THE PROOF OF A DISTRIBUTED SYSTEM IN GYPSY

Donald I. Good

Technical Report 30   September 1982

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Institute for Computing Science
The University of Texas at Austin
Austin, Texas 78712

Abstract

The Gypsy methods for specifying, implementing and mechanically proving distributed systems have been used very successfully in their first major trial application. The application was a special interface between a host and an IMP on the ARPANET. The interface contained 4211 lines of executable high level language program. The basic concepts of Gypsy and its methods for distributed systems are described and illustrated on a small example.

Acknowledgements

During its eight year existence, well over 50 people have contributed to the Gypsy project, and I will not attempt to name them all. Gypsy, however, has been a true team effort, and the current team consists of Mike Smith, Ann Siebert, Bill Young, Rich Cohen, John McHugh, Larry Akers, Larry Smith, Bill Bevier, Judy Merriam, Ben DiVito, Sandy Olmstead, and Ayshegul Musallam.

The Gypsy project has been sponsored in part by the U. S. Naval Electronic Systems Command through contract N00039-81-C-0074, by Digicomp Research Corporation, and by Digital Equipment Corporation.

1. Introduction

Since the mid 1960's formal program specification and deductive proof methods have offered the potential for building programs that are very much more reliable than those produced by conventional methods. The Gypsy project was begun in 1974 with the goal of developing that potential. Gypsy is a set of methods for building real computer programs that are deductively proved to meet formal specifications. We do not intend to use Gypsy to prove all real programs, but we certainly do intend to use Gypsy to prove complex enough systems so that we can realistically evaluate the technical feasibility of applying these methods in actual practice.

The design of the Gypsy language is based on a carefully integrated set of methods for specifying, implementing and proving programs. The language has facilities for expressing both specifications and implementations of programs; and it is designed so that it is always possible to construct automatically theorems, called verification conditions, which are sufficient to show that an implementation satisfies its specification. The Gypsy Verification Environment (GVE) implements the tools that are needed to apply these methods in actual practice. The GVE includes tools for constructing specifications, implementations, verification conditions, and proofs. It also contains various tools for running the resulting programs.

Gypsy specifications, implementations and proofs can be formulated at arbitrarily high levels of abstraction. Very abstract specifications and implementations can be used to express a design, and proofs can be constructed at this design level. Usually, these higher levels of abstraction are where formal specification and proof are most effective. Gypsy also provides mechanisms for defining precise relations among different levels of abstraction. With these mechanisms, the highest levels of design can be related precisely, possibly through many intermediate levels, to the lowest levels of coding. Although a program may contain many levels of abstraction, the same methods of specification, implementation and proof are used for each level. This reflects our view that the main difference between "design" and "coding" is the level of abstraction.

One of the main strengths of Gypsy is its proof methods for concurrent programs. These methods are based on simple message passing, and they are designed so that all of the preceding abstraction mechanisms apply. Therefore, concurrent systems can be factored, in a manner fully analogous with sequential programs, into components that can be proved independently. The message passing mechanism also allows the specification and proof methods for concurrent programs to be applied unaltered to many kinds of distributed systems.

Gypsy has been used very successfully in its first major trial application. The application is an special interface between a host and an IMP on the ARPANET. The host is assumed to have sensitive message traffic, and the ARPANET is assumed to have no mechanism for protecting these messages. The interface consists of 4211 lines of executable Gypsy which is distributed over two PDP 11/03 computers. The specification for the interface was that all packets moving from the host to the IMP are properly protected and that all packets moving from the IMP to the host are properly unprotected. This specification does not, in any way, describe everything that the interface does, but it does describe completely its desired message protection property. Even if the interface fails in some other way, its protection property is preserved.

This specification and the full interface implementation were expressed in Gypsy, through all layers of abstraction, and the interface was proved mechanically with the GVE. All of the 2600 verification conditions were constructed automatically, and 88% of these were proved fully automatically. The remaining 12% were proved with the interactive theorem prover. Only two unproved lemmas were assumed in these proofs. The verified interface has been successfully tested and demonstrated on the ARPANET. The complete interface project, including specification, implementation, proof, testing, demonstration and documentation was performed over a very discontinuous 60 month period from January 1977 through December 1981 with a total effort of 1144 person-days and 444 CPU hours of DEC 2060 computer time. A total of 7305 lines of Gypsy text (specification and implementation for all levels) was produced for an overall productivity rate of 6 lines/person-day and 16 lines/CPU-hour. It is important to note that if the specification were stronger, more effort would have been required.

In the following discussion, we will first describe the basic elements of Gypsy and its verification environment, and then describe the special mechanisms for concurrent and distributed systems. Finally, we will illustrate the use of Gypsy to specify, implement and prove a distributed two channel message mover that multiplexes messages across a single connection between two separate machines.

2. Gypsy

The following describes the basic concepts of specification, implementation and proof in the Gypsy language, [Good 78a], and the major tools of the verification environment.

2.1 Basic Concepts

The main objective of Gypsy is to prove that a program is implemented so that it always runs as specified. The specification states desired properties of its run time behavior, and we must be able to construct a deductive proof that the program always has these properties whenever it is runs.

The fundamental building blocks of all Gypsy programs are two kinds of routines, PROCEDURES and FUNCTIONS. These are the only components of a Gypsy program that actually run. Gypsy also has CONSTANTS, LEMMAS, and TYPES. Constants and lemmas are just special kinds of functions, and types are specifications for data parameters to routines. Procedures, functions, constants, lemmas and types are the five kinds of Gypsy units, and all Gypsy program specifications and implementations are composed of these. In general, implementations are defined by procedures, functions, and constants; and specifications are defined by types, functions, constants and lemmas. Example of all of these units appear in Appendix A.

The foundation of Gypsy is a set of predefined procedures and functions; and every Gypsy program, no matter how complex, is some composition of these. The predefined routines differ from defined ones in that the predefined ones normally require a special syntax to call them. For example, the composition

```
        #ASSIGN(X,#ADD(#SELECT(A,I),Y))
```

```
must be written as
```

```
        X := A[I] + Y.
```

#ASSIGN is the primitive assignment procedure, #SELECT is a function that takes an array A and an index I and returns the value of element I of the array, and #ADD is the addition function. Defined routines can be composed from the predefined and other defined routines.

The only effects that any Gypsy routine can produce are to transform data objects and to SIGNAL exception CONDITIONs. The general form of all Gypsy routines is

```
  PROCEDURE P(VAR X:X_TYPE; Y:Y_TYPE) UNLESS (COND C) =
  BEGIN
    ...
  END;

  FUNCTION F(Y:Y_TYPE):F_TYPE UNLESS (COND C) =
  BEGIN
    ...
  END;
```

The parts of these forms that are displayed above define the formal data and condition parameters. The parts that are suppressed define the composition of the routine in terms of other routines. Operationally, there are only two differences between procedures and functions. First, functions return a value as a result, whereas procedures produce a result by transforming the values of their var parameters. Second, functions are expected to be deterministic. Given the same input values, a function should always produce the same output value. To accommodate concurrency, procedures may or may not produce deterministic results.

Gypsy constants and lemmas are functions which have a special syntactic form. A constant definition has the general form

```
        CONST N:N_TYPE = ...
```

A constant is a function of zero arguments, and therefore, it always returns the same value. A lemma definition has the form

```
        LEMMA L(X:X_TYPE) = ...
```

Lemmas are boolean valued functions which are used only in specification and proof.

Gypsy routines may have only three kinds of parameters, VAR, CONST, and COND, and all three are passed by reference. Var and const parameters are data parameters. The difference is that a routine may change the value of a var parameter, but it may not change the value of a const parameter. A cond parameter is a label parameter.

In general, routines may have any number of data parameters, but there are some restrictions. First, only procedures may have explicitly declared var parameters. Second, functions always have exactly one var parameter, named RESULT. It is the value returned by the function, and it is declared implicitly for every function. Further, routines may refer to external data and condition objects only through their parameter lists. Nonlocal referencing to data and condition objects is not allowed.

These restrictions have several important consequences. First, the parameter list is the sole means by which a routine obtains access to external data and condition objects. Second, the value of an external data object can be changed only if it appears as a var parameter to a procedure. Third, functions have absolutely no side-effects on data objects. These consequences greatly enhance the ability to determine the run time effects of a routine independently of its external environment.

All routines have one implicitly declared cond parameter called ROUTINEERROR. The use of ROUTINEERROR is described a bit later. Routines may have any number of other cond parameters.

The sole purpose of Gypsy types is to define specifications for data parameters to routines. A Gypsy type defines a set of values, and a type must be specified for every formal data parameter of every routine. The run time value of a parameter always must be in the value set of its type. Gypsy contains a number of predefined types, and additional types can be defined by composing these. Type definitions have the form

```
        TYPE T = ...
```

Type T defines a new set of values from the sets mentioned in the ... part of the definition. Gypsy also contains a data abstraction mechanism for limiting the visibility of the composition of a type.

Gypsy routines are analyzed and proved under two basic assumptions. First, data parameters satisfy their type specifications. Second, there is no potentially harmful aliasing among data parameters. Aliasing is potentially harmful if there are dependencies among the data parameters such that a change in one of them may cause changes in the values of some of the others. This non-aliasing assumption greatly simplifies the amount of analysis needed to prove a routine. When actuals are bound to formals, these assumptions must be verified. They usually can be verified statically, but if not, they must be checked dynamically at run time.

The core of Gypsy is its parameter mechanism. The general requirements for all parameters are the following:

1. All actual parameters are bound to formal parameters by position.
2. All parameters are passed by reference.
3. The number of actuals must equal the number of formals.

Data parameters also must meet the following requirements:

1. When an actual is bound to its formal parameter, the actual must refer to a data object whose value is in the value set specified by the type of the formal. This restriction is enforced for both var and const parameters. If this restriction must be checked at run time, a VALUEERROR is signalled if it is violated.

2. When an actual is bound to a var formal parameter, the value set of the type of the formal must be wholly contained in the value set of the type of the actual (otherwise, at some point while the routine runs, the actual might have to take on values outside the value set of its type). If this restriction must be checked at run time, a VARERROR is signalled if it is violated.
3. An actual var parameter must not be aliased by any other actual parameter (var or const). If this restriction must be checked at run time, an ALIASERROR is signalled if it is violated. (This restriction is not enforced for predefined routines on the assumption that predefined routines have been rigorously analyzed to determine that any aliasing that might occur is not harmful. The #ASSIGN procedure is an example of aliasing that is not harmful.)

Actual condition parameters can be supplied according to the general parameter requirements. Also, there is a default convention. It is possible to state no actual parameters for a routine that has formal cond parameters. In this case, actual cond parameters with names identical to the corresponding formals are created by default. In designing Gypsy, we have tried very hard to avoid any kind of default conventions so that all things that cause effects in a program are textually visible. The reason for allowing this one is that most of the predefined routines have condition parameters. For example, the #SELECT function mentioned above for referring to elements of an array has a formal INDEXERROR condition. Supplying actual parameters for all of these predefined conditions would largely destroy the readability of the text. For consistency, the default convention also is allowed for defined routines.

The Gypsy predefined types include INTEGER, RATIONAL, BOOLEAN, CHARACTER, ARRAY, RECORD, SET, SEQUENCE, and MAPPING, and predefined functions are associated with each type. Types integer, rational, boolean, set, sequence, and mapping are the well known mathematical structures, and arrays and records can be defined in terms of these. For example, type integer is the complete unbounded set of positive and non-positive integers. The fact that these types correspond to the standard mathematical structures allows us to draw from a highly developed body of knowledge about how to prove things about these objects.

These potentially unbounded structures clearly present some serious difficulties for running Gypsy programs on real machines. Each Gypsy implementation, however, may define an implementation prelude for a specific target machine. This prelude restricts Gypsy to what is efficiently implementable on that machine. These restrictions, however, must conform to all aspects of Gypsy semantics. For example, the prelude might include

```
TYPE MACHINE_INT = INTEGER[MIN_INT..MAX_INT];
CONST MIN_INT = ...
CONST MAX_INT = ...
```

This defines type MACHINE_INT as the set of integers in the range from MIN_INT to MAX_INT. The implementation, then, need not implement integer arithmetic outside of this set, the conditions on the predefined routines may be signalled instead; but, within the set, it must produce a running program that satisfies all the rules of normal integer arithmetic -- e.g. addition must commute, associate, etc.

The presence of these potentially unbounded mathematical structures means that we can write Gypsy programs that can not run on any real machine. The specification and proof methods, however, can be applied to all Gypsy programs regardless of whether they can be run or not.

Gypsy text is composed of definitions of the five kinds of Gypsy units. These definitions may not be nested; and thus, Gypsy has a flat, instead of a hierarchical, name space. This name space may be partitioned with SCOPEs. A scope is simply a textual grouping of units into one partition of the name space. The name of each unit within a scope must be unique, and all scope names must be unique. Therefore, the scope name and the local unit name, together, define a globally unique name for the unit. A NAME declaration enables inter-scope referencing. Lexically, Gypsy text is wholly composed of a sequence of scopes. All units must be defined in some scope, and each scope contains only definitions of units, name declarations and comments.

2.2 External Specification

Every Gypsy routine has an external specification that describes what effect that the routine has on its external environment. These effects are described strictly in terms of relations among the initial and final values of formal parameters. The external specification, therefore, plays a very important abstraction role. It describes what effects are caused by the routine, but it hides the details of how they are caused.

The external specification of a routine consists of two parts, a mandatory interface specification and an optional functional specification. The interface specification defines the name of the routine, its formal parameters and their types. The functional specification describes the effects of the routine on its parameters. The general forms of these specifications are

```
PROCEDURE P(VAR X:X_TYPE; Y:Y_TYPE) UNLESS (COND C) =
BEGIN
ENTRY PRE_P(X,Y);
EXIT (CASE
      IS NORMAL: NORMAL_POST_P(X,X’,Y);
      IS      C: C_POST_P(X,X’,Y));
END;

FUNCTION F(Y:Y_TYPE):F_TYPE UNLESS (COND C) =
BEGIN
ENTRY PRE_F(Y);
EXIT (CASE
      IS NORMAL: NORMAL_POST_F(RESULT,Y);
      IS      C: C_POST_F(RESULT,Y));
END;
```

The interface specification isolates the effects that a routine can have on its external environment. It defines all possible conditions that can be signalled back to the external environment, and it completely identifies all external data objects that are accessible to the routine. It also completely specifies which data parameters can and can not be modified by the routine. Therefore, the most that a routine can do to its external data environment, even if it signals one of its conditions, is to take the initial values of its data parameters and produce new values for its var parameters. Because of the type consistency rules of the parameter passage mechanism, we know that whatever new value is produced for a var parameter, it will be within the type of the formal parameter. Thus, the interface specification both isolates the effects of the routine, and it defines a very weak functional specification for the routine which says that the routine produces some new value of the appropriate type for each of its var parameters.

The functional part of the external specification of a routine is used to state a stronger functional specification than the weak one implied by the interface specification. The functional specification states a relation between the initial and final values of the data parameters of the routine. Initial values may be different from final values only for var parameters, and the notation x’ refers to the initial value of parameter x.

The functional specifications are stated by the ENTRY and EXIT parts of the specification. The relation that is stated is that if the entry specifications is true when the routine is entered, then the exit specification is true if the routine terminates. (Methods for stating and proving termination have not yet been installed in Gypsy.) The exit spec may state a different relation for each cond parameter, and the commonly used form

```
EXIT (CASE
      IS NORMAL: NORMAL_POST(X,X’,Y);
      IS      C: TRUE);
```

`can be abbreviated as just`

```
EXIT NORMAL_POST(X,X’,Y);
```

One of the common uses of external specifications is to define "specification" functions. This is an informal term that refers to functions that are intended to be used only in writing specifications. For example, we might define NORMAL_POST as follows:

```
FUNCTION NORMAL_POST(U,V,W:INTEGER):BOOLEAN =
BEGIN
EXIT (ASSUME RESULT IFF
        W = IF V LE 0 THEN V ELSE FACTORIAL(U) FI);
END;

FUNCTION FACTORIAL(N:INTEGER):INTEGER =
BEGIN
ENTRY N GE 0;
EXIT (ASSUME RESULT =
       IF N = 0 THEN 1 ELSE FACTORIAL(N-1)*N FI);
END;
```

This defines NORMAL_POST in terms of another function, FACTORIAL which is defined (recursively) in terms of Gypsy primitives. ASSUME means that these exit specifications are to be assumed without proof. Note that these functions have only external specifications. No implementations are given, and therefore, it would not be possible to run these functions. They are stated just for the purpose of defining precisely the functions that are used to write specifications for other routines.

2.3 Implementation and Internal Specification

The effect of a routine is caused by its implementation. Syntactically, the implementation follows the external spec of a routine, and it also may contain certain internal specifications. Semantically, the implementation of a routine defines how it is composed from other routines, and the internal specs state properties of the internal state of the routine. Internal specs differ from the external specs in that they are not visible to the external environment.

The following is an example of a fully specified and implemented routine:

```
PROCEDURE FIND_IT (X : AN_OBJECT; A : AN_OBJECT_ARRAY;
               VAR K : AN_INDEX) UNLESS (NO_SUCH_OBJECT) =
BEGIN
   ENTRY LO LE HI;
   EXIT CASE (IS NORMAL : ISIN (X, A, LO, HI) & A[K] = X;
      IS NO_SUCH_OBJECT : NOTIN (X, A, LO, HI) & K = K’);
   VAR I:AN_INDEX;
   I := LO;
   LOOP ASSERT NOTIN (X, A, LO, I - 1) & I IN [LO - 1..HI];
      IF A[I] = X THEN K := I; LEAVE
      ELIF I = HI THEN SIGNAL NO_SUCH_OBJECT
      ELSE I := I + 1
      END;
   END
END;
```

The ASSERT statement is an internal spec. It is a relation that is to be satisfied every time control reaches that point in the implementation. An assert relation may refer to formal parameters and to local objects, and thus, it may describe how the internal state relates to the parameters. Internal specifications may be used throughout the implementation, but they are not required by the Gypsy syntax. They are required, however, to prove routines with loops. To prove a routine, an assert spec must occur on every iteration of every loop. In the proof, the assert relation becomes the induction step of an induction proof.

The implementation of a routine consists of the declaration of local objects (var, const, cond) followed by statements that perform transformations on these objects and the parameters of the routine. The local objects and the parameters of a routine, are the entire set of data and condition objects that are accessible to the routine. The objects of a routine are transformed by a sequence of procedure calls.

The basic control statements in Gypsy are IF, CASE, AND LOOP. They are illustrated in Appendix A. The IF statement has an ELIF part which is an abbreviation for ELSE IF. The CASE statement has an optional ELSE part that covers all cases not specifically enumerated in the IS part. LOOP statements are terminated by performing a LEAVE statement. A loop may contain any number of leave statements, and executing a leave terminates the most tightly enclosing loop.

Exception handling in Gypsy is done with conditions and signal statements. A SIGNAL statement causes a forward jump to a COND label. All of the Gypsy statements that can contain other statements end with END. Any statement that has an END also may have condition handlers which are signified by WHEN. The general form is

```
{OPENING} KEYWORD
          .....
          WHEN
          IS C: P1(X);
          ....
          ELSE: PN(X);
{CLOSING} END
```

When a condition is signalled within the KEYWORD...WHEN part of the statement, control is immediately transferred to the WHEN part to the procedure labelled by that condition name, and then to the END of the statement. The semantics of the WHEN part is like a case statement except that if the condition does not appear in the WHEN part, control is transferred to the next most tightly enclosing WHEN part. The search of WHEN parts for the condition name continues until the boundary of a user defined routine is reached. At that point, the condition is automatically converted into the predefined condition ROUTINEERROR and signalled to the external environment. This conversion prevents inadvertent revelation of implementation detail to the calling environment.

2.4 Verification

The full text of a Gypsy routine, including its external and internal specifications and its implementation, can be viewed as a formula in a program logic. The formula expresses a specific set of properties about a specific routine. The main purpose of Gypsy is to state and verify that these formulas always are true. Gypsy supports three strategies of verification: conventional testing, validation at run time and deductive proof.

We can, of course, do conventional program testing on any Gypsy program. The presence of formal specifications in Gypsy makes it possible to go further and turn a single test into a proof about that one case. In general, however, proofs about specific test cases are not sufficient to show that a program formula is valid.

Actually, it is not necessary that a program formula be completely valid; it is only necessary that it be true for those cases that arise in actual practice. Therefore, one strategy for obtaining "adequate validity" is to test the program formula as the program actually runs. To do this, Gypsy specifications in program formulas may have an OTHERWISE C suffix where C is a condition. This is the designation for run time validation. With an OTHERWISE suffix, the specification is actually evaluated at run time and tested for TRUE or FALSE. If it is TRUE, control continues in the normal manner; otherwise, condition C is signalled. Whether or not a specification can be validated at run time depends on what parts of Gypsy are supported by a particular implementation, and the usefulness of run time validation depends on whether there is a reasonable recovery from the specification not being satisfied.

The most powerful verification method is deductive proof, and this is assumed to be the desired approach. All specifications in program formulas are marked either PROVE or ASSUME. PROVE is the designation for deductive proof. ASSUME means that the specification is to be assumed without verification. It typically is used to define spec functions, but it also can be used on any kind of specification in a program formula. The textual absence of both of these markings on a spec is interpreted as PROVE.

Figure 1: Proof Structure

```
                 Gypsy         External Specs
Type             Program          of Called
Info             Formula          Procedures
 |                  |                   |
 |                  |                   |
 +--------------------------------------+
 |                  |
 |                  v
 |        +------------------------+
 |        | Verification Condition |
 |        |       Generator        |
 |        +------------------------+
 |                  |
 |                  |
 |              Predicate           Properties of
 |              Calculus            Spec Functions
 |              Formulas         (Definitions, Lemmas)
 |                  |                      |
 |------------------+----------------------+
                    |
                    |
                    v
             +-------------+
             |  Theorem    |
             |   Prover    |
             +-------------+
                    |
                    |
                    v
                  Proof
```

The program proof strategy is to transform program formulas into predicate calculus formulas. The entire Gypsy language is designed so that program formulas can be proved as illustrated in Figure 1. The program formula, the external specs of the procedures it calls, and certain type information are transformed into predicate calculus formulas called "verification conditions" (vcs). These are formulas which are sufficient, but not always necessary, to show that the program formula is valid. Conventional predicate calculus proof methods then are used to prove the verification conditions. The vcs are composed of Gypsy functions, and the proofs are based on properties of these functions. These properties are stated as ordinary Gypsy external specifications about the functions. These properties usually take the form of function definitions and lemmas.

A very important characteristic of this proof structure is that the proof of a routine is totally independent of the implementation of the routines it calls. The only information that is assumed about a called routine is that it satisfies its external spec. Thus, the proof of calling routine is valid for any implementation of the called routines so long as they satisfy their external specs. This is the characteristic that allows higher level routines to be proved before lower level routines are implemented; and this, therefore, is what allows us to do very high design level proofs in Gypsy in advance of doing low level coding. This characteristic of Gypsy is called the "independence principle," and it has been one of the foundations of the language design. This principle is the basis for decomposing proofs of large systems into small pieces, and the maintenance of this principle in concurrent programs is one of the major achievements of Gypsy.

2.5 Abstraction

The key to specifying and proving sizeable Gypsy programs is abstraction. The Gypsy abstraction mechanisms are how a large proof is broken into manageable pieces.

There are three items that are required in the proof of every routine, its specifications, its implementation, and the deductions that comprise its proof. Gypsy provides abstraction mechanisms that can be applied to each of these. Function and type composition provide a way of stating abstract specifications. Function and procedure composition provide a way of stating abstract implementations. Lemmas provide a way of constructing abstract proofs. These mechanisms can be used to attain an arbitrarily high, or low, level of abstraction.

The key to a manageable proof of a routine is to use the abstraction mechanisms in such a way that its specifications, implementation and proof are at nearly the same levels of abstraction. A common mistake is to try to state low level specifications for a high level implementation. When this happens, the specs often begin to dominate the implementation of the routine textually, and the proof is forced to span the distance between these levels. The result can be an overwhelming morass of detail. (The abstraction inversion problem, which is described later in the discussion of distributed systems, is an example of this problem.)

2.6 Verification Environment

Building verified systems in practice requires more than just methods and a language. These methods must be implemented effectively, and for Gypsy, these implementations are contained in the Gypsy Verification Environment (GVE). In spirit, the goal of the GVE is to be a single, well integrated environment that supports all phases of Gypsy program development from initial conception through regular maintenance. The primary responsibility for building and maintaining a verified Gypsy program remains with the human user of the GVE. The purpose of the GVE is to provide tools in a powerful working environment that amplifies human abilities and reduces the probability for human error.

The building of a complete working environment is a very major undertaking, and given the research nature of the Gypsy project, we have chosen to implement only those tools that are most directly involved with program proofs. The tools that presently are available in the GVE are shown in Figure 2. These tools are integrated with respect to a data base that contains all information known to the GVE. This data base contains Gypsy text, cross reference information, verification conditions, proof status information, etc. The user interacts with the GVE executive to invoke the GVE tools. In general, the tools take input from the data base and put results back into the data base. Some of the tools also use ordinary files for input and output.

The GVE executive provides the interactive interface to the user. Considerable attention has been given to developing an effective user interface simply because a bad one can mask the usefulness of even the most powerful tools. A fairly extensive set of help facilities and automatic command prompting are provided.

The major task of the executive, however, is to implement incremental verification. We may be willing to recompile an entire 4000 line program to accommodate a one line change, but we certainly are not going to be willing to reprove it, particularly if substantial parts of the proof are interactive. Therefore, the executive tracks dependencies among specifications, implementations and proofs, and it attempts to preserve as much of the proof as possible as the Gypsy text is modified in response to either initial development or to subsequent maintenance. The first incremental verification component of the executive was described in [Moriconi 77]. It has evolved considerably since then, and it is now being completely redesigned and reimplemented.

The Emacs editor [Stallman 80] is a very sophisticated screen based text editor that can be used either to create or to modify units in the data base. A user may direct Emacs to edit any Gypsy unit. When the user exits from Emacs, the new text is automatically reparsed by the GVE parser and put back into the data base.

The parser reads Gypsy text from either a file or an Emacs buffer and saves it in the data base. One may either define new units or replace the definitions of existing ones. The parser also is called indirectly by other parts of the GVE, most notably the interactive theorem prover. The GVE parser is a table driven LALR parser

Figure 2: Gypsy Verification Environment

```
                                 +---------+
                          +----->|  Emacs  |-------->|
                          |      |  Editor |         |
                          |      +---------+         |
                          |      +---------+         |
                          +----->| Parser  |-------->|
                          |      +---------+         |
                          |      +--------------+    |
                          |----->| VC Generator |--->|
            +-----+       |      +--------------+    |
            |  E  |       |      +----------+        |
            |  X  |       |      | Theorem  |        |
            |  E  |       |----->|  Prover  |------->|
            |  C  |       |      +----------+        |
USER ---->  |  U  | ----->|      +------------+      |
            |  T  |       |----->| Interpreter|----->|
            |  I  |       |      +------------+      |
            |  V  |       |      +----------+        |
            |  E  |       |----->| Compiler |------->|
            +-----+       |      +----------+        |
               |          |           |              |
               |          |           +-->Bliss file |
               |          |      +----------------+  |
               |          |----->| Ada Translator |->|
               |          |      +----------------+  |
               |          |           |              |
               |          |           +-->Ada file   |
               |          |      +-----------+       |
               |          |----->| Optimizer |------>|
               |                 +-----------+       |
               +-------------------------------------+
               |                 |                   |
               v                 v                   v
          +-----------+    +-----------+    +---------+
          | Data Base |    | Algebraic |    | Pretty  |
          |  Handler  |    | Simplifier|    | Printer |
          +-----------+    +-----------+    +---------+
```

that was built with the BOBS-W parser generator [Burger 74].

The verification condition generator produces the vcs that are sufficient to prove any Gypsy unit in the data base. The generator makes two passes. It first constructs the paths of control between adjacent assertions, and then performs a symbolic execution along each path to produce one or more verification conditions. A trace of the construction of the formula also is given so that the verification condition can be related back to the actual Gypsy text. The algebraic simplifier is used to simplify the vcs as they are constructed. When the construction is complete, the vcs are stored in the data base.

Normally, many of the verification conditions are proved automatically in the verification condition generator by the algebraic simplifier. Those formulas that do not simplify to TRUE must be proved with the theorem prover. The theorem prover is an interactive, natural deduction prover that has been adapted from [Bledsoe 74]. The prover is really more like a proof checker than a proof constructor. It provides a list of sound deductive manipulations that a user can apply interactively to the current subgoal. The user directs the prover by telling it what manipulations to perform until the formula is reduced to TRUE. The prover also has some ability to construct proofs and subgoals on its own.

To compile a Gypsy program, the user specifies the name of the top level Gypsy routine where execution is to begin, the name of the target machine (an implementation prelude), and the name of an output file. The compiler then produces a Bliss program on the output file. The first version of this compiler is described in [Smith 80]. Once the Bliss file is created, one must leave the GVE and cross compile the Bliss from the DEC 2060 system (which runs the GVE) to a PDP 11. The PDP 11 target was dictated by application constraints. The compiler presently can produce Bliss for two targets, Unix and an LSI-11.

A translator from Ada to Gypsy also presently is being developed. The main purpose of this translator is to "do something" about verifying Ada programs without rebuilding the entire verification environment and without getting our verification research bogged down in the Ada pit. With the translator, we can use use the regular GVE tools to build verified Gypsy programs and then translate them into Ada. This gives a reasonable claim to at least producing verified Ada programs. The translator also gives us a primitive way of running Gypsy programs on any machine that runs Ada.

A high level optimizer for Gypsy programs also is under development. Programs with the high degree of abstraction that normally is needed for verification usually need more optimization than ordinary programs. At present, the optimizer is focused on in-line expansion of routines, converting formal parameters into global references, eliminating ghost variables, and eliminating run-time checks for exceptions. The optimizer makes full use of Gypsy specifications where it can and produces "optimization conditions" (analogous to verification conditions) which are sufficient to prove that certain optimizations are sound. These optimization conditions then are passed to the algebraic simplifier and theorem prover in the usual way.

A very primitive Gypsy interpreter also has been implemented for the purpose of doing ordinary testing, and eventually, it is intended for more sophisticated symbolic testing. The interpreter, however, has not yet been sufficiently well developed to become a really useful tool. Nevertheless, we believe that it has great potential.

There also are several important GVE components that are not invoked directly with user commands. They are called as utilities by the preceding components. The data base is used by all system components. All information, including Gypsy units, is stored in an internal functional prefix form. Some of the data base is core resident, and some may reside on disk. The pretty printer is used by any component that displays Gypsy text. It converts the internal prefix form into a standard textually formatted form. The algebraic simplifier is one of the most fundamental parts of the GVE. It is used by the executive, the verification condition generator, the theorem prover, the compiler, and the interpreter. At present, the simplifier consists of an ad hoc set of rewrite rules which incorporate the semantics of Gypsy expression evaluation.

The GVE could be extended almost indefinitely to include other tools for program development. For example, one might have tools to support conventional program testing, documentation, configuration management, project management,.... In the foreseeable future, however, we expect the GVE to remain focused on verification issues.

The GVE currently consists of over 50000 lines of ELISP [Hedrick 81] that run on a DEC 2060 computer under the TOPS-20 operating system. ELISP is a version of Lisp that enables use of a 22 bit extended address space on the 2060. The GVE, therefore, is fairly tightly coupled to the 2060, and migration to other machines is not encouraged.

3. Concurrent Processes

The Gypsy methods for specifying and proving distributed systems are based on the methods for concurrent processes, [Good 79], [Good 78b]. Processes in Gypsy are created by a cobegin statement such as

```
COBEGIN
  P(A,B);
  Q(B,C);
END;
```

The COBEGIN is a generalization of an ordinary procedure call. Instead of calling just one procedure sequentially, a COBEGIN calls several procedures which are scheduled and run concurrently. The only requirement that Gypsy makes of the scheduling is fairness -- i.e., each process eventually must get a chance to run. The calling routine is suspended when the concurrent calls are made, and it is resumed only when all of the processes terminate. A COBEGIN that contains just one procedure call is exactly equivalent to an ordinary sequential call.

The scheduling requirements are purposefully weak. The only requirement is fairness, and proofs about COBEGINs assume that the scheduling is completely non-deterministic. Thus, proofs about concurrent processes are valid regardless of the order in which processes are scheduled. In particular, the proofs are valid if the individual processes are run on the same processor or on different processors. This is the basis for proofs about distributed systems. Proofs are made about COBEGIN programs, and then the individual processes are distributed to separate processors.

Inter-process communication in Gypsy is restricted to message passing. Any actual var parameter that is accessible to more than one process under a COBEGIN must be composed wholly of message buffers (a single buffer, an array of buffers, a record with all buffer components, etc.) For example, if processes P and Q above are going to communicate, they each must be passed an actual parameter, such as B, which is a message buffer.

Message buffers are a Gypsy predefined type. For example,

```
TYPE A_MSG_BUFFER = BUFFER (2) OF CHARACTER
```

defines a buffer that contains at most two characters. Buffers are first-come, first-served queues protected by mutual exclusion. The predefined Gypsy procedures for buffers are

```
SEND X TO B;
RECEIVE X FROM B;
```

If buffer B is full when the SEND is called, it is said to be blocked on the buffer. When a SEND is blocked on a buffer, its scheduling is suspended until some other process makes the buffer not full. Similarly, RECEIVE blocks if B is empty. When either SEND or RECEIVE gains access to a buffer, it has exclusive access to it.

Specifications for concurrent processes are stated in terms of buffer histories. These histories are potentially unbounded sequences that record all objects sent to and received from a particular buffer. For each buffer, there are two kinds of histories, "global" and "local." Figure 3 illustrates a case in which processes P and Q are sending to buffer B, and R and S are receiving from it. Two global histories are defined for every buffer B, ALLTO(B) and ALLFROM(B). ALLTO(B) is the sequence of all objects sent to B by all processes that have access to it, and similarly, ALLFROM(B) is the sequence of all objects received from B. These functions satisfy the relation

```
ALLTO(B) = ALLFROM(B) @ CONTENTS(B)
```

where "@" is sequence append and CONTENTS(B) is the sequence of objects currently in buffer B.

Figure 3: Buffer Histories

```
----------------         ----------------
|              |         |              |
|  Process P   |         |  Process Q   |
|              |         |              |
-------G--------         -------H--------
       |                        |
       --------------------------
                    |
                    I
                  -----
                  |   | Buffer B
                   \ /
                    J
                    |
       --------------------------
       |                        |
-------K--------         -------L--------
|              |         |              |
|   Process R  |         |   Process S  |
|              |         |              |
----------------         ----------------

Global Histories: I = ALLTO(B)
                  J = ALLFROM(B)

Local Histories:  G = OUTTO(B,P)
                  H = OUTTO(B,Q)
                  K = INFROM(B,R)
                  L = INFROM(B,S)
```

The same buffer may be accessible to several processes, and then, it usually is necessary to refer to the objects sent to or received from a buffer by a particular process. This is done with the "local" histories, INFROM(B,P) and OUTTO(B,P). The P parameter is called an ACTIVATIONID, and it is the unique identifier of a process. INFROM(B,P) is the sequence of all objects received from buffer B by just process P, and similarly for OUTTO(B,P). The local histories always satisfy the relations

```
OUTTO(B,P) SUB ALLTO(B)
INFROM(B,P) SUB ALLFROM(B)
```

where SUB means "is a subsequence of."

The effect of a COBEGIN is to append a sequence of objects onto the history of each buffer that is passed to any of its procedures. This new sequence is expressed in terms of the histories that are local to the procedures in the COBEGIN, and the effects of each procedure on its local history are described in its external spec. Therefore, the proof of a COBEGIN does not depend on the implementation of the called procedures; it depends only on their external specs. This is how the independence principle is attained for concurrent systems.

One interesting characteristic of concurrent processes is that they often are programmed intentionally never to terminate. In this case, the specification of a desired property when or if the process terminates is of no value. To address this problem, Gypsy has a BLOCK specification. It is an external specification that is to be true whenever the process is blocked. The potential blockage points for a process can be determined by examining the text. SEND and RECEIVE are potential blockage points, and so is a COBEGIN. A COBEGIN is blocked if each of its processes are either blocked or terminated, and at least one of them has not terminated. Calls to procedures with buffer parameters also are potential blockage points. A BLOCK specification is proved by proving that it holds at each of its potential blockage points. For blockage points that are sequential procedure calls or COBEGINS, we assume only that the called procedure(s) meet their BLOCK specifications, and therefore, the proofs of BLOCK specifications can be proved independently for each procedure in the same manner as EXIT specifications. A blockage point can be thought of as a temporary halting point, when the system is completely stable, at which we make an instantaneous assertion about the state of the system. The stability ensures that there is no activity on the buffers, which are the only possible shared objects.

Gypsy buffers can produce the currently popular communicating sequential process (CSP) mechanism [Hoare 78] simply by using buffers of size zero. By the normal operation of SEND and RECEIVE, when a process sends an object to a buffer of size zero, it is blocked until some other process receives the object. When the object is received, then both the sending and receiving processes continue. Similarly, when a process RECEIVEs an object from a buffer of size zero, it is blocked until some other process sends it one. For buffers B of size zero, CONTENTS(B) is always the empty sequence, and the global history axiom reduces to

```
ALLTO(B) = ALLFROM(B)
```

This reduction generally leads to more powerful hypotheses in the verification conditions, and therefore, often to easier proofs.

4. Distributed Systems

A distributed system is viewed in Gypsy as a system of concurrent processes that are distributed over multiple machines. The Gypsy language does not have a formal mechanism for expressing how processes are distributed over machines, so this part of the proof process must be done informally. The semantics of the Gypsy COBEGIN, however, allow a very simple equivalence argument to be applied in many cases.

The non-deterministic scheduling semantics of a COBEGIN allow any of it’s procedures to be distributed. Suppose we have a procedure with a simple COBEGIN form

```
PROCEDURE P(VAR X{ON M}, Y{ON N} :A_BUFFER)=
BEGIN
  VAR U{ON M}, V{ON N}:A_BUFFER;
  COBEGIN
    P1(X,U);
    P2(U,V);
    P3(V,Y);
  END;
END;
```

The important parts of this form are that the procedure contain only local variables and a single COBEGIN. The semantics of the COBEGIN are satisfied, and therefore the proof methods remain valid, regardless of whether COBEGIN procedures run on one machine or span several machines.

We can describe the distribution of this system by deciding what buffers or other data objects reside on what machines. Suppose we have two machines, M and N. The residency decisions are indicated by the Gypsy comments {ON M} and {ON N}. Under this distribution of the data objects, P and P2 are distributed over both machines, but P1 can be isolated to just M and P3 to just N. What we have done is to decompose a distributed process P into a more primitive distributed process P2 and two non-distributed processes.

With this kind of a simple COBEGIN structure, this kind of decomposition can be continued until we reach a level of procedure that does not have the simple COBEGIN structure or we reach a primitive distributed procedure that corresponds to some existing piece of hardware that couples machines. This hardware, for example, might be the cable and appropriate parts of the IO hardware on the two machines. This kind of decomposition leads to a COBEGIN tree such as the one shown in Figure 4. Each non-leaf node in this tree is a procedure with the simple COBEGIN structure shown above.

This COBEGIN tree defines the logical structure of the Gypsy program. However, the only processes in this COBEGIN tree that actually run are the leaf nodes. Therefore, we can get an equivalent program by

Figure 4: Example Cobegin Tree

```
                     P
                     |
   --------------------------------------
   |           |           |            |
   P1          P2          P3           P4
   |                       |            |
-------               -----------     ------
|     |               |    |    |     |    |
P11   P12             P31  P32  P33   P41  P42
      |
   -------
   |     |
   P121  P122
```

grouping together all of the leaf nodes that are isolated to one machine and run them under a single COBEGIN on that machine. We call this new COBEGIN structure the physical structure of the system.

Normally, a distributed program needs to be proved in its logical form, and then manually converted to its physical form to be compiled and run. This is because if one tries to prove the program in its physical form, one encounters the problem of "abstraction inversion." In order to obtain an abstract proof, one wants to view the distributed system as an abstract concurrent program operating on high level abstract objects. In its physical form, however, the parameters to the top level procedure, the one that consists of the COBEGIN of the leaf processes that actually run, are not high level abstract objects but the very low level objects that interface with the lowest level primitive distributed processes. To write specs for this program, we would need to write specs for high level operations, but the abstraction levels of the data objects that are available are exactly inverted. Instead of having high level objects available to us, we have some of the most primitive ones. Attempts to write these kinds of specifications can lead to a terrible mess! By contrast, the processes in their logical form can be distributed so that the abstraction levels remain closely synchronized, and we can obtain much more tractable specifications and proofs.

5. A Distributed Two Channel Mover

We will now illustrate the use of Gypsy on a distributed two channel message mover that multiplexes and demultiplexes messages across a single physical channel connecting two machines M and N. The Gypsy text for this example is given in Appendix A. This text defines the logical structure of the mover, and the specifications and proofs will be done on this structure. Then we will convert this logical structure to an equivalent physical structure.

Normally, all of the activity of developing and proving Gypsy programs is done within the GVE. Due to space limitation, however, we will not include a detailed transcript of the GVE sessions. We will, however, include excerpts from those sessions in the discussion that follows. A complete transcript of the GVE session appears in [Good 82]. The final version of this transcript required 16 minutes of CPU time and 54 minutes of real time.

5.1 The Mover Procedure

The two channel mover is contained in scope TWO_CHANNEL_MOVER. The top level procedure is MOVER, and the next level procedures are MUX, TAGGED_MOVER and DEMUX.

The structure of MOVER is shown in Figure 5. MOVER is to implement two logically separate message channels, A and B, from machine M to machine N. The machines, however, are connected only by a single physical channel. MOVER connects four buffers. INA and INB reside on machine M, and OUTA and OUTB reside on machine N, and thus, MOVER is distributed over both machines.

Figure 5: Two Channel Mover

```
+---------Q-------------------------------S----------------+  ^
|         |            MOVER              |                |  |
|         v                               v                |
|        INA      BLOCK P SUB Q          INB               |  M
|         |           & R SUB S           |                |  A
|         |                               |                |  C
|    +----Q-------------------------------S----+           |  H
|    |                  MUX                    |           |  I
|    |       BLOCK CHANNEL_SEQ(A,Z) SUB Q      |           |  N
|    |           & CHANNEL_SEQ(B,Z) SUB S      |           |  E
|    +-------------------Z---------------------+           |
|                        |                                 |  M
|                    TAGGED_IN                             |
|                        |                                 |  |
|                +-------Y-------+                         |  v
|                |               |                         | ---
|                |               |                         |  ^
|                |               |                         |  |
|                |               |  USE LEMMA              |  C
|                | TAGGED_MOVER  |  W SUB Z ->             |  A
|                | BLOCK X SUB Y |    CHANNEL_SEQ(C,W)     |  B
|                |               |    SUB CHANNEL_SEQ(C,Z) |  L
|                |               |                         |  E
|                |               |                         |  |
|                |               |                         |  v
|                |               |                         | ---
|                +-------X-------+                         |  ^
|                        |                                 |  |
|                    TAGGED_OUT                            |
|                        |                                 |  M
|    +-------------------W---------------------+           |  A
|    |                 DEMUX                   |           |  C
|    |      BLOCK P SUB CHANNEL_SEQ(A,W)       |           |  H
|    |          & R SUB CHANNEL_SEQ(B,W)       |           |  I
|    +----P-------------------------------R----+           |  N
|         |                               |                |  E
|         v                               v                |
|        OUTA                           OUTB               |  N
|         |                               |                |  |
+---------P-------------------------------R----------------+  v
```

The specs for procedure MOVER are very straightforward. First, we specify MOVER to be non-terminating by specifying EXIT FALSE. The block spec, which is illustrated in Figure 5, says that each message channel is to deliver a subsequence of the messages it receives. This allows each channel to drop messages (due to hardware failure or whatever), but the messages that are delivered must be delivered exactly as they were received, they must be delivered in the same order they were received, and messages may not cross channels.

MOVER is composed of a COBEGIN of three concurrent processes. The multiplexor process, MUX, receives messages from the input buffers of the two channels, tags them with a channel identifier, and puts them into the single buffer TAGGED_IN. TAGGED_MOVER is a distributed, single channel that moves tagged messages from buffer TAGGED_IN on machine M to buffer TAGGED_OUT on machine N. The demultiplexor process, DEMUX, receives messages from TAGGED_OUT and routes them to the output buffer designated by their tags. This decomposes the distributed MOVER process into two non-distributed processes, MUX and DEMUX, and another distributed process, TAGGED_MOVER.

Before we can construct a proof of MOVER, we must have external specifications for MUX, TAGGED_MOVER and DEMUX. These also are illustrated in Figure 5, and they are discussed in greater detail in the subsequent sections. (It is important that the reader understand these specifications, before trying to understand the following proof.)

The GVE verification condition generator constructs two vcs for the proof of MOVER. One corresponds to the case when the COBEGIN is blocked and the other is for the potential termination of the COBEGIN. The latter is the proof that MOVER is a non-terminating process. Because we have specified that each procedure in the COBEGIN also is non-terminating, this proof is easy, and it is done automatically by algebraic simplification in the vc generator.

The vc that is constructed by the GVE for the block spec of MOVER is

```
    H1: ALLFROM (TAGGED_IN)
        = INFROM (TAGGED_IN, TAGGED_MOVER)
    H2: ALLFROM (TAGGED_OUT) = INFROM (TAGGED_OUT, DEMUX)
    H3: ALLTO (TAGGED_IN) = OUTTO (TAGGED_IN, MUX)
    H4: ALLTO (TAGGED_OUT)
        = OUTTO (TAGGED_OUT, TAGGED_MOVER)
    H5: ISBLOCKED (DEMUX)
    H6: ISBLOCKED (MUX)
    H7: ISBLOCKED (TAGGED_MOVER)
    H8: CHANNEL_SEQ (A, OUTTO (TAGGED_IN, MUX))
                SUB INFROM (INA, MUX)
    H9: CHANNEL_SEQ (B, OUTTO (TAGGED_IN, MUX))
          SUB INFROM (INB, MUX)
    H10: OUTTO (OUTA, DEMUX) SUB
           CHANNEL_SEQ (A, INFROM (TAGGED_OUT, DEMUX))
    H11: OUTTO (OUTB, DEMUX) SUB
           CHANNEL_SEQ (B, INFROM (TAGGED_OUT, DEMUX))
    H12: OUTTO (TAGGED_OUT, TAGGED_MOVER)
         SUB INFROM (TAGGED_IN, TAGGED_MOVER)
  ->
    C1: OUTTO (OUTA, DEMUX) SUB INFROM (INA, MUX)
    C2: OUTTO (OUTB, DEMUX) SUB INFROM (INB, MUX)
```

(The reader may find it very helpful to interpret this formula in terms of Figure 5.)

This vc can be proved as follows with the interactive prover. By deleting hypotheses H5 - H7, performing equality substitutions based on H1 - H4 and renumbering hypotheses, the vc is reduced to

```
    H1: ALLTO (TAGGED_OUT) SUB ALLFROM (TAGGED_IN)
    H2: CHANNEL_SEQ (A, ALLTO (TAGGED_IN))
```

```
         SUB INFROM (INA, MUX)
   H3: CHANNEL_SEQ (B, ALLTO (TAGGED_IN))
         SUB INFROM (INB, MUX)
   H4: OUTTO (OUTA, DEMUX)
         SUB CHANNEL_SEQ (A, ALLFROM (TAGGED_OUT))
   H5: OUTTO (OUTB, DEMUX)
         SUB CHANNEL_SEQ (B, ALLFROM (TAGGED_OUT))
 ->
   C1: OUTTO (OUTA, DEMUX) SUB INFROM (INA, MUX)
   C2: OUTTO (OUTB, DEMUX) SUB INFROM (INB, MUX)
```

Next we direct the prover to prove the following subgoal:

```
   H1: ALLTO (TAGGED_OUT) SUB ALLFROM (TAGGED_IN)
 ->
   C1: ALLFROM (TAGGED_OUT) SUB ALLTO (TAGGED_IN)
```

This is proved automatically using knowledge built into the algebraic simplifier about the global buffer histories and transitivity of subsequence relations. With this subgoal proved, it can be used to replace hypothesis H1 in the previous subgoal to obtain

```
   H1: ALLFROM (TAGGED_OUT) SUB ALLTO (TAGGED_IN)
   H2: CHANNEL_SEQ (A, ALLTO (TAGGED_IN))
         SUB INFROM (INA, MUX)
   H3: CHANNEL_SEQ (B, ALLTO (TAGGED_IN))
         SUB INFROM (INB, MUX)
   H4: OUTTO (OUTA, DEMUX)
         SUB CHANNEL_SEQ (A, ALLFROM (TAGGED_OUT))
   H5: OUTTO (OUTB, DEMUX)
         SUB CHANNEL_SEQ (B, ALLFROM (TAGGED_OUT))
 ->
   C1: OUTTO (OUTA, DEMUX) SUB INFROM (INA, MUX)
   C2: OUTTO (OUTB, DEMUX) SUB INFROM (INB, MUX)
```

We now reach the key step in the proof, the use of the CHANNEL_SUB_SEQ lemma. See Figure 5.

```
 LEMMA CHANNEL_SUB_SEQ (C : A_CHANNEL;
                     R, S : A_TAGGED_MSG_SEQ)=
   [ASSUME S SUB R ->
          CHANNEL_SEQ (C, S) SUB CHANNEL_SEQ (C, R)];
```

This says that if S is a subsequence of R, then the result of extracting the channel C messages from S is a subsequence of extracting channel C messages from R. We must manually "use" this lemma and obtain a new hypothesis H6,

```
   H1: ALLFROM (TAGGED_OUT) SUB ALLTO (TAGGED_IN)
   H2: CHANNEL_SEQ (A, ALLTO (TAGGED_IN))
         SUB INFROM (INA, MUX)
   H3: CHANNEL_SEQ (B, ALLTO (TAGGED_IN))
         SUB INFROM (INB, MUX)
   H4: OUTTO (OUTA, DEMUX)
         SUB CHANNEL_SEQ (A, ALLFROM (TAGGED_OUT))
   H5: OUTTO (OUTB, DEMUX)
         SUB CHANNEL_SEQ (B, ALLFROM (TAGGED_OUT))
   H6: S$ SUB R$ ->
       CHANNEL_SEQ (C$,S$) SUB CHANNEL_SEQ (C$,R$)
 ->
   C1: OUTTO (OUTA, DEMUX) SUB INFROM (INA, MUX)
   C2: OUTTO (OUTB, DEMUX) SUB INFROM (INB, MUX)
```

The symbols with the suffix "$" are how the prover handles the logical quantifiers (for all and there exists). Hypothesis H6 is true for all values that might be substituted for "S$", "R$", and "C$."

The next step is to make the right substitutions. The substitutions for S$ and R$ can be done automatically by the prover once it is manually directed to apply a modus ponens argument to H1 and H6. The substitution for C$ must be done manually through a "put" command. After we do these steps for both channels, and the subgoal is cleaned up slightly, we have

```
  H1: OUTTO (OUTA, DEMUX) SUB
        CHANNEL_SEQ (A, ALLFROM (TAGGED_OUT))
  H2:   CHANNEL_SEQ (A, ALLFROM (TAGGED_OUT))
          SUB CHANNEL_SEQ (A, ALLTO (TAGGED_IN))
  H3:         CHANNEL_SEQ (A, ALLTO (TAGGED_IN))
              SUB INFROM (INA, MUX)
  H4: OUTTO (OUTB, DEMUX) SUB
        CHANNEL_SEQ (B, ALLFROM (TAGGED_OUT))
  H5:   CHANNEL_SEQ (B, ALLFROM (TAGGED_OUT))
          SUB CHANNEL_SEQ (B, ALLTO (TAGGED_IN))
  H6:         CHANNEL_SEQ (B, ALLTO (TAGGED_IN))
                SUB INFROM (INB, MUX)
->
  C1: OUTTO (OUTA, DEMUX) SUB INFROM (INA, MUX)
  C2: OUTTO (OUTB, DEMUX) SUB INFROM (INB, MUX)
```

The prover now can complete the proof automatically using transitivity of subsequences.

5.2 The Multiplexor Procedure

The specs for MUX are somewhat more subtle than the ones for MOVER. In writing specifications, we have found that there often is a strong tendency to over specify. When we don’t know what else to specify, we set about specifying everything! This often leads to very complicated and unwieldy specifications and to investing considerable effort in creating specs only to find that they are not in a form that supports a tractable proof, or even worse, that they are not required in the proof at all!

MUX is a simple example of such a temptation. We are tempted to try to think of OUTTO(MUX_OUT,MYID) as some function of INFROM(INA,MYID) and INFROM(INB,MYID). But, if we pursue this, we find that we need to know the relative order in which the messages from the two channels were received in order to state such a function. The Gypsy time stamping mechanism would allow us to do this, but the specification becomes unnecessarily complex.

The specifications for both MUX and DEMUX can be stated quite simply by defining a specification function.

```
FUNCTION CHANNEL_SEQ (C : A_CHANNEL;
                      S : A_TAGGED_MSG_SEQ) : A_MSG_SEQ =
BEGIN
EXIT (ASSUME RESULT =
      IF S = NULL(A_TAGGED_MSG_SEQ) THEN NULL(A_MSG_SEQ)
      ELSE IF LAST(S).TAG = C THEN
              CHANNEL_SEQ(C,NONLAST(S)) <: LAST(S).DATA
           ELSE CHANNEL_SEQ(C,NONLAST(S))
      FI   FI);
END {CHANNEL_SEQ};
```

This function extracts, in order, messages tagged C from the sequence S. With this specification function, the external specifications for MUX can be stated quite simply. They say that if we examine the sequence of all messages tagged A that were sent out to buffer MUX_OUT, then that is a subsequence of all messages received in from buffer INA. The specification for channel B is similar.

The multiplexor is implemented as a simple loop that continuously polls buffers INA and INB. To prove MUX, the vc generator constructs a total of seven vcs. One is proved automatically in the vc generator, and the rest must be proved with the interactive prover. One of these is the proof that the ASSERT is true the first time it is reached. The local histories are all empty at this point, and the vc is proved easily by expanding the definition of CHANNEL_SEQ. The remaining six vcs are for the various paths around the loop. The proofs of all of these are done in essentially the same way.

The proof of the vc that corresponds to a loop cycle in which a message is received from INA, but not from INB, is proved as follows. We delete irrelevant hypotheses and get

```
H1: CHANNEL_SEQ (A, OUTTO (MUX_OUT, MYID))
    SUB INFROM (INA, MYID)
H2: CHANNEL_SEQ (B, OUTTO (MUX_OUT, MYID))
    SUB INFROM (INB, MYID)
->
C1: CHANNEL_SEQ
    (A, OUTTO (MUX_OUT, MYID)
        @ [SEQ: M WITH (.DATA:=M.DATA; .TAG:=A)])
    SUB INFROM (INA, MYID) @ [SEQ: M.DATA]
C2: CHANNEL_SEQ
    (B, OUTTO (MUX_OUT, MYID)
        @ [SEQ: M WITH (.DATA:=M.DATA; .TAG:=A)])
    SUB INFROM (INB, MYID)
```

We now expand the definitions of CHANNEL_SEQ in the conclusions and get

```
  H1: CHANNEL_SEQ (A, OUTTO (MUX_OUT, MYID))
      SUB INFROM (INA, MYID)
  H2: CHANNEL_SEQ (B, OUTTO (MUX_OUT, MYID))
      SUB INFROM (INB, MYID)
->
  C1: CHANNEL_SEQ (A, OUTTO (MUX_OUT, MYID))
                            @ [SEQ: M.DATA]
      SUB INFROM (INA, MYID) @ [SEQ: M.DATA]
  C2: CHANNEL_SEQ (B, OUTTO (MUX_OUT, MYID))
      SUB INFROM (INB, MYID)
```

The proof is now completed automatically based on built in knowledge about sequences.

In creating MUX, a typographical error was made, quite unintentionally. The polling segment for channel B was copied from the one for channel A and then modified for channel B except that the reference to buffer INA was not changed to INB. This meant that some channel A messages would have been delivered to channel B -- exactly what we wanted not to happen! In the initial construction of MUX vcs, four of the seven were provable, but three of them were not. Upon studying why they were not provable, the error was discovered.

The error was corrected by using Emacs to edit just the MUX procedure. The top level executive then noted that the vcs for MUX needed to be regenerated. It did this and obtained seven new vcs. It automatically noted that two of these were isomorphic to ones that had been proved previously, and therefore, no additional proof was needed for these two. It also noted that this change to MUX did not invalidate the previous proof of MOVER. The four new vcs then were proved in the same general manner as the one shown above.

5.3 The Demultiplexor Procedure

The specs for DEMUX also are expressed in terms of CHANNEL_SEQ. They say that what we send out to buffer OUTA is a subsequence of the messages tagged A that were received in from buffer DEMUX_IN, and similarly for channel B. The demultiplexor is implemented as a simple switch with a loop that cycles through a case statement.

To prove DEMUX, the vc generator constructed five vcs. One of them was proved automatically in the vc generator. These proofs all proceed in essentially the same way. The proof for the case in which we receive a message tagged A and block on the SEND statement is done as follows. By retaining only relevant hypotheses, we get

```
    H1: M.TAG = A
    H2: OUTTO (OUTA, MYID) SUB
        CHANNEL_SEQ (A, INFROM (DEMUX_IN, MYID))
    H3: OUTTO (OUTB, MYID) SUB
        CHANNEL_SEQ (B, INFROM (DEMUX_IN, MYID))
  ->
    C1: OUTTO (OUTA, MYID) SUB
        CHANNEL_SEQ (A, INFROM (DEMUX_IN, MYID)
                        @ [SEQ: M])
    C2: OUTTO (OUTB, MYID) SUB
        CHANNEL_SEQ (B, INFROM (DEMUX_IN, MYID)
                        @ [SEQ: M])
```

As in the proof of MUX, the key step is to expand the definition of CHANNEL_SEQ in the conclusions. This gives

```
    H1: M.TAG = A
    H2: OUTTO (OUTA, MYID) SUB
        CHANNEL_SEQ (A, INFROM (DEMUX_IN, MYID))
    H3: OUTTO (OUTB, MYID) SUB
        CHANNEL_SEQ (B, INFROM (DEMUX_IN, MYID))
  ->
    C1: OUTTO (OUTA, MYID) SUB
        IF M.TAG = A THEN
          CHANNEL_SEQ (A, INFROM (DEMUX_IN, MYID))
                    @ [SEQ: M.DATA]
        ELSE CHANNEL_SEQ (A, INFROM (DEMUX_IN, MYID))
        FI
    C2: OUTTO (OUTB, MYID) SUB
        IF M.TAG = B THEN
          CHANNEL_SEQ (B, INFROM (DEMUX_IN, MYID))
                    @ [SEQ: M.DATA]
        ELSE CHANNEL_SEQ (B, INFROM (DEMUX_IN, MYID))FI
```

The prover now can complete the proof automatically.

5.4 The Channel_sub_seq Lemma

Our normal procedure for proving a routine is first to prove its vcs from certain lemmas. This keeps the proof at an abstract level. Then, we descend a level and prove the lemmas that were used. These proofs might be further decomposed by stating and using additional lemmas and then turning our attention to proving these additional ones. By continuing in this fashion, very complex proofs can be decomposed effectively into steps that are small enough to be proved by a mechanical prover. Eventually, the proofs should be decomposed to the point where they are based only on an acceptable and consistent set of axioms.

Our proofs of MUX and DEMUX used only the definition CHANNEL_SEQ, but the proof of MOVER used the CHANNEL_SUB_SEQ lemma. We now need to prove that the lemma follows from the definition of CHANNEL_SEQ. This particular lemma yields very simply to a proof by structural induction. Unfortunately, GVE prover does not have the capability for induction proofs, and therefore, insofar as Gypsy is concerned, we would have to leave this lemma as an assumption.

This lemma, however, is not beyond the scope of mechanical verification. It is the kind of theorem that can be proved quite effectively by the Boyer-Moore theorem prover [Boyer 79]. So, to complete the proof of the two channel mover, this lemma and the definition of channel_seq were transformed manually into formulas in the Boyer-Moore theory, and the lemma was proved easily by the mechanical prover.

5.5 One Channel Mover

The specifications for TAGGED_MOVER, which is the MOVER procedure in scope ONE_CHANNEL_MOVER, are almost identical to the spec for each channel of the two channel mover. The key difference is that the one channel MOVER moves tagged messages.

MOVER is implemented by three concurrent processes, DISASSEMBLER, CHAR_MOVER and REASSEMBLER. The DISASSEMBLER process disassembles tagged messages into streams of single characters. CHAR_MOVER is a primitive distributed process that moves single characters across a hardware interface between machines M and N according to a subsequence property. (The subsequence property allows the device to drop characters.) The REASSEMBLER process reassembles streams of individual characters back into tagged messages. This decomposition isolates DISASSEMBLER to machine M and REASSEMBLER to machine N.

For purposes of this paper, the functional specifications, implementations and proofs of DISASSEMBLER and REASSEMBLER are left pending. The completion of these, however, is nontrivial because of the potentially faulty CHAR_MOVER process. We must obtain reliable transmissions across a potentially unreliable interface. The Gypsy proof methods have been used directly on communications protocols such as these [DiVito 81], but more effective methods for proving protocols are described in [DiVito 82]. A very important aspect of our decomposition, however, is that we have isolated these difficulties to this very low level of abstraction, and they manifest themselves in the higher levels only by virtue of the subsequence relations on message sequences.

5.6 Physical Structure

The two channel mover has a simple cobegin structure that can easily be converted into an equivalent physical structure as described previously. These two structures are shown in Figure 6. MUX and DISASSEMBLER have been isolated to machine M, REASSEMBLER and DEMUX have been isolated to machine N, and CHAR_MOVER is primitive. FOR_M and FOR_N indicate the cobegin programs that would be compiled and run for those respective machines. (To fully appreciate the problem of abstraction inversion, we suggest that the reader try to specify and prove the physical mover.)

Automation of this transition from logical to equivalent physical program remains an intriguing research topic. One approach would be to extend Gypsy to describe a physical, target architecture and to specify how the logical program is to be mapped onto that target. The GVE would verify that the specified mapping was possible, and then the compiler would use these specifications to produce code for the appropriate target machines.

6. Conclusion

The preceding sections describe the basic principles of specifying, implementing and verifying concurrent, distributed systems in Gypsy and how these methods are supported by the GVE. The two channel message mover illustrates these methods on a small, but realistic, example. These are the same methods that were used to prove the 4211 lines of the special interface between the host and the IMP on the ARPANET. By way of comparison, our proof of the message mover covers 35 lines (of executable code).

Gypsy is unique in its comprehensive approach to building verified systems. Gypsy contains well integrated methods for specifying, implementing and verifying programs, and these are implemented in a powerful verification environment. The Gypsy proof methods are based on the fundamental ideas of [Naur 66], [Floyd 67] and [Hoare 69]. These methods have been developed to the point where they cover the complete Gypsy programming language.

Figure 6: Mover Cobegin Tree

```
                   LOGICAL
                    MOVER
                      |
 ---------------------------------------------
 |                    |                      |
MUX              TAGGED_MOVER              DEMUX
                      |
        ------------------------------
        |             |              |
   DISASSEMBLER   CHAR_MOVER    REASSEMBLER


                   PHYSICAL
                    MOVER
                      |
     -----------------------------------
     |                |                 |
   FOR_M          CHAR_MOVER          FOR_N
     |                                  |
 ----------                         -----------
 |        |                         |         |
MUX  DISASSEMBLER                REASSEMBLER  DEMUX
```

Gypsy was designed by beginning with a semantic subset of Pascal [Jensen 74], and then extending it to include exception handling, dynamic storage management without pointers, data abstraction, concurrency that supports distributed systems, and specifications. The programming extensions were made in conformity with the principles of structured programming [Dijkstra 72]. The subset was extended only if we could define precise methods for specifying and proving the extensions. The constraint of tractable, mechanical construction of verification conditions also was imposed. The Gypsy data structures were influenced strongly by [Hoare 72]. The result of this design is a powerful, provable language with a broad range of potential applications.

One of the most important aspects of Gypsy is its abstraction mechanisms. The need for abstraction has been stated very clearly.

> Summarizing: as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure [Dijkstra 72].

The Gypsy abstraction mechanisms provide the way of decomposing the proof of a large problem into pieces that are small enough to think about. The previous work on structured programming has generally recognized the need for abstraction in implementation. In order to construct tractable proofs, abstraction also is necessary in specification and proof. Gypsy provides mechanisms for constructing abstract specifications, abstract implementations, and abstract proofs, and the same methods are used at all levels of abstraction.

Abstraction breaks the large problem into many small pieces, and the GVE manages and assists in the proof of those pieces. The GVE is built in the spirit of [Good 70] and [Good 75]. It is not intended to be a fully automatic program verifier. That is expecting too much. Its purpose is to provide a powerful working environment that amplifies human proof capability and reduces the probability of human error. To do this, the GVE provides a wide range of fully integrated tools that support incremental verification. The tools include a very sophisticated screen editor, parser, verification condition generator, interactive theorem prover, compiler, optimizer, Ada translator and interpreter. This range of tools and the power of the Gypsy language provide a uniquely comprehensive approach to building verified systems.

Gypsy also is unique in its mechanical verification accomplishments. The ARPANET interface application is very encouraging, and we interpret this as strong evidence for the technical feasibility of deductively proving significant, real programs in practice. The day of routine mechanical verification of programs, however, has not yet come. The current verification methods and tools can be very expensive, both in terms of computer costs, and more importantly, in terms of human costs. These costs need to be reduced dramatically, probably by a factor of a 100 or more. If this can be done, then mechanical verification may very well become an accepted part of conventional programming practice and lead to a major advance in the quality of computing systems.

Appendix A
Gypsy Text for Two Channel Mover

```
scope two_channel_mover =
begin

  PROCEDURE MOVER (var  ina,  inb {on M} : a_msg_inbuf;
                   var outa, outb {on N} : a_msg_outbuf) =
  begin
  block outto (outa, myid) sub infrom (ina, myid)
      & outto (outb, myid) sub infrom (inb, myid);
  exit false;
    var tagged_in  {on M},
        tagged_out {on N} : a_tagged_msg_buf;
    cobegin
      mux (ina, inb, tagged_in);
      tagged_mover (tagged_in, tagged_out);
      demux (tagged_out, outa, outb);
    end {cobegin};
  end {MOVER};

  PROCEDURE MUX (var ina, inb {on M} : a_msg_inbuf;
                 var mux_out  {on M} : a_tagged_msg_outbuf) =
  begin
  block channel_seq (a, outto (mux_out, myid))
          sub infrom (ina, myid)
      & channel_seq (b, outto (mux_out, myid))
          sub infrom (inb, myid);
  exit false;
    var m : a_tagged_msg;
    loop {polling input buffers}
      assert channel_seq (a, outto (mux_out, myid))
              sub infrom (ina, myid)
           & channel_seq (b, outto (mux_out, myid))
              sub infrom (inb, myid);
      if not empty (ina) then
        receive m.data from ina;
        m.tag := a; send m to mux_out;
      end {if};
      if not empty (inb) then
        receive m.data from inb;
        m.tag := b; send m to mux_out;
      end {if};
    end {loop};
  end {MUX};

  PROCEDURE DEMUX (var   demux_in {on N} : a_tagged_msg_inbuf;
                   var outa, outb {on N} : a_msg_outbuf) =
  begin
  block outto (outa, myid)
          sub channel_seq (a, infrom (demux_in, myid))
      & outto (outb, myid)
          sub channel_seq (b, infrom (demux_in, myid));
  exit false;
    var m : a_tagged_msg;
    loop
      assert outto (outa, myid)
               sub channel_seq (a,infrom (demux_in, myid))
           & outto (outb, myid)
               sub channel_seq (b,infrom (demux_in, myid));
      receive m from demux_in;
      case m.tag
        is a: send m.data to outa;
        is b: send m.data to outb;
      end {case};
   end {loop};
  end {DEMUX};


  function channel_seq (c : a_channel;
                        s : a_tagged_msg_seq) : a_msg_seq =
  begin
  exit (assume result =
        if s = null(a_tagged_msg_seq) then null(a_msg_seq)
        else if last(s).tag = c then
                channel_seq(c,nonlast(s)) <: last(s).data
             else channel_seq(c,nonlast(s))
        fi   fi);
  end {channel_seq};

  lemma channel_sub_seq (c : a_channel;
                      r, s : a_tagged_msg_seq)=
    [assume s sub r ->
            channel_seq (c, s) sub channel_seq (c, r)];

  const buffer_size:integer = pending;

  type a_channel = (a,b);

  type a_msg_buf = buffer (buffer_size) of a_msg;
  type a_msg_inbuf = a_msg_buf<input>;
  type a_msg_outbuf = a_msg_buf<output>;
  type a_msg_seq = sequence of a_msg;

  name a_msg, a_tagged_msg, a_tagged_msg_buf, a_tagged_msg_inbuf,
       a_tagged_msg_outbuf, a_tagged_msg_seq,
       tagged_mover = mover
       from one_channel_mover;

end {scope two_channel_mover};
{
   }
scope one_channel_mover =
begin

  PROCEDURE MOVER(var x {on M} : a_tagged_msg_inbuf;
                  var y {on N} : a_tagged_msg_outbuf) =
  begin
  block outto (y, myid) sub infrom (x, myid);
  exit false;
    var u,v:a_char_buf;
    cobegin
      disassembler(x,u);
      char_mover(u,v);
      reassembler(v,y);
    end;
  end {MOVER};

  PROCEDURE DISASSEMBLER(var x {on M} : a_tagged_msg_inbuf;
                         var y {on M} : a_char_outbuf) =
  pending;

  PROCEDURE REASSEMBLER(var x {on N} : a_char_inbuf;
                        var y {on N} : a_tagged_msg_outbuf) =
  pending;

  const buffer_size:integer = pending;

  type a_msg = pending;

  type a_tagged_msg = record(data:a_msg; tag:a_channel);
  type a_tagged_msg_buf = buffer (buffer_size) of a_tagged_msg;
  type a_tagged_msg_inbuf = a_tagged_msg_buf<input>;
  type a_tagged_msg_outbuf = a_tagged_msg_buf<output>;
  type a_tagged_msg_seq = sequence of a_tagged_msg;

  name a_channel from two_channel_mover;

  name a_char, a_char_buf, a_char_inbuf, a_char_outbuf,
       char_mover = mover
       from character_mover;

end {one_channel_mover};
{
}
scope character_mover =
begin

  PROCEDURE MOVER(var x {on M} : a_char_inbuf;
                  var y {on N} : a_char_outbuf) =
  begin
  block [assume outto (y, myid) sub infrom (x, myid)];
  exit [assume false];
    {This is assumed to be a primitive procedure, implemented
     in hardware, that moves one character from machine
     M to N.}
  end {MOVER};

  const buffer_size:integer = pending;

  type a_char = pending;
  type a_char_buf = buffer (buffer_size) of a_char;
  type a_char_inbuf = a_char_buf<input>;
  type a_char_outbuf = a_char_buf<output>;

  name a_msg from one_channel_mover;

end;
```

References

[Bledsoe 74] W.W. Bledsoe, P. Bruell.
A Man-Machine Theorem-Proving System.
In *Advance Papers of Third International Joint Conference on Artificial Intelligence*. W.W. Bledsoe, 5-1 (Spring), 1974.

[Boyer 79] R. S. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

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

[Dijkstra 72] E.W. Dijkstra.
Structured Programming.
Academic Press, 1972, Chapter Notes on Structured Programming.

[DiVito 81] B. L. DiVito.
A Mechanical Verification of the Alternating Bit Protocol.
Technical Report ICSCA-CMP-21, University of Texas at Austin, June, 1981.

[DiVito 82] B. L. DiVito.
Verification of Communications Protocols and Abstract Process Models.
PhD thesis, University of Texas at Austin, 1982.
Technical Report 25, Institute for Computing Science.

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

[Good 70] D.I. Good.
Toward a Man-Machine System for Proving Program Correctness.
PhD thesis, University of Wisconsin, 1970.

[Good 75] D.I. Good, R.L. London, W.W. Bledsoe.
An Interactive Program Verification System.
In *Proceedings of 1975 International Conference on Reliable Software*. D.I. Good, 1975.

[Good 78a] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

[Good 78b] D. I. Good and R. M. Cohen.
Verifiable Communications Processing in Gypsy.
In *Proceedings of Compcon '78*. IEEE, September, 1978.

[Good 79] D. I. Good, R. M. Cohen, and J. Keeton-Williams.
Principles of Proving Concurrent Programs in Gypsy.
In *Proceedings of 6th Symposium of Principles of Programming Languages*. ACM, January, 1979.

[Good 82]	Donald I. Good, Robert S. Boyer. Proof Logs of Two Channel Message Mover. Internal Note #44, Institute for Computing Science, The University of Texas at Austin. August, 1982
[Hedrick 81]	Charles L. Hedrick. *ELISP: A Large Address Space Implementation of LISP for the DECSYSTEM-20* Rutgers University, 1981.
[Hoare 69]	C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM* 12-10, 1969.
[Hoare 72]	C.A.R. Hoare. *Structured Programming.* Academic Press, 1972, Chapter Notes on Data Structuring.
[Hoare 78]	C. A. R. Hoare. Communicating Sequential Processes. *CACM* 21(8), August, 1978.
[Jensen 74]	K. Jensen, N. Wirth. *Pascal User Manual and Report.* Springer Verlag, 1974.
[Moriconi 77]	Mark S. Moriconi. *A System for Incrementally Designing & Verifying Programs*. PhD thesis, The University of Texas at Austin, 1977. ICSCA-CMP-9.
[Naur 66]	P. Naur. Proof of Algorithms by General Snapshots. *BIT* 6, 4, 1966.
[Smith 80]	Lawrence Mark Smith. Compiling from the Gypsy Verification Environment. Master's thesis, The University of Texas at Austin, 1980. Technical Report 20, Institute for Computing Science.
[Stallman 80]	Richard M. Stallman. *EMACS Manual For Twenex Users* MIT Artificial Intelligence Laboratory, 1980.

THE PROOF OF A DISTRIBUTED SYSTEM IN GYPSY

Table of Contents

List of Figures