECONTROL statements.
- SIGNAL* contains entries for all conditions for which handlers are currently active. Its structure is a stack of lists. S_PUSH statements create a new layer of conditions, S_POP statements remove a layer. SIGNAL statements copy the objects in the current control context plus the information generated by the signaling operation to the topmost entry for the condition named in the SIGNAL.

As the tree is traversed, the state is modified in response to the T-ASSIGN and P-ASSIGN statements, under the influence of the special state components. PARALLEL statements cause evaluation of each arm and a merge of the final arm states. RSIGNAL statements merge the current state with a final condition state. PATH statements cause a split in evaluation with no subsequent merge. LOOP statements perform a transitive closure of the loop's final state starting from an empty state, followed by an application of the closure to the loop entry state. ABORT statements cause path evaluation to terminate and return an empty state.

At the end of evaluation, we have as the final state values of each var parameter object, a list of all objects from or through which information could flow during the routine's execution. A separate list is created for each condition that the routine could signal.

8.5 Discussion

The information flow analysis is carried out in substantially more detail than is necessary for the detection of ghosts. It is hoped that this detail will simplify the addition of an information flow security tool to the GVE at a future time. In practice, the ghost analysis is a concept whose real worth has yet to be assessed. The motivation for this analysis arose from the observation that many Gypsy programs being written for specification and design purposes contained ghost variables. Typically these variables contained buffer histories, a feature of the language which is not implemented. The examples which we have, being executable cannot contain references to buffer histories except in non-executable specifications. If the code generation (BLISS translator) phase of the GVE is modified to use the ghost information, then it will be possible to introduce history ghosts into executable Gypsy programs and subsequently eliminate them.

As we shall see in the discussion of optimization results in Chapter 9, the usual catch from ghost analysis is rather insignificant, consisting of activationids and primed var parameters which can be identified much more easily by simply noting that they are not used in the executable portion of the program. As is the case with other aspects of this investigation, a larger sample of verified, executable programs based on a more complete implementation of the language is highly desirable.

Chapter 9
RESULTS

The optimization techniques described in the previous chapters have been applied to two programs of moderate size and several small examples with mixed but encouraging results. The two larger examples are:

- CHASE, a game playing program, written without specifications.
- FLOMOD, the message flow modulator described in [Good 82].

The smaller examples include:

- FM-GHOST, a fragment of a version of the flow modulator with ghost variables.
- SNEAK, a pathological example from the security world.

Listings of portions of these programs appear in appendices B through D. The most complete analyses have been performed on CHASE and FLOMOD, and these will be discussed first. The smaller examples are contrived to illustrate particular aspects of the optimization process and complete analyses do not necessarily contribute to the discussion.

9.1 CHASE

CHASE implements a video game in which a lone human (the user of the program) tries to escape from a flock of robots. It was translated from a PDP-10 version shortly after Gypsy was first implemented for the PDP-11 and consists of 44 routines plus numerous types and constants. Routines range, in the present version, from about 5 to over 100 lines of code. Initially, CHASE had no specifications at all. Prior to the current optimization effort, the author added loop assertions (typically the negation of the loop exit criterion immediately following the exit) to facilitate the generation of optimization conditions. Appendix C contains the routines MAIN and PLAY from the analyzed version of CHASE as well as the original version of the PLAY routine.

Using the original version of CHASE, no problems were encountered in performing the global parameter analysis. Generation of optimization conditions posed major difficulties. After finding over 3000 execution paths and consuming 5 cpu hours on a DEC2060 in the PLAY routine, the verification condition generator exhausted its available list space and broke. CHASE was then modified to reduce the path structure by breaking some of the larger routines into pieces. This reduced the number of paths in PLAY to 72, still a very large number by verification standards and reduced the cpu time required for VC and OC generation to about one hour.

Only two routines in CHASE, one procedure and one function meet the criteria for inline expansion. These are each called exactly once and would be candidates for expansion on that basis in any event. Twenty six other routines are called only once and are thus suitable candidates for expansion.

9.1.1 Parameter Globalization

Table 9-1 describes the potential gains which could be obtained by converting parameters to global variables in CHASE. The first column of the table shows the number of sites involved in the analysis. In the upper part of the table, this is the number of routines or called sites, while in the lower part it is the number of calling sites. The second column is the total number of parameters involved. The third and fourth columns indicate the number and percentage of parameters which could be made global at any level in the program, while the fifth and sixth indicate the number and percentage of parameters which are either locals in the main routine or are parameters to the main routine. This latter group can be made global without increasing the effective size of the program since they would have to be allocated for the entire life of the program. The upper and lower parts of the table analyze the globalization in terms of both called and calling sites. Within each category the analysis is performed for all parameters and then repeated for data and condition parameters. Each analysis is also performed for all routines, for ,those with a single call site and for those with more than one call site.

If we assume that all routines called once are expanded in line, we can save about 160 parameter passages. At three words per passage, this is about 500 words or 10% of the code size (5300 words) of the program. Because access to a global is potentially faster and smaller than access to a parameter in the BLISS implementation of Gypsy, globalizing parameters should also make the routines which are affected smaller and faster also.

9.1.2 Exception Sites in CHASE

VC and OC generation for CHASE provided 60 verification conditions and 481 optimization conditions. Of these, 59 and 43 were proved during the generation process. The 481 OCs cover 1084 potential exception sites while the 43 proved OCs cover 548 of these. OCs covering 758 sites can be trivially proved using a state consisting of the global constants from the program. Tables 9-2 - 9-5 show the status of the VCs and OCs immediately after generation. It appears that a substantial number of the OCs can be proved easily in the prover. Clearly, some OCs should not be provable since the program depends on implicit exceptions for control in several places. It is difficult to estimate exactly the savings to be obtained by eliminating the checking code. The present GVE's Gypsy to BLISS translator performs some optimization of this sort already, eliminating "valueerror" checks whenever the types of the objects concerned are identical. This probably includes a majority of the "valueerror" sites identified in table 9-5. The last column of this table "Sites True" counts those sites for which the optimization condition had a true conclusion during VC generation. The valueerror sites identified by the translator probably comprise a large subset of these. The translator is also aware that "divideerror" cannot occur on the PDP-11. From this, we infer that the translator suppresses exception checking at about half of the exception sites. With little or no effort, we can suppress this code at another quarter of the sites. Most of these are "valueerror" sites for which the checking code involves a minimum of 6 words if performed inline (and about the same for a condition returning out of line check). We estimate a direct savings of about 1000 words of code for eliminating these additional checks. This is nearly 20% of the size of CHASE.

There are 257 optimization conditions from CHASE which are awaiting proof. Inspection of a random sample of these indicates that the proofs are likely to be non-trivial in many cases. This is due to the general lack of strong typing in CHASE and to the use of exceptions for control purposes at many places in the program. The lack of specifications in CHASE also contributes to the difficulty of the proofs. The only information available to aid in the proof process is that which is derived from type information and from flow control information and assignment state information in the program itself.

Table 9-1: Parameter Globalization for CHASE

Analysis for Called routines

Analysis	Count	Params	Globals	Percent	Mains	Percent
All parameters						
All routines	43	196	156	(80%)	133	(68%)
Called once	28	121	113	(93%)	90	(74%)
Multiple calls	15	75	43	(57%)	43	(57%)
Data Parameters						
All routines	43	138	108	(78%)	86	(62%)
Called once	28	84	81	(96%)	59	(70%)
Multiple calls	15	54	27	(50%)	27	(50%)
Cond parameters						
All routines	43	58	48	(83%)	47	(81%)
Called once	28	37	32	(86%)	31	(84%)
Multiple calls	15	21	16	(76%)	16	(76%)

Analysis of calling sites

Analysis	Count	Params	Globals	Percent	Mains	Percent
All parameters						
All call sites	121	541	277	(51%)	254	(47%)
Omit single calls	93	420	164	(39%)	164	(39%)
Data parameters						
All call sites	121	395	201	(51%)	179	(45%)
Omit single calls	93	311	120	(39%)	120	(39%)
Cond parameters						
All call sites	121	146	76	(52%)	75	(51%)
Omit single calls	93	109	44	(40%)	44	(40%)

Table 9-2: Initial VC status for CHASE

Unit Name	Proof Gen	Status Prvd
CHANGE_COORDS	1	1
CLEAR_SCREEN	1	1
CRLF	1	1
FIND_DIR	1	1
GET_COORDS	1	1
INITIALIZE_BOARD	1	1
INITIALIZE_BOARD_CONTENTS	2	2
INITIALIZE_BOARD_LEVEL	1	1
INITIALIZE_BOARD_ROBOTS	2	2
INTRO_INIT	1	1
INTRO_INIT_TERM	1	1
KILL_IT	1	1
MAIN	1	1
MAIN_FINISH	1	1
MOVE_OBJECT	2	2
MOVE_OBJECT_SUPER	1	1
MOVE_OBJECT_UPDATE	1	1
PLACE_THEM	2	2
PLAY	2	2
PLAY_CLEAR_BUFFER	1	1
PLAY_DO_CMD	1	1
PLAY_GET_CMD	1	1
POSITION	1	1
PRINT	1	1
PRINT_BLANK	2	2
PRINT_BOARD	2	2
PRINT_BOARD_BOTTOM	2	2
PRINT_BOARD_SIDES	2	2
PRINT_BOARD_TOP	2	2
PRINT_NUMBER	2	2
PRINT_NUMBER_CONVERT	2	2
PRINT_PAUSE	2	2
PRINT_STAT	1	1
PRINT_STRING	3	2
PUT_CHAR	1	1
RAND_GEN	1	1
RAND_NUM	1	1
RECEIVE_DIRECTION	1	1
SET_LEVEL	1	1
SHOOT_IT	1	1
SHOOT_IT_SHOTS	1	1
UPDATE	2	2
UPDATE_DISTANCE	1	1
UPDATE_NEW_SUPER	1	1
Calling tree summary	60	59

Table 9-3: Initial OC status for CHASE

Unit Name	Optimization Conditions					
	Gen	Vcg*	Xvl*	Pvbl	Total	False
CHANGE_COORDS	17	1	0	0	1	0
CLEAR_SCREEN	5	1	0	0	1	0
CRLF	3	1	0	0	1	0
FIND_DIR	40	0	21	0	21	0
GET_COORDS	4	1	0	0	1	0
INITIALIZE_BOARD	21	1	0	2	3	0
INITIALIZE_BOARD_CONTENTS	22	1	5	6	12	0
INITIALIZE_BOARD_LEVEL	1	1	0	0	1	0
INITIALIZE_BOARD_ROBOTS	9	1	1	2	4	0
INTRO_INIT	6	1	1	1	3	0
INTRO_INIT_TERM	1	1	0	0	1	0
KILL_IT	11	1	4	0	5	0
MAIN	12	1	2	1	4	0
MAIN_FINISH	10	1	1	1	3	0
MOVE_OBJECT	10	1	0	0	1	0
MOVE_OBJECT_SUPER	3	1	0	0	1	0
MOVE_OBJECT_UPDATE	2	1	0	0	1	0
PLACE_THEM	16	1	0	1	2	0
PLAY	97	1	31	39	71	0
PLAY_CLEAR_BUFFER	1	1	0	0	1	0
PLAY_DO_CMD	6	1	0	0	1	0
PLAY_GET_CMD	6	1	0	0	1	0
POSITION	9	1	0	1	2	0
PRINT	4	1	1	2	4	0
PRINT_BLANK	5	1	0	1	2	0
PRINT_BOARD	30	1	13	9	23	0
PRINT_BOARD_BOTTOM	4	1	0	0	1	0
PRINT_BOARD_SIDES	10	1	2	1	4	0
PRINT_BOARD_TOP	6	1	1	0	2	0
PRINT_NUMBER	7	1	0	0	1	0
PRINT_NUMBER_CONVERT	17	1	1	1	3	0
PRINT_PAUSE	5	1	0	0	1	0
PRINT_STAT	12	1	7	1	9	0
PRINT_STRING	5	1	0	2	3	0
PUT_CHAR	1	1	0	0	1	0
RAND_GEN	4	1	0	0	1	0
RAND_NUM	6	1	0	0	1	0
RECEIVE_DIRECTION	3	1	0	0	1	0
SET_LEVEL	1	1	0	0	1	0
SHOOT_IT	9	1	2	0	3	0
SHOOT_IT_SHOTS	6	1	2	0	3	0
UPDATE	26	1	9	3	13	0
UPDATE_DISTANCE	6	1	2	0	3	0
UPDATE_NEW_SUPER	2	1	1	0	2	0
Calling tree summary	481	43	107	74	224	0

Table 9-4: Initial OC site status for CHASE

Unit Name	Optimization Sites						
	Total	Prvd	Rtn Prvd	Safe Entry	T in State	Tc in State	F
CHANGE_COORDS	18	2	2	2	2	2	0
CLEAR_SCREEN	6	2	2	2	2	2	0
CRLF	7	5	5	5	5	5	0
FIND_DIR	42	0	21	21	21	21	0
GET_COORDS	11	7	7	7	7	7	0
INITIALIZE_BOARD	69	36	40	40	40	36	0
INITIALIZE_BOARD_CONTENTS	25	2	14	14	14	7	0
INITIALIZE_BOARD_LEVEL	12	12	12	12	12	12	0
INITIALIZE_BOARD_ROBOTS	10	2	5	5	5	3	0
INTRO_INIT	28	23	25	25	25	24	0
INTRO_INIT_TERM	12	12	12	12	12	12	0
KILL_IT	19	9	13	13	13	13	0
MAIN	78	48	55	55	55	54	0
MAIN_FINISH	34	24	26	26	26	25	0
MOVE_OBJECT	65	50	50	50	50	50	0
MOVE_OBJECT_SUPER	3	1	1	1	1	1	0
MOVE_OBJECT_UPDATE	7	5	5	5	5	5	0
PLACE_THEM	32	13	14	14	14	13	0
PLAY	133	32	106	106	106	65	0
PLAY_CLEAR_BUFFER	3	3	3	3	3	3	0
PLAY_DO_CMD	28	23	23	23	23	23	0
PLAY_GET_CMD	30	25	25	25	25	25	0
POSITION	20	6	7	7	7	6	0
PRINT	8	5	8	8	8	6	0
PRINT_BLANK	8	4	5	5	5	4	0
PRINT_BOARD	41	9	33	33	33	23	0
PRINT_BOARD_BOTTOM	6	3	3	3	3	3	0
PRINT_BOARD_SIDES	18	9	12	12	12	11	0
PRINT_BOARD_TOP	7	2	3	3	3	3	0
PRINT_NUMBER	14	8	8	8	8	8	0
PRINT_NUMBER_CONVERT	21	5	7	7	7	6	0
PRINT_PAUSE	5	1	1	1	1	1	0
PRINT_STAT	71	48	62	62	62	61	0
PRINT_STRING	6	3	0	0	4	2	0
PUT_CHAR	6	6	6	6	6	6	0
RAND_GEN	4	1	1	1	1	1	0
RAND_NUM	9	4	4	4	4	4	0
RECEIVE_DIRECTION	7	5	5	5	5	5	0
SET_LEVEL	12	12	12	12	12	12	0
SHOOT_IT	47	34	36	36	36	36	0
SHOOT_IT_SHOTS	12	7	9	9	9	9	0
UPDATE	74	31	54	54	54	51	0
UPDATE_DISTANCE	10	4	6	6	6	6	0
UPDATE_NEW_SUPER	6	5	6	6	6	6	0
Calling tree summary	1084	548	754	754	758	678	0

Table 9-5: Initial OC status by condition for CHASE

	Opt Conds		Opt Sites	Sites	Sites
Condition Name	Gen	True	Fnd	Prvd	True
ADDERROR	75	12	89	14	0
DIVIDEERROR	6	0	6	0	0
INDEXERROR	51	16	81	28	1
MINUSERROR	1	0	1	0	0
MULTIPLYERROR	3	0	3	0	0
OVERSCALE	14	0	16	0	0
SUBTRACTERROR	45	9	45	9	0
UNDERSCALE	11	6	16	9	8
VALUEERROR	281	187	822	693	533
Assignments	173	115	373	294	219
Procedure Arguments	136	95	446	396	311
Function Arguments	3	3	3	3	3
ZERODIVIDE	4	4	5	5	5
Summary	491	234	1084	758	547

9.1.3 Ghosts and information flow in CHASE

Information flow and ghost analysis of CHASE do not provide substantial optimization information. This is to be expected, since without specifications, there is no reason to introduce specification only variables into the program. Ghost analysis does identify a number of instances where implicit data items appear as ghosts. These are the activationids, MYID, of procedures and the primed var parameters of procedures. Both of these items appear in Gypsy to facilitate proofs, but are seldom referenced in executable code, although the primed var parameters which are copies of the entry values of var parameters can be useful under some circumstances. It is much easier to eliminate these items from the run time image of the program by noting that they are never used in the code than by the information flow and ghost elimination process. This is, in fact, done in the present BLISS translator.

9.1.4 Inline expansions in CHASE

As can be seen from table 9-1, of the 43 routines which are called at least once in CHASE, 28 are called exactly once. Implementing these routines as blocks in the ALGOL sense would eliminate their calling and parameter passage overhead. Using our previous estimate of 3 words of code per parameter per call site, this would eliminate another 363 words of code in parameter passage. The calls themselves require 2 words, so another 56 words could be saved. The total to be gained by expanding all calls of routines called only once as blocks is thus about 8% of the code size of the program.

9.1.5 Conclusions

CHASE is an example of an unspecified, poorly structured program. Despite this, it seems to be possible to make modest improvements in its size and performance. The major contribution in this area comes from exception suppression. The current BLISS translation is probably about 2000 words shorter than it would be if it did not suppress "valueerror" checks in cases of type equality. Suppressing exception checking at the remaining sites having true optimization conditions would save an additional 1000 words. It is not possible to estimate the additional savings made possible by opening up the code for additional optimizations when the exception paths are eliminated. Expanding those routines called only once at their call sites and globalizing parameters to the remaining routines would save another 900 or so words. Together these optimizations would reduce the size of CHASE by about 35% of its present size. If we consider a completely checked version as the base case, the reduction is on the order of 50%.

9.2 FLOMOD

The flow modulator is described in detail in [Good 82]. Unlike CHASE, the flow modulator has formal specifications and has been proved. The code for the portions of the program appears in appendix D. The executable part of the program consists of 50 units from seven scopes. In addition to the executable units, there are numerous specification functions used to support the proof process. For the purposes of our analysis, we consider the entire program to have been proved prior to the optimization effort. Of the 206 proof VCs[22] generated for the executable routines of the flow modulator, 58 were proved during the generation process and the remaining ones were proved using the interactive theorem prover of the GVE.

The executable portions of the flow modulator comprise 556 lines of Gypsy code which compile into 3849 words of code for the LSI-11. This makes the flow modulator about 75% of the size of CHASE.

9.2.1 Parameter Globalization

As can be seen from Table 9-6, the flow modulator contains far fewer parameter passage sites (258) than does CHASE (541). Still fewer of these sites are subject to globalization. Indeed, if we eliminate the routines which have only a single call site, there are a total of 122 parameter passes in the flow modulator of which only 38 are potentially convertible to global references and only 29 of these are subject to a relatively simple implementation as globals at or above the main routine. For data parameters, the percentage subject to globalization is approximately the same as for CHASE i.e. 35% vs. 39%, while for condition parameters, it is much lower i.e. 22% vs. 40%. Looking at the code, we discover that the culprit is a WHEN ELSE construct in LSI_MAIN, the main program. There are paths to several of the multiply called routines from both within and without this handler so that the usual match of each routines "routineerror" to the formal "routineerror" of the main program does not apply. The flow modulator does not use exceptions for control purposes, and has no explicitly declared conditions. The WHEN ELSE serves as a trap for any unexpected errors during the operation of the modulator. Unfortunately, it uses several low level utility routines to issue it message that something is wrong. The same routines are also used in the body of the modulator. These calls are, of course, outside the scope of the handler. Implementing a global parameter mechanism for the flow modulator would save about 90 words or slightly over 2% of the program size. Time savings could be more significant since most of the savings appear to lie deep within the inner loops of the program.

9.2.2 Exception sites in the flow modulator

During the generation of verification and optimization conditions for the flow modulator, a total of 511 potential exception sites were identified. These sites are covered by 223 OCs. Of the 223 OCs, 87 covering 368 of the potential sites were either found to be true during VC generation or trivially proved using a global constant state. This leaves 136 optimization conditions to be proven using the interactive theorem prover in the GVE. Tables 9-7 and 9-8 summarize the initial proof status of the optimization conditions for the executable routines of the flow modulator. Because we have proven the verification conditions of the flow modulator, the various entry criteria for suppressing exception code produce the same results.

[22] [Good 82] notes that there were 348 VCs generated for the flow modulator. This figure includes 44 lemmas and VCs from nearly 100 non-executable specification functions.

Table 9-6: Parameter Globalization for FLOMOD

Analysis for Called routines

Analysis	Count	Params	Globals	Percent	Mains	Percent
All parameters						
All routines	49	163	130	(80%)	69	(42%)
Called once	40	136	120	(88%)	62	(46%)
Multiple calls	9	27	10	(37%)	7	(26%)
Data Parameters						
All routines	49	114	91	(80%)	30	(26%)
Called once	40	96	84	(88%)	26	(27%)
Multiple calls	9	18	7	(39%)	4	(22%)
Cond parameters						
All routines	49	49	39	(80%)	39	(80%)
Called once	40	40	36	(90%)	36	(90%)
Multiple calls	9	9	3	(33%)	3	(33%)

Analysis of calling sites

Analysis	Count	Params	Globals	Percent	Mains	Percent
All parameters						
All call sites	76	258	158	(61%)	91	(35%)
Omit single calls	36	122	38	(31%)	29	(24%)
Data parameters						
All call sites	76	182	114	(63%)	47	(26%)
Omit single calls	36	86	30	(35%)	21	(24%)
Cond parameters						
All call sites	76	76	44	(58%)	44	(58%)
Omit single calls	36	36	8	(22%)	8	(22%)

Table 9-7: Initial OC status for FLOMOD

Unit Name	Optimization Conditions					
	Gen	Vcg*	Xvl*	Pvbl	Total	False
Units from scope FLOW_MODULATOR						
ADD_MSG_CHAR	8	1	1	0	2	0
BAD_FILTER	1	1	0	0	1	0
BOM	2	1	0	1	2	0
EOM	2	1	0	1	2	0
MODULATOR	1	1	0	0	1	0
MSG_COMPLETED	4	1	0	0	1	0
PROCESS_MSG	1	1	0	0	1	0
RESET_MSG	2	1	0	0	1	0
START_MODULATOR	4	1	1	0	2	0
Units from scope MAIN_PROGRAMS						
LSI_MAIN	4	1	1	1	3	0
Units from scope MSG_LOGGER						
DASHES	1	1	0	0	1	0
IS_MSG_LINE	4	1	0	1	2	0
LOG_MSG	3	1	0	1	2	0
LOG_MSG_TEXT	8	1	2	0	3	0
LOG_PATTERN_MATCH	11	1	3	0	4	0
LOG_REJECTION_HEADER	3	1	0	1	2	0
MESSAGE	1	1	0	0	1	0
MESSAGE_SEGMENT	1	1	0	0	1	0
MESSAGE_TOO_LONG	1	1	0	0	1	0
NEW_LINES	5	1	2	0	3	0
NEXT_CHAR_POSITION	4	1	1	0	2	0
REJECTED_TEXT	1	1	0	0	1	0
SEND_CHAR	6	1	1	0	2	0
SEND_LINE	2	1	1	0	2	0
SEND_STRING	5	1	0	1	2	0
Units from scope MSG_SENDER						
SEND_MSG	5	1	0	1	2	0

Table 9-7, concluded

Unit Name	Optimization Conditions					
	Gen	Vcg*	Xvl*	Pvbl	Total	False
Units from scope PATTERN_MATCHER						
CORRECTLY_ORDERED	6	1	0	2	3	0
FILTER_MSG	3	1	1	0	2	0
INDEX_CHAR	1	1	0	0	1	0
MATCH_ALL_PATTERNS	8	1	1	0	2	0
MATCH_PATTERN	11	1	1	0	2	0
NEXT_CHAR	6	1	0	0	1	0
NONDELIMITER	1	1	0	0	1	0
NORMALIZE_MSG	15	1	4	0	5	0
PATTERN_NONDELIMITER	1	1	0	0	1	0
PATTERN_OK	10	1	0	1	2	0
REMOVE_LEADING_DELIMITERS	2	1	0	0	1	0
SET_FILTER_INDEX	4	1	1	0	2	0
SET_INDEX_SECTION	7	1	0	1	2	0
VALID_FILTER	6	1	1	0	2	0
Units from scope STRING_UTILITIES						
BLANK	1	1	0	0	1	0
CARRIAGE_RETURN	2	1	0	0	1	0
DOT	1	1	0	0	1	0
LINE_FEED	2	1	0	0	1	0
PRINTABLE	1	1	0	0	1	0
PRINTABLE_FORM	16	1	0	1	2	0
UPPER_CASE	4	1	0	0	1	0
Units from scope UNKNOWN_GYPSY						
APPEND_STRING	7	1	1	0	2	0
SEQ_EQ_SUBSEQ	10	1	1	0	2	0
SUBSEQ_SELECT	8	1	0	0	1	0
Calling tree summary	223	50	24	13	87	0

50 units analyzed of which 50 have a proof status of PROVED

Table 9-8: Initial OC status for FLOMOD

Unit Name	Optimization Sites						
			Rtn	Safe	T in	Tc in	
	Total	Prvd	Prvd	Entry	State	State	F
Units from scope FLOW_MODULATOR							
ADD_MSG_CHAR	13	6	7	7	7	7	0
BAD_FILTER	2	2	2	2	2	2	0
BOM	2	1	2	2	2	1	0
EOM	2	1	2	2	2	1	0
MODULATOR	28	28	28	28	28	28	0
MSG_COMPLETED	8	5	5	5	5	5	0
PROCESS_MSG	17	17	17	17	17	17	0
RESET_MSG	4	3	3	3	3	3	0
START_MODULATOR	16	13	14	14	14	14	0
Units from scope MAIN_PROGRAMS							
LSI_MAIN	20	16	18	18	18	17	0
Units from scope MSG_LOGGER							
DASHES	2	2	2	2	2	2	0
IS_MSG_LINE	5	3	3	3	3	2	0
LOG_MSG	17	14	15	15	15	14	0
LOG_MSG_TEXT	17	10	12	12	12	12	0
LOG_PATTERN_MATCH	22	11	14	14	14	14	0
LOG_REJECTION_HEADER	9	6	7	7	7	6	0
MESSAGE	2	2	2	2	2	2	0
MESSAGE_SEGMENT	2	2	2	2	2	2	0
MESSAGE_TOO_LONG	2	2	2	2	2	2	0
NEW_LINES	8	4	6	6	6	6	0
NEXT_CHAR_POSITION	7	4	5	5	5	5	0
REJECTED_TEXT	2	2	2	2	2	2	0
SEND_CHAR	17	12	13	13	13	13	0
SEND_LINE	9	8	9	9	9	9	0
SEND_STRING	10	6	7	7	7	6	0
Units from scope MSG_SENDER							
SEND_MSG	9	5	6	6	6	5	0

Table 9-8, concluded

Unit Name	Optimization Sites						
			Rtn	Safe	T in	Tc in	
	Total	Prvd	Prvd	Entry	State	State	F
Units from scope PATTERN_MATCHER							
CORRECTLY_ORDERED	16	12	13	13	13	11	0
FILTER_MSG	22	20	21	21	21	21	0
INDEX_CHAR	2	2	2	2	2	2	0
MATCH_ALL_PATTERNS	26	18	19	19	19	19	0
MATCH_PATTERN	29	18	19	19	19	19	0
NEXT_CHAR	6	1	1	1	1	1	0
NONDELIMITER	2	2	2	2	2	2	0
NORMALIZE_MSG	23	8	12	12	12	12	0
PATTERN_NONDELIMITER	2	2	2	2	2	2	0
PATTERN_OK	20	11	12	12	12	11	0
REMOVE_LEADING_DELIMITERS	6	5	5	5	5	5	0
SET_FILTER_INDEX	12	8	10	10	10	10	0
SET_INDEX_SECTION	14	9	9	9	9	7	0
VALID_FILTER	11	6	7	7	7	7	0
Units from scope STRING_UTILITIES							
BLANK	2	2	2	2	2	2	0
CARRIAGE_RETURN	4	3	3	3	3	3	0
DOT	2	2	2	2	2	2	0
LINE_FEED	4	3	3	3	3	3	0
PRINTABLE	2	2	2	2	2	2	0
PRINTABLE_FORM	17	2	3	3	3	2	0
UPPER_CASE	6	3	3	3	3	3	0
Units from scope UNKNOWN_GYPSY							
APPEND_STRING	8	2	3	3	3	3	0
SEQ_EQ_SUBSEQ	13	4	5	5	5	5	0
SUBSEQ_SELECT	10	3	3	3	3	3	0
Calling tree summary	511	333	368	368	368	354	0

50 units analyzed of which 50 have a proof status of PROVED

Table 9-9: Initial OC status by condition for FLOMOD

	Opt Conds		Opt Sites	Sites	Sites
Condition Name	Gen	True	Fnd	Prvd	True
ADDERROR	34	0	34	0	0
INDEXERROR	37	5	38	6	2
NOSUCC	1	0	1	0	0
OVERSCALE	4	0	4	0	0
SUBTRACTERROR	11	0	11	0	0
UNDERSCALE	4	2	4	2	2
VALUEERROR	136	84	419	360	325
Assignments	119	72	249	198	176
Procedure Arguments	38	31	156	149	136
Function Arguments	10	9	14	13	13
Summary	227	91	511	368	329

In table 9-9 we see that the translator would probably omit exception checking at most of the 329 sites for which the optimization conditions have true conclusions on generation. Using global constant information and XEVAL allow us to add another 39 sites for a savings of about 230 words of code or about 6% of the program size. Further exception suppression requires proving of some of the 136 remaining OCs.

Using the interactive theorem prover of the GVE, proofs were found for all of the remaining OCs. In several cases, it was necessary to slightly modify type definitions or code in order to achieve the proof. One such modification involved limiting the length of a sequence in a utility routine, APPEND_STRING, to 32766 entries instead of the 32767 originally specified. Since the actual parameters to the routine have lengths which are limited to much smaller sizes, the change does not affect the correctness of the routine. In another case, LOG_PATTERN_MATCH, it was necessary to add portions of the entry specification concerning the relationships between the values of two constant parameters to the loop assertion in order to prove the OCs. Again, this change would not affect the correctness of the routine.

The prover used in the GVE lacks knowledge about several aspects of the Gypsy language. In particular, relationships among subranges of scalar types are particularly difficult to prove. Several times in proving the optimization conditions, it was necessary to assume unprovable but obviously TRUE lemmas about such relationships. Typical of these lemmas are:

```
    lemma UC_IN_STAR_TO_Z (C4 : CHARACTER) =
          C4 in ['A..'Z] -> C4 in ['*..'Z];
```

for character subranges and

```
    lemma ORD_OF_LITTLE_A (C1 : CHARACTER) =
          'a le C1 -> 97 le ORD (C1);

    lemma ORD_OF_LITTLE_Z (C4 : CHARACTER) =
          C4 le 'z -> ORD (C4) le 122;
```

relating character variables and their mappings onto the integers. These lemmas are similar, and in some case equivalent to those used in the correctness proof of the program.

Because of the nature of the VC generation and proof processes, it is difficult to determine the exact extent to which the proofs depend on user written specifications. Examination of the OCs and the code to which they apply in the cases in which the proofs involve more than simple substitutions, indicate that specification information is involved in quite a number of cases. Some of these proofs use range information which could probably be obtained by stronger typing of program variables, but OCs in the key routines in the pattern matcher require the instantiation of rather complex specification functions for their proofs.

The total time required to prove the OCs was about 40 man hours and 5 DEC 2060 CPU hours. This compares favorably with the 13 person-months and 220 CPU hours required to develop the program ([Good 82]). The most complicated proof, that of MATCH_ALL_PATTERNS#8 (Figure 9-1) required 250 interactions with the prover. This is because it was necessary to perform multilevel expansions of several routines which contained nested IF expressions in their specifications which in turn caused multiple branches in the proof tree. The routine in question, MATCH_ALL_PATTERNS, and the required specification routines are contained in appendix D.

```
H1: B = END_FILTER_SECTION (FC, STAR, FM[ST])
H2: CORRECT_FILTER_INDEX (FC, FILTER)
H3: DOING_FILTER_MATCH (M[ST..SIZE (M)], ST = 1 & AT_BOM, FILTER,
                        PASS_IT', PASS_IT, OUTTO (LOG, MYID),
                        I, B)
H4: MSGS_OK (M, AT_BOM, FM, ST)
H5: B + 1 le I
H6: ST le SIZE (FM)
H7: STAR
-->
C1: FM[ST] in ['*..'Z]
```

Figure 9-1: Verification condition MATCH_ALL_PATTERNS#8

9.2.3 Ghosts and information flow in FLOMOD

As was the case with CHASE, the information flow and ghost analysis procedures do not provide any useful information for the flow modulator which cannot be more easily derived by other methods. This is because the executable version of the program was constrained by the implementation not to use buffer histories in executable code. The motivation for the ghost analysis came from an earlier, design version of the program which passed a large state object, including buffer histories among its procedures.

9.2.4 Inline expansions in FLOMOD

In addition to the 40 routines of the flow modulator which are called only once, the program contains 4 functions which meet our secondary criteria for inline expansion. The results of this expansion analysis are shown in table 9-10. Note that all but one of the expandable functions represents a constant. Structured constants often appear in Gypsy programs as constant functions because of a deficiency in the parser which has a very narrow view of compile time computability.

These routines account for only 12 calls, but given that 13 other routines which are called only once also meet the single statement criteria for expansion, it is likely that performing both the single call site and the single statement expansions will open up the control structure of the program so as to make other optimizations possible. Performing all the indicated expansions would save about 530 words or about 14% of the program size.

9.2.5 Conclusions

The optimization results for the flow modulator are rather disappointing. Without proving the remaining optimization conditions, only about 20% further reduction in size can be obtained directly. Proving these appears to be possible, but substantial effort is involved. The estimated savings, were they proved is about 650 words or 17% of the current program size.

One reason why the flow modulator shows a relatively small gain from the optimization techniques

Table 9-10: Expandable Functions from FLOMOD

```
Expansion of routine CARRIAGE_RETURN::STRING_UTILITIES
        Formals:
        Expansion: RESULT := SCALE (13, CHARACTER)
        Called 4 times from :
             SEND_MSG::MSG_SENDER(22.1)
             SEND_MSG::MSG_SENDER(21.1)
             IS_MSG_LINE::MSG_LOGGER(11.7)
             NEW_LINES::MSG_LOGGER(19.1)

Expansion of routine LINE_FEED::STRING_UTILITIES
        Formals:
        Expansion: RESULT := SCALE (10, CHARACTER)
        Called 3 times from :
             SEND_MSG::MSG_SENDER(23.1)
             IS_MSG_LINE::MSG_LOGGER(11.10)
             NEW_LINES::MSG_LOGGER(20.1)

Expansion of routine EOM::FLOW_MODULATOR
        Formals:
        Expansion: RESULT := "NNNN"
        Called 2 times from :
             RESET_MSG::FLOW_MODULATOR(11.3)
             MSG_COMPLETED::FLOW_MODULATOR(14.1)

Expansion of routine INDEX_CHAR::PATTERN_MATCHER
        Formals:    P
        Expansion: RESULT := if P = NULL (A_PATTERN)
                             then '* else P[1] fi
        Called 3 times from :
             CORRECTLY_ORDERED::PATTERN_MATCHER(24.2)
             CORRECTLY_ORDERED::PATTERN_MATCHER(17.2)
             SET_INDEX_SECTION::PATTERN_MATCHER(17.4)
```

used here may be due to its good design and effective use of strong typing. The fact that the program does relatively little calculation, but rather moves its input data from one place to another based on the values of the data means that there are few calculated results and the types of objects across assignments and calls may be made to match exactly in most cases. The type equality tests used in the translator serve to eliminate a large number of potential "valueerror" tests. Most of the tests which remain involve the lengths of sequences which often require substantial effort to prove away in the GVE prover. It appears that user supplied specification information is used in a substantial number of the proofs.

Although the size reduction which might be obtained for the flow modulator is not as large as hoped, the elimination of the exception checks in the innermost sections of the pattern matcher should have a substantial effect on the speed of the program.

9.3 Small Examples

9.3.1 FM-GHOST

This fragment, which is contained in appendix B.1 represents the top level of a new flow modulator, similar in function to the one presented in Appendix D. The proof of this version will involve assertions about the relative times of various buffer operations. To prove these assertions, it will be necessary to pass functions of buffer histories to various executable routines. One such instance can be seen in the last parameter of the call to MODULATE_MESSAGE in MODULATOR. Figures 9-2 and 9-3 show the results of the ghost analysis for MODULATE_MESSAGE and MODULATOR. Note that the analysis also finds a number of other items which could be easily identified otherwise.

Figure 9-2: Ghost analysis for FM-GHOST

```
Ghost and non ghost objects from unit MODULATE_MESSAGE

  Ghosts:
    Formal Parameters
         CONSOLE.*INFROM
         DISPLAY.*OUTTO
         LOG.*OUTTO
         SINK.*OUTTO
         SOURCE.*INFROM
         TIME
    Local variables and constants
         CONSOLE', CONSOLE'.*INFROM, CONSOLE'.*MSGS, CONSOLE'.*OUTTO
         DISPLAY', DISPLAY'.*INFROM, DISPLAY'.*MSGS, DISPLAY'.*OUTTO
         LOG', LOG'.*INFROM, LOG'.*MSGS, LOG'.*OUTTO
         SINK', SINK'.*INFROM, SINK'.*MSGS, SINK'.*OUTTO
         SOURCE', SOURCE'.*INFROM, SOURCE'.*MSGS, SOURCE'.*OUTTO
         STATE'

  Non ghosts:
    Formal Parameters
         CONSOLE, CONSOLE.*MSGS, CONSOLE.*OUTTO
         DISPLAY, DISPLAY.*INFROM, DISPLAY.*MSGS
         LOG, LOG.*INFROM, LOG.*MSGS
         MYID
         SINK, SINK.*INFROM, SINK.*MSGS
         SOURCE, SOURCE.*MSGS, SOURCE.*OUTTO
         STATE
         TABLE, TABLE[*], TABLE[*][*]
    Local variables and constants
         C
         S
```

Note that the formal parameter TIME in MODULATE_MESSAGE is identified as a ghost. The corresponding actual in MODULATOR is an expression involving a function call. In performing the optimization to eliminate the ghost, it is necessary to examine the call of MODULATE_MESSAGE to discover that the formal TIME is a ghost. This justifies eliminating the actual in MODULATE_MESSAGE, and hence the evaluation of the call to MAX_TIME_STAMP.

Figure 9-3: Ghost analysis for FM-GHOST - concluded

```
Ghost and non ghost objects from unit MODULATOR

 Ghosts:
  Formal Parameters
       MYID
  Local variables and constants
       CONSOLE', CONSOLE'.*INFROM, CONSOLE'.*MSGS, CONSOLE'.*OUTTO
       DISPLAY', DISPLAY'.*INFROM, DISPLAY'.*MSGS, DISPLAY'.*OUTTO
       LOG', LOG'.*INFROM, LOG'.*MSGS, LOG'.*OUTTO
       SINK', SINK'.*INFROM, SINK'.*MSGS, SINK'.*OUTTO
       SOURCE', SOURCE'.*INFROM, SOURCE'.*MSGS, SOURCE'.*OUTTO

 Non ghosts:
  Formal Parameters
       CONSOLE, CONSOLE.*INFROM, CONSOLE.*MSGS, CONSOLE.*OUTTO
       DISPLAY, DISPLAY.*INFROM, DISPLAY.*MSGS, DISPLAY.*OUTTO
       LOG, LOG.*INFROM, LOG.*MSGS, LOG.*OUTTO
       SINK, SINK.*INFROM, SINK.*MSGS, SINK.*OUTTO
       SOURCE, SOURCE.*INFROM, SOURCE.*MSGS, SOURCE.*OUTTO
       TABLE, TABLE[*], TABLE[*][*]
  Local variables and constants
       STATE
```

9.3.2 Information flow analysis of SNEAK

The following example is taken from security information flow analysis. The value sent to the output buffer of main is the same as the value received from the input buffer. The example is a contrived one and merely serves to illustrate that information can be transmitted through the blockage of Gypsy buffers or similar objects. The necessity of dealing with this type of transfer is one of the factors which makes the information flow analysis so expensive as it introduces a potential information flow into all execution paths following the potential blockage. If we could prove that certain operations would never block, much of this potential flow would be eliminated.

A careful analysis of the example will show that the program, as it stands is correct and that the output will be the same as the input. If the program is modified slightly, by inserting

```
SEND 1 TO SYNCH;
```

just before the COBEGIN in the main program, the information flow analysis will remain unchanged, but the actual output of the program will depend on the scheduler for the concurrent processes rather than the value read from the input buffer.

Figure 9-4 shows the information flow for the main routine of SNEAK. The code is contained in appendix B.2. Note that the analysis successfully tracks the flow from the messages component of the input buffer to the messages component of the output buffer.

Figure 9-4: Information flow for MAIN::SNEAK

```
Normal return:
  Var parameter component INP.*INFROM obtains information from:
       Parameters: INP.*INFROM
       Via intermediate routines: SENDER::SNEAK
  Var parameter component INP.*OUTTO obtains information from:
       Parameters: INP.*OUTTO
       Via intermediate routines: SENDER::SNEAK
  Var parameter component INP.*MSGS obtains information from:
       Parameters: INP.*MSGS
       Via intermediate routines: SENDER::SNEAK
  Var parameter component OUTP.*INFROM obtains information from:
       Parameters: INP.*MSGS, OUTP.*INFROM
       Literals: 0, 1
       Via intermediate locals: NONPOS.*MSGS, POS.*MSGS,
                                SYNCH.*MSGS, TAKE.*MSGS
       and routines: HELPER::SNEAK, RECEIVER::SNEAK,
                     SENDER::SNEAK
  Var parameter component OUTP.*OUTTO obtains information from:
       Parameters: OUTP.*OUTTO
       Via intermediate routines: RECEIVER::SNEAK
  Var parameter component OUTP.*MSGS obtains information from:
       Parameters: INP.*MSGS, OUTP.*MSGS
       Literals: 0, 1
       Via intermediate locals: NONPOS.*MSGS, POS.*MSGS,
                                SYNCH.*MSGS, TAKE.*MSGS
       and routines: HELPER::SNEAK, RECEIVER::SNEAK,
                     SENDER::SNEAK

Abnormal return signaling ROUTINEERROR does not occur under
         the analysis assumptions.
```

Chapter 10
A PRODUCTION QUALITY COMPILER FOR A VERIFIABLE LANGUAGE

Any program together with the semantics of the language in which it is written, constitutes a specification for a computation. The object of code generation is to realize an implementation of that specification which will perform the computation. Often the semantics of the language only partially define the results of certain operations in the language. In such a case, we are satisfied when the implementation produces the result predicted by the program for only those cases where the semantics indicate that a result should be produced. This leaves the implementor a good bit of freedom since, in general, the user does not care about the undefined behavior of the program.

Verified programs and languages intended for the writing of verified programs place additional constraints on the implementor of the language. A program which is to be verified will have a separate set of formal specifications which reflect those aspects of the program behavior which are to be verified. Verification consists of generating and proving a set of theorems that demonstrate the equivalence between the program and the specification. Either partial or total correctness may be sought. For partial correctness, it is necessary to prove that; if the program terminates normally,[23] its result is that indicated by the specification. For total correctness, it is also necessary to show that the program, in fact, terminates normally.

If we consider the specifications to be complete, i.e., they totally characterize the desired behavior of the program, then it is simply necessary to assure that the implementation is consistent with the specification in the sense of partial or total correctness. Under this assumption, the program text is a guide for the implementation. The implementation need only contain enough to support the specifications and need not support the entire program text. Since program specifications are not usually complete, it is necessary for the implementation to reflect the program behavior and to maintain consistency with the specifications. The remainder of this section will outline an optimization and code generation scheme which preserves rigorous semantics of a verifiable language while permitting the application of modern optimization and code generation techniques.

[23]Some languages such as Gypsy allow specifications to be provided for some causes of abnormal termination. For such programs, we will include all specified terminations under the umbrella of "normal".

10.1 Overview

The basic optimizer and code generator is patterned on the PQCC [Wulf 80] model. This model views code generation and optimization as a transformation of a high level, abstract, program representation into a concrete implementation by the successive and possibly alternative application of a series of small transformations undertaken by a series of "expert" processes. In this case, the experts are constrained by the need to preserve certain aspects of the program behavior under exception signaling. In assuring that the transformations are applicable, the "experts" may call upon other facilities usually found in a verification environment such as a powerful simplifier and a theorem prover.

We assume an integrated programming and specification language so that the internal representation of the program contains both the executable program text and the specifications. In addition, the internal representation makes explicit all aspects of the program including exception sites, exception handlers, and allocation and initialization of local variables. The internal representation must be rich enough to include the results of program transformations such as parameter globalization and expansion of routines at call sites even though the corresponding constructs are not representable in the source language.

While occasional reference will be made to Gypsy, the strategy is, in general, language independent and could apply to other verifiable languages. In particular, an implementation of a verifiable subset of Ada could use an identical approach.

10.2 Organization

The proposed optimizer and code generator is organized into a sequence of transformers which pass over the program representation altering its form as they are applied. Some of the transformers reorder the abstract form, others produce code sequences for a node in the abstract representation, while still others may transform the code.

At the start of the optimization and code generation process, we assume that the internal representation of the program to be processed is that produced by a parser for the language in question and still reflects the structure and organization of the program written by the programmer. In addition to the program text, we assume that verification conditions for the program have been generated and proved and that optimization conditions for the suppression of exception tests have been generated. Links between the program representation and the VCs and OCs exist so that the information contained in these forms can be used in subsequent proof processes as code transformations proceed. The proof status of the individual OCs is not specified. The subsequent transformation and code generation process is, in part, dependent on the status of the OCs.

In the discussion which follows, the general form of the PQC described by Wulf in [Wulf 80] is followed with emphasis on the modifications and constraints introduced by verification and exceptions.

10.2.1 The Machine Independent Optimizers

We envision first a series machine independent optimization phases which attempt to perform such optimizations as common subexpression evaluation, removal of loop invariant code from loop bodies, dead code elimination, constant expression evaluation, and strength reductions. Expansion of routine bodies at call sites may also be done. Note that many of these optimizations may interact and it may be desirable to iterate over a cycle of phases. These phases are constrained by the presence of exceptions, exception handlers, and OCs in several ways.

1. Exception paths can reduce the size of regions in which code motion can take place. For this reason, it is desirable to prove as many OCs as possible prior to these optimizations.

2. The presence of handlers inhibits code motion as discussed in section 4.7. The code motion routines must be aware of these constraints which can be eased by the use of global dataflow analysis or by relaxing the semantics of the language with PRESERVE constructs as discussed in section 4.7.1.

3. Optimizations which change the operators used in the code must prove that the new construct has equivalent exception semantics. This may require the construction and proof of additional verification or optimization conditions. The transformed VCs and OCs may use the original proved VCs and OCs to facilitate their proofs. Interaction with a theorem prover may be required here.

10.2.2 Code Generation Phases

Our proposed code generation scheme closely follows that of the PQC with the addition of the constraints imposed by exceptions and handlers. A code generator generator similar to that described in section 8 of [Wulf 80] can generate the necessary code patterns. In our case, it is necessary to generate patterns for both the cases in which it is necessary to check for exception conditions and those in which it is not. This will result in a much larger number of code patterns that would be necessary if exceptions did not need to be considered. For simple patterns involving a single operator with a single possible exception, two versions of each possible code sequence may be derived. For higher level language constructs involving numerous potential exceptions, the problem is much worse, since, in addition to patterns in which every exception is checked and in which no exceptions are checked, all the combinations in between could be considered. More work is required to develop techniques for choosing an effective set of code patterns for a given language and machine.

The algebraist for a verifiable language PQC may be similar to that of section 7 of [Wulf 80], but it must take into account the possible introduction of exceptions due to the reordering of the code. This phase would have access to the VC and OC information within the internal representation of the program to aid in justifying its actions. As in the machine independent phases, recourse to an automatic or interactive theorem prover may be necessary.

The temporary allocator (*ibid* section 5)is also constrained by the possibility of exceptions. In Gypsy, the exception semantics forbid us to destroy a variable as a side effect of a computation which subsequently issues an exception. The temporary allocator must therefore avoid proposing code sequences which use one of the variables in the computation to hold the result of the computation if any exceptions are possible after that location has been altered. This may further be complicated by the fact that variables are sometimes preserved in more than one location, *e.g.* a variable may be held both in a register and in a memory location. In such a case, it could be permitted to destroy the register copy if the exception handler knew to use the memory copy.

Finally the actual choice of an effective code sequence (*ibid* section 5) from those proposed is constrained by the need to match checked and unchecked operations. Note that we can always match a code pattern involving exception checks to a computation which specifies that no exceptions will occur with a possible loss of efficiency. This may minimize costs when a suitable unchecked pattern is not available to complete a larger match.

10.3 Summary

A PQC for a verifiable language can be built following the model given by Wulf. A number of phases of the compiler may require the services of a theorem prover. At first glance, this may appear to be an undue burden on the user of the system, but experience with the Gypsy system has shown that the proofs involved in the optimization process require only a small percentage of the time that verification of the same program required. These techniques are generally used only when the risks of program failure and the need for performance justify them. The time required to use them should be considered in that light.

Chapter 11
CONCLUSIONS

In this dissertation, we have demonstrated a number of optimization techniques which seem to be particularly applicable to languages which are intended to support verification. We have shown that it is possible to compensate, for the most part for the rigid semantics of verifiable languages. In particular, we have shown that reducing the potential domination of verifiable programs by exception paths is tractable using a combination of static specifications, i.e. strong typing, and dynamic specifications, i.e. user supplied assertions. While proofs of the optimization conditions which result from this approach are expensive in terms of both manpower and machine cycles, the time involved is small compared to that used in producing a verified program. In addition, we have shown that the dominant structural characteristic of verified programs, numerous function and procedure calls, is subject to modifications during code production which will significantly reduce the costs of the call operations. The machine time required for these analyses is small compared to the time required for code generation.

The utility of eliminating proof only objects from the code remains to be conclusively demonstrated. There are indications that the availability of the technique will alter the style of verifiable programs.

11.1 Influence on Gypsy and the GVE

The optimization effort reported here has had, and continues to have, a substantial impact on the design of Gypsy and its implementation, the GVE. Areas of influence include:

- Changes in the parameter passing syntax of Gypsy to permit the passage of actual condition parameters to any operation which may raise an exception.
- Elimination of the default injection of formal condition parameter names into calling environments.
- Enhancements in the symbolic evaluator component of the GVE to increase its ability to deal with inequalities and subranges.
- Improvements in the provers typelist and subrange mechanisms to take advantage of knowledge concerning sequence sizes, variable ranges, etc.
- Introduction of a uniform dialect of prefix (the GVE internal representation of Gypsy) to permit all components of the GVE to exchange information.

11.2 Directions for Future Research

The present work has demonstrated the feasibility of a number of techniques for the generation of efficient code from verifyable languages. The techniques themselves can be extended in a number of directions. The concepts behind the techniques have applicability in other areas, some of which may not be immediately obvious.

11.2.1 Additional Implementation Efforts

An obvious extension of the present work is to apply the techniques to a more complete implementation, one which directly produces machine code for a verifyable language. A vehicle such as the PQC described in chapter 10 appears to be suitable for these investigations. There are a number of research topics to be considered during the course of a PQC based implementation. These include:

- Investigation of appropriate proof stratigies for use during code sequence selection.

  The code sequence selection process in the PQC considers a number of possible code sequences in parallel and chooses the most efficient one based on a cost metric which is influenced by the context of the operation. Our version of this model includes code sequences containing exception checking code as well as sequences without exception checks. The latter are guarded by optimization conditions. To use an unchecked code sequence, it is necessary to prove its associated OC. Effective and efficient **automatic** proof procedures are required if such an implementation is to be useful. It is suspected that strategies for proving most of these OCs can be developed.

- Application of conventional optimization techniques under the constraints imposed by the presence of exception handlers.

  Handlers act as barriers to the movement of code. The program state expected along the execution path through the handler and beyond further constrains the code transformations which can be made within the scope of the handler. Development of a formalism to describe these constraints and techniques for proving that proposed code transformations satisfy the constraints should be fruitful areas for research.

- More quantitative techniques for routine expansions.

  The development of a model, parameterized with respect to the cost of the call, the call site, the called routine, and the target machine would be useful. Such a model could aid in determining when to replace a routine call by an expansion.

- Interprocedural extension of dead and useless code elimination techniques.

  The elimination of proof only objects discussed in chapter 8 could be viewed as an extension of the local optimizations commonly used to eliminate dead and/or useless code. Extending these techniques might provide ways to eliminate proof only objects with a relatively small additional analysis expense.

11.2.2 Other Proof Based Analyses

The use of proof techniques to show that exceptions cannot occur is central to the present work. Other aspects of program behavior are also subject to analysis using proof techniques. This section considers several areas where further investigation might prove fruitful. These include information flow analysis for security policy enforcement and performance analysis.

11.2.2-A Information Flow Security Analysis

A rigorous treatment of information flow for security analysis yeilds flows from almost every object to almost every object when exceptions and process blockages are considered. The resulting flows are almost useless for demonstrating that a program conforms to a given security policy. Using the techniques developed in this work for dealing with exceptions and extending them to eliminate certain potential blockage based information flows should produce a more useful information flow analysis.

Conventional security information flow models assign a fixed security level to each program object and attempt to show that there is no potential information flow from a high security object to one of a lower security level. This type of analysis cannot be applied to programs, such as the message flow modulator contained in appendix D because program objects may contain values of various security levels and the program and information flow are controlled by the security level of these values. In such programs, it should be possible to construct theorems to guard specific information flow paths and to use them to prove that the actual flow along the a path conforms to the security policy even if the normal analysis admits a potential violation of the policy.

11.2.2-B Proof of Performance

Accurate estimates of program performance are highly desirable. Analysis of the algorithms embodied in programs can provide approximate estimates of program performance but these are often inadequate. Analysis of the object code for a path through a program can provide an accurate estimate of the execution time for the path if it is carried out with respect to a suitable timing model for the target machine. The large numbers of potential paths and the combinatoric problems of using the path segments to describe a program execution render this type of timing analysis impractical for large programs (but see [McHugh 75]). By combining the type of path tracing used during VC generation, with object code analysis for the paths thus produced, and generating theorems to define the conditions under which a given combination of paths would be executed, it may be possible to provide detailed and accurate performance predictions under certain circumstances.

Appendix A
Operator Definition Functions

```
{This file contains operator and builtin function definitions for use
in specifying PDP-11 definitions.  Note that only the abnormal exit
specs need be given for exception suppression purposes of machine
independent operations since the normal cases are assumed as rewrite
rules in Xeval. This is not the case for machine dependent operations
as we may want to assume the normal exits of such functions to
simplify the proofs of such things as VALUEERROR vcs. }

{These definitions assume that typenames may be used as parameters.
 the following name conventions will be used.
        type__          typename parameters
        type_           type of objects with this type
                        (to avoid semantic errors)
        typename_       the type of typename parameters
        basetype_       the name of the basetype of type_
        element_        the type of an arbitrary array, sequence,
                        or buffer element
        array_          an arbitrary array
        sequence_       an arbtrary sequence
        buffer_         an arbitrary buffer
}

{ TO USE THIS FILE AS A BASIS FOR THE GENERATION AND PROOF OF
  OPTIMIZATION VCS, THE FOLLOWING PROCESS MUST BE FOLLOWED.
    1. PARSE a file containing a scope definition for the scope
       'IMPLEMENTATION'. This must contain the exception
       specification routines named below.
    2. PARSE this file.
    3. Enter LISP and EDIT the global variable 'DATABASE'. The best
       way to do this is to use two window mode with this file in
       the other window. It is necessary to edit each routine which
       uses TYPE__ as a formal parameter name. The editing consists
       of substituting TYPE__ for TYPE_::PREDEFINED at every prefix
       occurance coresponding to the source occurances of TYPE_{_}
{   4. Save the resulting database.
    5. RESTORE the database before parsing files for which
       optimization involving exception suppression will be
       required. }


scope predefined = begin

{First name in the exception specification routines from the
implementation dependent scope IMPLEMENTATION }

        name    integer_baddivide, integer_badadd,
                integer_badmultiply, integer_badminus,
                integer_badsubtract, integer_badpower
        from    IMPLEMENTATION;

{Then define some types and leave them hanging as much as possible. }

        type basetype_ = pending;

        const lower_:basetype_ = pending;
        const upper_:basetype_ = pending;
        type type_ = basetype_[lower_..upper_];

        type typename_ = pending;

        type element_ = pending;

        type array_ = array (type_) of element_;

        type sequence_ = sequence of element_;

        type buffer_ = buffer of element_;

{Now define functions for all implemented gypsy operators which can
signal predefined conditions at runtime}

{    pred(x) }       function pred_(x:type_; type__:typename_):type_
                                  unless (nopred)=
                     begin
                        exit case
                           (is nopred:  x = lower(type_{_}));
                     end;

{   succ(x)    }    function succ_(x:type_; type__:typename_):type_
                          unless (nosucc)=
                     begin
                        exit case
                          (is nosucc: x = upper(type_{_}));
                     end;

 {   scale(x,s) }    function scale_(k:integer;
                                    type__:typename_): basetype_
                            unless (underscale, overscale) =
                     begin
                        exit case
                           (is underscale: k<0;
                            is overscale:  ord (upper(type_{_}))<k);
                     end;

{    x div y     }       function integer_divide(x,y:integer):integer
                                   unless (divideerror, zerodivide)=
                        begin
                            exit case
                                 (is normal:
                                     (y ne 0) and
                                     (not integer_baddivide(x,y));
                                  is divideerror:
                                     integer_baddivide(x,y);
                                  is zerodivide:
                                     y=0);
                        end;


{    x + y    }           function Integer_add (x,y:Integer):Integer
                                       unless (adderror) =
                          begin
                            exit case (is normal:
                                          not Integer_badadd(x,y);
                                       is adderror:
                                          Integer_badadd(x,y));
                          end;

{    x * y     }      function Integer_multiply
                                     (x,y:Integer):Integer
```

```
                                          unless (multiplyerror) =
                          begin
                            exit case (is normal:
                                          not Integer_badmultiply(x,y);
                                      is multiplyerror:
                                          Integer_badmultiply(x,y));
                          end;

{   -x        }             function Integer_minus (x:Integer):Integer
                                          unless (minuserror) =
                            begin
                              exit case (is normal:
                                          not Integer_badminus(x);
                                      is minuserror:
                                          Integer_badminus(x));
                            end;

{   x - y     }           function Integer_subtract
                                      (x,y:Integer) : Integer
                                      unless (subtracterror) =
                          begin
                            exit case (is normal:
                                      not Integer_badsubtract(x,y);
                                  is subtracterror:
                                      Integer_badsubtract(x,y));
                          end;

{   x ** y    }           function Integer_power (x,y:Integer) : Integer
                                      unless (powererror,
                                              negativeexponent,
                                              powerindeterminate) =
                          begin
                             exit case
                               (is normal:
                                   (y > 0 and x > 0 and
                                   not Integer_badpower(x,y));
                                is powererror: y > 0 and
                                   Integer_badpower(x,y);
                                is negativeexponent:
                                   y < 0;
                                is powerindeterminate:
                                   x = 0 and y = 0);
                          end;

{   a [i]       }         function array_select(a:array_; i:basetype_;
                                              lb,ub:type_):element_
                                      unless (indexerror) =
                               begin
                                 exit case
                                   (is indexerror:
                                       not i in [lb..ub]);
                          end;

{   a with ([i]:=x)  } function array_componentassign
                                (a:array_; i:basetype_; x:element_;
                                 lb,ub:type_): array_
                            unless (indexerror) =
                           begin
                             exit case
                                  (is indexerror:
                                         not i in [lb..ub]);
                           end;
```

```
{    S[i]    }          function sequence_select
                                  (S: sequence_;
                                   i: integer;
                                   lhs: boolean): element_
                                  unless (indexerror) =
                             begin
                               exit case
                                    (is indexerror:
                                         if lhs
                                            then
                                              not i in [0..size(S)+1]
                                            else
                                              not i in [1..size(S)]
                                            fi);
                             end;

{    S[i..j] }          function sequence_subselect
                                        (S: sequence_;
                                         i,j: integer): sequence_
                                        unless (indexerror) =
                             begin
                               exit case
                                 (is indexerror:
                                        not (i le j+1
                                               or i in (1..size(S) + 1)
                                               or j in (0..size(S))));
                               end;

{    s with([i]:=x) }  function sequence_componentassign
                                         (s:sequence_;
                                          i:integer;
                                          x:element_):sequence_
                                         unless (indexerror,
                                                    spaceerror) =
                                 begin
                                   exit case
                                      (is indexerror:
                                           not i in [1..size(s)]);
                                 end;

     procedure buffer_send_T(x:element_; var B:buffer_;
                                   a:activationid)
                        unless(senderror) =
     begin
        block full(B);
        exit case
              (is normal: outto(B,a) = outto(B',a) <: X
                        and infrom(B,a) = infrom(B',a);
              is senderror: outto(B,a) = outto(B',a)
                           and infrom(B,a) = infrom(B',a));
     end;

     procedure buffer_receive(var x:element_; var B:buffer_;
                                   a:activationid)
                         unless(receiveerror) =
     begin
        block empty(B);
        exit case
              (is normal: infrom(B,a) = infrom(B',a) <: x
                        and outto(B,a) = outto(B',a);
              is receiveerror: infrom(B,a) = infrom(B',a)
                              and outto(B,a) = outto(B',a));
     end;

end;

scope IMPLEMENTATION = begin

{Specification functions for a PDP-11 implementation of Gypsy
 assuming the use of the hardware provided 16 bit twos complement
 arithmetic for the gypsy operators / + * - - ** }

        Const MAXINT:integer = 32767;
        Const MININT:integer = -32768;

        Function integer_baddivide(x, y:integer): boolean =
                { divideerror } begin
                { No divide error is possible in 16 bits }
                exit (assume result =( false));
        end;

        Function integer_badadd(x, y:integer): boolean =
                { adderror } begin
                exit (assume result =( not (x+y) in [MININT..MAXINT]));
        end;

        Function integer_badmultiply(x, y:integer): boolean =
                { multiplyerror } begin
                exit (assume result =( not x*y in [MININT..MAXINT]));
        end;

        Function integer_badminus(x:integer): boolean =
                { minuserror } begin
                exit (assume result = (x = MININT));
        end;

        Function integer_badsubtract(x, y:integer): boolean =
                { subtracterror } begin
                exit (assume result =( not x-y in [MININT..MAXINT]));
        end;

        Function integer_badpower(x, y:integer): boolean =
                { powererror } begin
                exit (assume result =((not x=0 and y=0)
                      and (not y<0)
                      and not x**y in [MININT..MAXINT]));
        end;

 end
```

Appendix B
Small Examples

This appendix contains small examples and fragments of Gypsy programs which serve to illustrate various optimization techniques.

B.1 A flow modulator fragment

Only the executable routines are included here.

```
scope flow_modulator =
begin

procedure modulator (var  source : a_source_buffer<input>;
                     var    sink : a_sink_buffer<output>;
                     var console : a_console_buffer<input>;
                     var display : a_display_buffer<output>;
                     var     log : a_log_buffer<output>;
                           table : a_modulator_constant)
        unless (incorrect_table) =
begin
  exit case (
    is normal:
      modulated_flow (xinfrom(source,myid), xinfrom(console,myid),
                      outto(sink,myid), outto(display,myid),
                      outto(log,myid), initial_state(table), table);
    is incorrect_table:
      no_modulated_flow (infrom(source,myid), infrom(console,myid),
                         outto(sink,myid), outto(display,myid),
                         outto(log,myid), table));
  var state: a modulator_state;
  initialize (state, display, log, table) unless (incorrect_table);
  loop
    assert   modulation (xinfrom(source,myid),
                         xinfrom(console,myid),
                         outto(sink,myid),
                         outto(display,myid),
                         outto(log,myid),
                         initial state(table), state, table)
           & correct_table(table);
    if stop_modulation (state) then leave
    else modulate_message (source, sink, console, display,
                           log, state, table,
                           max_time_stamp (xinfrom(source,myid),
                                           xinfrom(console,myid),
                                           beforetime))

    end;
  end;
 end;

 procedure modulate_message (var  source : a_source_buffer<input>;
                             var    sink : a_sink_buffer<output>;
                             var console : a_console_buffer<input>;
                             var display : a_display_buffer<output>;
                             var     log : a_log_buffer<output>;
                             var   state : a_modulator_state;
                                   table : a_modulator_constant;
                                    time : integer) =
begin
  entry correct_table(table); {and TIME is earlier than the present}
  exit
    (prove modulation (xinfrom(source,myid), xinfrom(console,myid),
                       outto(sink,myid), outto(display,myid),
                       outto(log,myid), state', state, table);
     assume input_times (xinfrom(source,myid),
                         xinfrom(console,myid), time));
  var s: a_source_object;
  var c: a_console_object;
  receive s from source;
  send s to display;
  receive c from console;
  send c to display;
  if c = escape then state := quit
  elif c = null_character then
    send s to log;
    send c to log;
  else send table[s,c] to sink
  end;
end;

procedure initialize (var   state : a_modulator_state;
                      var display : a_display_buffer<output>;
                      var     log : a_log_buffer<output>;
                            table : a_modulator_constant)
        unless (incorrect_table) =
begin
  exit case (
    is normal:   state = initial_state (table)
               &   outto(display,myid)
                 = initial_display (state, table)
               & outto(log,myid) = initial_log (state, table)
               & correct_table (table);
    is incorrect_table:     outto(display,myid)
                          = table_error_display(table)
                        & outto(log,myid) = table_error_log(table)
                        & not correct_table(table));
  var c: character := null_character;
  state := run;
  send formfeed to display;
  send formfeed to log;
  loop
    if table[c,c] ne c then signal incorrect_table
    elif c ne upper(character) then c := succ(c)
    else leave
    end;
  end;
end;

function escape : character =
begin
  result := scale (27, character);
end;

function formfeed : character =
begin
```

```
  pending;

function null_character : character =
begin
  result := scale (0, character);
end;

function stop_modulation (state : a_modulator_state) : boolean =
begin
  result := state = quit;
end;

const beforetime : integer = pending;

type an_input_seq = sequence of an_input_event;
type a_char_array = array (character) of character;
type a_console_buffer = buffer (console_buffer_size) of a_console_object;
type a_display_buffer = buffer (display_buffer_size) of a_display_object;
type a_log_buffer = buffer (log_buffer_size) of a_log_object;
type a_modulator_constant = array (character) of a_char_array;
type a_modulator_state = (run, quit);
type a_sink_buffer = buffer (sink_buffer_size) of a_sink_object;
type a_source_buffer = buffer (source_buffer_size) of a_source_object;
type a_timed_console_seq = sequence of a_timed_console_object;
type a_timed_source_seq = sequence of a_timed_source_object;

end; {scope flow_modulator}
```

B.2 Sneak

```
Scope Sneak = begin

Procedure Main (var  inp: in_buf<input>;
                var outp: out_buf<output>) =
begin
  var pos, nonpos, synch, take : sig_buf;
  cobegin
    Sender( inp, pos, synch, nonpos);
    Helper( pos, take, synch, 1);
    Helper( nonpos, take, synch, 0);
    Receiver( take, outp);
  end;
end;

Procedure Sender (var        inp: in_buf<input>;
                  var pos,nonpos: sig_buf<output>;
                  var      synch: sig_buf<input>) =
begin
  var x,s: bit;
  receive x from inp;
  if x > 0 then
    send 0 to pos;
    receive s from synch;
    send 0 to nonpos;
  else
    send 0 to nonpos;
    receive s from synch;
    send 0 to pos;
  end;
  receive s from synch;
end;
```

```
Procedure Helper (var   sig: sig_buf<input>;
                  var  take: sig_buf<output>;
                  var synch: sig_buf<output>;
                          c: bit) =
begin
  var s: bit;
  receive s from sig;
  send c to take;
  send 0 to synch;
end;

Procedure Receiver (var take: sig_buf<input>;
                    var outp: out_buf<output>) =
begin
  var r: bit;
  receive r from take;
  receive r from take;
  if r = 1 then
    send 0 to outp;
  else
    send 1 to outp
  end;
end;

Type in_buf = buffer (1) of bit;
Type out_buf = buffer (1) of bit;
Type sig_buf = buffer (1) of bit;
Type bit = integer[0..1];

end;
```

Appendix C
Chase: A Computer Game

C.1 Analyzed version: MAIN and PLAY

```
{ Game of CHASE for the LSI-11
        written in GYPSY
                by Dwight Hare
                        10/3/78         }

{ Modified by J. McHugh to include loop assertions for
   optimization and to use types related to the
   implementation. }

Scope Chase =
begin
{

                Data Representation

NOTE:  It is important to keep the data representation consistent.

BOARD:

        Every board position has its contents (either +, #, X, $,
or blank) and a number corresponding to the array index of the
robot in that position if there is a robot there.  This number
is not valid if the contents is not either a '+' or '#'.

ROBOTS:

        The array of robots has information about each robot.
First is an indicator ("alive") which tells whether or not this
robot is still on the board.  If the robot is dead, the
coordinates are not valid.  The coordinates give the x and y
coordinate of the robot.

Consistency:            ( which should be proved using specs!! )

        If robot [i] is alive with coordinates (x,y) then
 board[x,y].contents = '+ and board[x,y].robot_num = i.
Likewise, if position (x,y) has a '+' or a '#' in that
position  then robot[board[x,y].robot_num] is alive and has
coordinates (x,y).

        Don't forget the display!  It must remain consistent
with the internal representation (a more interesting
specification problem too!).  Whenever anything like shooting,
killing, etc is done, you must update the appropriate status
and the display too.

Direct any questions to Ken Shotting or Dwight Hare (at CMP) }

{**********}

Procedure Main (var ttyin, ttyout : tty) =
        { required parameters for compiler, first is input from
                tty, second is output to tty }
begin
  var human_won, robots_won : machine_int;
  var board : board_type;
  var stat : stat_type;
  var random : random_seed;
               { this is the current random seed,
                 carried everywhere }
  var success : boolean;
  var term: a_term_type;
  cond human_dead, robots_dead;

  intro_init (stat.level, random, term, ttyin, ttyout);
  stat.term_type := term;
 loop
  clear_screen(term,ttyout);
  initialize_board (board, stat, random);
  print_board (board, term, ttyout);
  print_stat (stat, ttyout);
  begin
     play (board, stat, random, ttyin, ttyout)
          unless (human_dead, robots_dead);
  when
    is robots_dead :
        position(x_width + 4, 1, term, ttyout);
        Print (
         "Congratulations - you have survived the merciless attack!",
         false, ttyout);
        human_won := human_won + 1;

    is human_dead :
        position(x_width + 4, 1, term, ttyout);
        Print ("Gotcha!   You have lost again, insignificant human!",
               false, ttyout);
        robots_won := robots_won + 1;

    else : Position (x_width + 4, 1, term, ttyout);
           Print ("Error signalled - tell Dwight", false, ttyout);
    end;

    main_finish (ttyin, ttyout, stat, success,
                 human_won, robots_won);

  end; { non terminating loop }
 end;  { of Procedure Main }


 procedure main_finish (var ttyin,ttyout : tty; var stat: stat_type;
                        var success:boolean;
                        human_won, robots_won : machine_int) = begin
   var count : machine_int;
   CRLF (ttyout);
   CRLF (ttyout);

   Print ("Humans : ", false, ttyout);
   Print_number (human_won, ttyout);
   Print ("    Robots : ", false, ttyout);
   Print_number (robots_won, ttyout);
   CRLF (ttyout);
   CRLF (ttyout);

   Print ("(type 'h', 'e', or 'n' to change the level) ",
          false, ttyout);

   count := 32700;
   loop                { wait about 5 seconds for defeat or victory
```

```
                        to sink in }
   count := count - 1 * 1 div 1;       { delay computation }
   if count < 0 or not empty (ttyin) then leave end;

  end;

  if not empty (ttyin) then
      set_level (stat.level, success, ttyin, ttyout);
  end;
  end; {Procedure main_finish }

{**********}

Procedure Play (var board : board_type; var stat : stat_type;
                var random : machine_int; var ttyin, ttyout : tty)
                     unless (human_dead, robots_dead)        =
begin

      { main playing program }

  var first_char, second_char : character;
      { first character of command, and second character }
  var count, time_count : machine_int;
      { count used for time out,
        time_count is current second count }
  const null_char : character = scale(0, character);
  const term : a_term_type = stat.term_type;
  cond command;

  loop  { major loop - play the game }
    time_count := stat.max_time;     { start at the beginning time }
    play_clear_buffer (ttyin);
    first_char := null_char;
    second_char := null_char;
    Position (8, y_width + 11, term, ttyout);
    print_number (time_count, ttyout);  { print the current time}

    loop  { wait for a move, time out }
      count := second_count;

      loop

        Play_get_cmd (first_char, second_char, board,
                                  stat, ttyin, ttyout)
                    unless (command, human_dead, robots_dead);
        count := count - 1;
        if count < 0 then leave end;
        assert count ge 0;
                { end of second count }

      end;

      time_count := time_count - 1;   { else, print out new time }
      Position (8, y_width + 11, term, ttyout);
      print_number (time_count, ttyout);
      if time_count le 0 then leave end;        { ran out of time }
      assert time_count > 0;

    when
      is command :
    end;  { of wait loop }

    Play_do_cmd (first_char, second_char, board,
                              stat, random, ttyout)
                 unless (human_dead, robots_dead);

  end;  { of major loop }


end;  { of procedure play }

Procedure Play_clear_buffer (var ttyin : tty ) =
begin
  var first_char: character;
    if not empty (ttyin) then          { flush buffer }
        receive first_char from ttyin;
    end;
end;

Procedure Play_get_cmd (var first_char, second_char : character;
                        var board : board_type; var stat : stat_type;
                        var ttyin, ttyout : tty)
                         unless (command, human_dead, robots_dead) =
begin
  const null_char : character = scale(0, character);
  const bell : character = scale(7, character);

        if not empty (ttyin) then       { something has been typed }
          if first_char = null_char then  { read first character }
                receive_direction(first_char,ttyin);
                if first_char < '1 or first_char > '9
                                   or first_char = '5
                    then send bell to ttyout;   { illegal character }
                         first_char := null_char;
                end;
             else      { already got first character, get second one }
                receive second_char from ttyin;
                if second_char = scale (27, character) then
                        first_char := null_char;  { escape, ignore }
                        second_char := null_char;
                elif second_char = '. then          { shoot ! }
                        shoot_it (board, stat, ord(first_char) - 48,
                                              ttyout)
                                  unless (human_dead, robots_dead);
                        first_char := null_char;
                        second_char := null_char;  { now keep going }
                elif second_char < '0 or second_char > '9
                    then { illegal character, ignore }
                         send bell to ttyout;
                         second_char := null_char;
                else signal command
                end;
           end;
         end;
 end; { Procedure Play_get_cmd }

 Procedure Play_do_cmd (var first_char, second_char : character;
                        var board : board_type; var stat : stat_type;
                        var random: random_seed;
                        var ttyout : tty)
                        unless (human_dead, robots_dead) =
 begin
   const stat_x : machine_int =  stat.human_x;
   const stat_y : machine_int =  stat.human_y;
   const null_char : character = scale(0, character);
     if second_char = null_char or second_char = '0
```

```
        { robots move now }
        then update (board, stat, random, ttyout)
                    unless (human_dead, robots_dead)
    else move_object(board, stat,stat_x , stat_y,
                     ord(first_char) - 48,
                     ord(second_char) - 48,  ttyout)
                                unless (human_dead, robots_dead);
         update (board, stat, random, ttyout)
                 unless (human_dead, robots_dead);
                        { robots get their chance }
    end;

end; { Procedure Play_do_cmd }

end;  { of scope Chase }
```

C.2 Original version: PLAY

```
Procedure Play (var board : board_type; var stat : stat_type;
                var random : int; var ttyin, ttyout : tty)
                        unless (human_dead, robots_dead)        =
begin

        { main playing program }

  var first_char, second_char : character;
        { first character of command, and second character }
  var count, time_count : int;
        { count used for time out, time_count is current second count }
  const null_char : character = scale(0, character);
  const bell : character = scale(7, character);
  const term : a_term_type = stat.term_type;
  cond command;

  loop  { major loop - play the game }
    time_count := stat.max_time;        { start at the beginning time }
    if not empty (ttyin) then           { flush buffer }
        receive first_char from ttyin;
    end;
    first_char := null_char;
    second_char := null_char;
    Position (8, y_width + 11, term, ttyout);
    print_number (time_count, ttyout);  { print the current time}

    loop  { wait for a move, time out }
      count := second_count;

      loop
        if not empty (ttyin) then        { something has been typed }
          if first_char = null_char then  { read first character }
                receive_direction(first_char,ttyin);
                if first_char < '1 or first_char > '9 or first_char = '5
                    then send bell to ttyout;   { illegal character }
                         first_char := null_char;
                end;
            else        { already got first character, get second one }
                receive second_char from ttyin;
                if second_char = scale (27, character) then
                        first_char := null_char;  { escape, ignore both }
                        second_char := null_char;
                elif second_char = '. then        { shoot ! }
```

```
                   shoot_it (board, stat, ord(first_char) - 48,
                                  ttyout)
                                 unless (human_dead, robots_dead);
                   first_char := null_char;
                   second_char := null_char;  { now keep going }
               elif second_char < '0 or second_char > '9
                 then send bell to ttyout;
                    { illegal character, ignore }
                    second_char := null_char;
               else signal command
               end;
            end;
         end;

         count := count - 1;
         if count < 0 then leave end;
               { end of second count }

        end;

        time_count := time_count - 1;   { else, print out new time }
        Position (8, y_width + 11, term, ttyout);
        print_number (time_count, ttyout);
        if time_count = 0 then leave end; { ran out of time }

      when
        is command :
      end;  { of wait loop }

      if second_char = null_char or second_char = '0
          { robots move now }
          then update (board, stat, random, ttyout)
                    unless (human_dead, robots_dead)
      else move_object(board, stat,0+stat.human_x, 0+stat.human_y,
           { the 0+ on the previous line is to stop aliaserror with stat.}
                     ord(first_char) - 48,
                     ord(second_char) - 48,  ttyout)
                           unless (human_dead, robots_dead);
           update (board, stat, random, ttyout)
                         unless (human_dead, robots_dead);
                   { robots get there chance }
      end;

    end;  { of major loop }

  end;  { of procedure play }
```

Appendix D
Flow Modulator: A Data Filter

D.1 The Heart of the Message Filter

```
procedure filter_msg (          m : a_msg;
                           at_bom : boolean;
                           at_eom : boolean;
                           filter : a_filter;
                               fc : a_filter_index;
                     var pass_it : boolean;
                     var     log : a_log_buffer<output>) =
begin
  entry   correct_filter_index (fc, filter)
        & correct_msg (m, at_bom, at_eom);
  block partial_string (outto(log,myid), filter_report(m,at_bom,
                                                        filter));
  exit   pass_it = pattern_filter (m, at_bom, filter)
       & outto(log,myid) = filter_report (m, at_bom, filter);
  var fm: a_msg;
  var st: an_int;
  pass_it := true;
  normalize_msg (fm, m, at_bom, at_eom);
  st := 1;
  loop
    assert   msgs_ok (m, at_bom, fm, st)
           & correct_filter_index (fc, filter)
           & partly_filtered (m, at_bom, filter,
                              pass_it, outto(log, myid),
                              m[st..size(m)], at_bom & st=1);
    if st > size(fm) then leave
    else
      match_all_patterns (fm, st, filter, fc, pass_it,
                          m, at_bom, log);
      next_char (fm, st, m, at_bom);
    end;
  end;
end;
```

```
procedure match_all_patterns (          fm : a_msg;
                                        st : an_int;
                                    filter : a_filter;
                                        fc : a_filter_index;
                              var pass_it : boolean;
                                         m : a_msg;
                                    at_bom : boolean;
                              var     log : a_log_buffer<output>) =
begin
  entry   msgs_ok (m, at_bom, fm, st) & st le size(fm)
        & correct_filter_index (fc, filter);
  block found_match (m[st..size(m)], at_bom & st=1, filter,
                     pass_it', outto(log,myid));
  exit did_filter_match (m[st..size(m)], at_bom & st=1, filter,
                         pass_it', pass_it, outto(log,myid));
  var i, b: an_int;
  var star: boolean;
  var c: a_pattern_char_index;
  i := 1;  b := fc['*'].bot;  star := true;
  loop
    assert   doing_filter_match (m[st..size(m)], at_bom & st=1,
                                 filter, pass_it', pass_it,
                                 outto(log,myId), i, b)
           & b = end_filter_section (fc, star, fm[st])
           & msgs_ok (m, at_bom, fm, st) & st le size(fm)
           & correct_filter_index (fc, filter);
    if i > b then
      if star then
        star := false;
        c := fc[fm[st]];
        i := c.top;
        b := c.bot
      else leave
      end
    else match_pattern (fm, st, filter[i], pass_it, m, at_bom, log);
         i := i + 1;
    end;
  end;
end;
```

D.2 Supporting Specification Functions for the Pattern Matcher

```
function end_filter_section (  fc : a_filter_index;
                             star : boolean;
                               ch : a_pattern_char) : integer =
begin
  exit (assume result = if star then fc['*'].bot
                        else fc[ch].bot
                        fi);
end;

function correct_filter_index (    fc : a_filter_index;
                               filter : a_filter)        : boolean =
begin
  exit (assume result = all ch: character,
                            ch in ['*..'Z]
                        -> [  section_range_ok (fc, ch, filter)
                            & missed_no_patterns (fc, ch, filter)]);
end;

function doing_filter_match (       m : a_msg;
                               at_bom : boolean;
                               filter : a_filter;
                              ipass_it : boolean;
                               pass_it : boolean;
                                   log : a_char_seq;
                                 i,bot : integer)   :  boolean =
begin
  exit (assume
    result =
      [  filter_match (pass_it, ipass_it, m, at_bom, filter[1..i-1])
       & filter_match_log (m, at_bom, filter, ipass_it)
         = log  filter_match_log (m,at_bom,
                                  filter[i..size(filter)],pass_it)
       & (pass_it -> log = null(a_char_seq))
       & i in [1..bot+1] & bot le size(filter)]);
end;

function msgs_ok (     m : a_msg;
                  at_bom : boolean;
                      fm : a_msg;
```

```
                        st : integer)  :  boolean =
begin
  exit (assume
    result =
      (  st in [1..size(fm) + 1] & size(fm) = size(m)
       & [at_bom -> has_bom (m)]
       & fm[st..size(fm)] = normal_form (m[st..size(m)],
                                        at_bom & st=1)));
end;

function normal_form (      m : a_msg;
                       at_bom : boolean) : a_char_seq =
begin
  exit (assume
    result =
      if m = null(a_msg) then null(a_char_seq)
      else if at_bom & has_bom(m) then
             bom_delimiter  normal_form (non_first(m,at_bom), false)
           else if m = eom then eom_delimiter
                else [seq: normalize(first(m))]
                       normal_form (nonfirst(m), false)
      fi   fi   fi);
end;

function normalize (ch : character) : character =
begin
  exit (assume result = if nondelimiter(ch) then upper_case(ch)
                        else dot
                        fi);
end;
```

D.3 Supporting Functions for Normalization

```
function nondelimiter (ch : character) : boolean =
begin
  exit result = (ch in ['0..'9] or
                 ch in ['A..'Z] or
                 ch in ['a..'z]);
  result := ['0 le ch and ch le '9] or
            ['A le ch and ch le 'Z] or
            ['a le ch and ch le 'z];
end;

function upper_case (c : character) : character =
begin
  exit result = if c in ['a..'z] then
                  scale (ord(c) - 32, character)
                else c
                fi;
  if 'a le c & c le 'z then
    result := scale (ord(c) - 32, character)
  else result := c
  end;
end;
```

Bibliography

[Ada 79] *Preliminary Ada Reference Manual*
Department of Defense, 1979.
Also published as Sigplan Notices, Volume 14, Number 6, Part A, June, 1979.

[Aho 77] Aho, A.V. and Ullman, J.D.
Principles of Compiler Design.
Addison Wesley, 1977.

[Akers 83] Akers, R. L.
A Gypsy-to-Ada Program Compiler.
Master's thesis, University of Texas at Austin, May, 1983.

[Anna 80] Krieg-Bruckner, B. and Luckham, D. C.
ANNA: Towards a Language for Annotating Ada Programs.
SIGPLAN Notices 15(11):128-138, November , 1980.

[Backus 67] Backus, J.W., *et al.*
The FORTRAN Automatic Coding System.
McGraw-Hill, New York, 1967, pages 29-47chapter 2A.

[Baker 72] Baker, F. T.
Chief Programmer Team Management of Production Programming.
IBM Systems Journal 11(1), 1972.

[Ball 79] Ball, J.E., Low, J. R., and Wiliams, G. J.
Preliminary ZENO Language Description.
SIGPLAN Notices 14(9):17-34, September , 1979.

[Bell 73] Bell, J.R.
Threaded Code.
CACM 16(6):370-372, June , 1973.

[Best 81] Best,E. and Cristian,F.
Systematic Detection of Exception Occurrences.
Technical Report 165, Computing Laboratory, University of Newcastle upon Tyne, April, 1981.

[Bledsoe 75] Bledsoe, W.W. and Tyson, M.
The UT Interactive Prover.
Technical Report ATP-17, University of Texas Mathematics Dept., May, 1975.

[Boyer 75] Boyer, R. S. and Moore, J. S.
Proving Theorems About Lisp Functions.
JACM 22(1), 1975.

[Boyer 80] Boyer, R. S. and Moore, J S.
A Verification Condition Generator for Fortran.
Technical Report CSL-103, SRI International, June , 1980.

[Boyer 81] Boyer, R. S. and Moore, J S.
MJRTY - A Fast Majority Vote Algorithm.
Technical Report ICSCA-CMP-32, Institute for Computing Science, University of Texas at Austin, February, 1981.

[Boyer 82] Boyer, R. S., Green, M. W. and Moore, J S.
The Use of a Formal Simulator to Verify a Simple Real Time Control Program.
Technical Report ICSCA-CMP-29, Institute for Computing Science, The University of Texas at Austin, July, 1982.

[Boyer 83] Boyer, R.S. and Moore, JS.
On Why it is Impossible to Prove that the BDX930 Dispatcher Implements a Time-Sharing System.
In *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers*, . SRI International, 1983.

[Burger 74] Burger, W.F.
BOBSW - A Parser Generator.
Technical Report SESLTR-7, University of Texas at Austin, December, 1974.

[Carter 74] Carter, H.B.
Code Optimization of Arithmetic Expressions in Stack Environments.
Technical Report NRL Report 7787, Naval Research Laboratory, Washington, D.C., September , 1974.

[Carter 77] Carter, J.L.
A Case Study of a New Code Generation Technique for Compilers.
CACM 20(12):914-920, December , 1977.

[Cheheyl 81] Cheheyl, M.H., Gasser, M., Huff, G.A. and Millen, J.K.
Verifying Security.
ACM Computing Surveys 13(3):279-340, September , 1981.

[Cocke 70] Cocke, J. and Schwartz, J.T.
Programming Languages and Their Compilers.
Courant Institute of Mathematical Sciences, New York University, 1970.

[Cohen 83] Cohen, R.C.
Proving Properties of Gypsy Programs.
PhD thesis, University of Texas at Austin, December, 1983.
Work in progress - Title and Date are tentative.

[Cristian 82] Cristian,F.
Exception Handling and Software Fault Tolerance.
IEEE Transactions on Computers C-31(6):531-540, June , 1982.

[De Millo 79] De Millo, R.A., Lipton, R.J., and Perlis, A.J.
Social Processes and Proofs of Theorems and Programs.
CACM 22(5):271-280, May, 1979.

[Denning 77] Denning, D.E. and Denning, P.J.
Certification of Programs For Secure Information Flow.
CACM 20(7):504-513, July , 1977.

[Dijkstra 68] Dijkstra, E.W.
GOTO Statement Considered Harmful.
CACM 11(3):147-148, March , 1968.

[Eggert 81] Eggert, P. R.
Detecting Software Errors before Execution.
Technical Report CSD-810402, UCLA Computer Science Department, April , 1981.

[Elliot 82] Elliot, W. D.
On Proving the Absence of Execution Errors.
Technical Report CSRG-141, Computer Systems Research Group, University of Toronto, March , 1982.

[Floyd 67] Floyd, R. W.
Assigning Meanings to Programs.
In Schwartz, J. T. (editor), *Proceedings of a Symposium in Applied Mathematics, Vol. 19*, pages 19-32. American Mathematical Society, 1967.

[German 78] German, S. M.
Automating Proofs of the Absence of Common Runtime Errors.
In *Conference Record of the Fifth Annual ACM Symposium on the Principles of Programming Languages*, pages 105-118. ACM, January , 1978.

[Good 77] Good, D.I.
Constructing Verifiably Reliable and Secure Communications Processing Systems.
Technical Report ICSCA-6, University of Texas at Austin, Institute for Computing Science and Computer Applications, January, 1977.

[Good 78] Good, D.I., Cohen, R.M., Hoch, C.G., Hunter, L.W., Hare, D.F.
Report on the Language Gypsy: Version 2.0.
Technical Report ICSCA-10, University of Texas at Austin, Institute for Computing Science and Computer Applications, September , 1978.

[Good 82] Good, D.I., Siebert, A.E., and Smith, L.M.
Message Flow Modulator Final Report.
Technical Report ICSCA-CMP-34, Institute for Computing Science, University of Texas at Austin, December, 1982.

[Goodenough 75] Goodenough, J. B.
Exception Handling: Issues and a Proposed Notation.
CACM 18(12):683-696, December, 1975.

[Graham 76] Graham, S.L. and Wegman, M.
A Fast and Usually Linear Algorithm for Global Flow Analysis.
JACM 23(1):172-202, January , 1976.

[Hare 79] Hare, D.F.
A Structure Program Editor for the Gypsy Verification Environment.
Master's thesis, University of Texas at Austin, 1979.

[Hoare 71] Hoare, C.A.R.
Program Development by Stepwise Refinement.
CACM :14, April, 1971.

[Hoare 76] Hoare, C.A.R. and Wirth, N.
An Axiomatic Definition of the Programming Language PASCAL.
Acta Informatica :2, 1976.

[INTEL 81] *Introduction to the iAPX 432 Architecture*
Intel Corporation, Santa Clara, California, 1981.
Order Number 171821-001.

[Jensen 74] Jensen, K. and Wirth, N.
Pascal: user manual and report.
Springer-Verlag, New York, 1974.

[Lampson 77] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L.
Report on the Programming Language Euclid.
SIGPLAN Notices 12(2):1-79, February , 1977.

[Landwehr 81] Landwehr, C.E.
Formal Models for Computer Security.
ACM Computing Surveys 13(3):247-278, September, 1981.

[London 70] London, R. L.
Certification of Algorithm 245.
CACM 13, June, 1970.

[Loveman 77] Loveman, D.B.
Program Improvement by Source-to-Source Transformation.
JACM 24(1):121-145, January, 1977.

[Low 83] Low, J.R. .
Private Communication.
August , 1983.

[Luckham 77] Luckham, D.C.
Program Verification and Verification-Oriented Programming.
American Elsevier, New York, 1977, pages 783-793.

[Luckham 80a] Luckham, D.C. and Polak, W.
A Practical Method of Documenting and Verifying Ada Programs with Packages.
SIGPLAN Notices 15(11):113-122, November, 1980.

[Luckham 80b] Luckham, D.C and Polak, W.
Ada Exception Handling: An Axiomatic Approach.
TOPLAS 2(2):225-233, April, 1980.

[McCarthy 63] McCarthy, J.
A Basis For a Mathematical Theory of Computation.
North-Holland, Amsterdam, 1963, pages 33-70.

[McCarthy 67] McCarthy, J. and Painter, J.
Correctness of a Compiler for Arithmetic Expressions.
In Schwartz, J. T. (editor), *Proceedings of a Symposium in Applied Mathematics, Vol. 19*, pages 33-41. American Mathematical Society, 1967.

[McHugh 75] McHugh, J.
Structured Development and Constructive Proof of a Real Time Data Acquisition System.
1975.
M. S. Scholarly Paper, University of Maryland.

[Melliar-Smith 82]
Melliar-Smith, P. M. and Schwartz, R. L.
Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System.
IEEE Transactions on Computers C-31(7):616-630, July, 1982.

[Mills 72] Mills, H. D.
Mathematical Foundations for Structured Programming.
Technical Report FSC 72-6012, IBM, 1972.

[Moriconi 77] Moriconi, M.S.
A System for Incrementally Designing and Verifying Programs.
Technical Report ICSCA-CMP-9, Institute for Computing Science, University of Texas at Austin, December, 1977.

[Moriconi 80] Moriconi, M. and Schwartz, R.L.
Automatic Construction of Verification Condition Generators from Hoare Logics.
Technical Report CSL-125, SRI International, Menlo Park, Ca, November , 1980.

[Patkau 79] Patkau, B.H.
An Analysis of the Legality Assertion Mechanism in Euclid.
1979.
Computer Systems Research Group, University of Toronto.

[PL/I 72] *PL/I (F) Reference Manual*
5th edition, International Business Machines Corporation, 1972.

[Reitman 79] Reitman, R.P. and Andrews, G.R.
Certifying Information Flow Properties of Programs: an Axiomatic Approach.
In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, pages 283-290. ACM-SIGPLAN, Jan, 1979.

[Scheifler 77] Scheifler, R.W.
An Analysis of Inline Substitution for a Structured Programming Language.
CACM 20(9):647-654, September , 1977.

[Smith 80] Smith, L.
Compiling from the Gypsy Verification Environment.
Master's thesis, University of Texas at Austin, 1980.

[Stanford 79] Stanford Verification Group.
Stanford Pascal Verifier User Manual.
Technical Report STAN-CS-79-731, Stanford University Computer Science Department, March , 1979.

[von Neumann 61]
von Neumann, J.
Planning and Coding Problems for an Electronic Computing Instrument.
In Taub, A.H. (editor), *John von Neumann, Collected Works, Volume V*, pages 80-235. Pergamon, 1961.

[Wortman 79] Wortman, D.B.
On Legality Assertions in Euclid.
IEEE Transactions on Software Engineering SE-5(4):359-367, July , 1979.

[Wortman 81] Wortman, D. B., Holt, R. C., Cordy, J. R., Crowe, D. R., and Griggs, I. H.
Euclid - A Language for Compiling Quality Software.
In *1981 National Computer Conference*, pages 257-264. AFIPS Press, 1981.

[Wulf 71] Wulf, W.A., Russel, D.B., and Habermann, A.M.
BLISS: a language for systems programming.
CACM 14(12):780-790, December , 1971.

[Wulf 78] Wulf, W. A. *et al.*.
(Preliminary) An Informal Definition of Alphard.
Technical Report CMU-CS-78-105, Carnegie-Mellon University Department of Computer Science, February , 1978.

[Wulf 80] Wulf, W.A.
PQCC: A Machine-Relative Compiler Technology.
Technical Report CMU-CS-80-144, Carnegie-Mellon University Computer Science Department, September, 1980.

[Young 80] Young, W. D. and Good, D. I.
Generics and Verification in Ada.
SIGPLAN Notices 15(11):123-127, November , 1980.

[Zelkowitz 74] Zelkowitz, M.V. and Bail, W.G.
Optimization of Structured Programs.
Software - Parctice and Experience 4(1):51-57, January - March , 1974.