Mechanical Proofs about Computer Programs

Donald I. Good

Technical Report #36 March 1, 1984

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This report is an unrevised edition of Technical Report 41, Institute for Computing Science and Computer Applications, The University of Texas at Austin. The original paper was presented at a meeting for Discussion of Mathematical Logic and Programming Languages The Royal Society of London February 15-16, 1984.

Abstract

The Gypsy verification environment is a large computer program that supports the development of software systems and formal, mathematical proofs about their behavior. The environment provides conventional development tools, such as a parser for the Gypsy language, an editor and a compiler. These are used to evolve a library of components that define both the software and precise specifications about its desired behavior. The environment also has a verification condition generator that automatically transforms a software component and its specification into logical formulas which are sufficient to prove that the component always runs according to specification. Facilities for constructing formal, mechanical proofs of these formulas also are provided. Many of these proofs are completed automatically without human intervention. The capabilities of the Gypsy system and the results of its applications are discussed.

Acknowledgements

The development and initial experimental applications of Gypsy have been sponsored in part by the U. S. Department of Defense Computer Security Center (Contracts MDA904-80-C-0481, MDA904-82-C-0445), by the U. S. Naval Electronic Systems Command (Contract N00039-81-C-0074), by Digital Equipment Corporation, by Digicomp Research Corporation and by the National Science Foundation (Grant MCS-8122039).

1. Introduction

One of the major problems with the current practice of software engineering is an absence of predictability. There is no sound, scientific way of predicting accurately how a software system will behave when it runs. There are many compelling examples of important software systems that have behaved in unpredictable ways. A space shuttle fails to launch. An entire line of automobiles is recalled because of problems with the software that controls the braking system. Unauthorized users get access to computer systems. Sensitive information passes into the wrong hands. The list goes on and on [Neumann 83a, Neumann 83b]. Considering the wide variety of tasks that now are entrusted to computer systems, it is truly remarkable that it is not possible to predict accurately what they are going to do!

Within current software engineering practice, the only sound way to make a precise, accurate prediction about how a software system will behave is to build it and run it. There is no way to predict accurately how a system will behave before it can be run. Thus, design flaws often are detected only after a large investment has been made to develop the system to a point where it can be run. The rebuilding that is caused by the late detection of these flaws contributes significantly to the high cost of software construction and maintenance. Even after the system can be run, the situation is only slightly better. A system that can be run can be tested on a set of trial cases. If the system is deterministic, a trial run on a specific test case provides a precise, accurate prediction about how the system will behave in *that one case*. If the system is rerun on the exact same case, it will behave in the exact same way. However, there is no way to predict, from the observed behavior of a finite number of test cases, how the system will behave in any other case. If the system is non-deterministic (as many systems are), the system will not even necessarily repeat its observed behavior on a test case. Thus, in current software engineering practice, predicting that a software system will run according to specification is based almost entirely on subjective, human judgment rather than on objective, scientific fact.

In contrast to software engineering, mathematical logic provides a sound, objective way to make accurate, precise predictions about the behavior of mathematical operations. For example, if **x** and **y** are natural numbers, who among us would doubt the prediction that **x+y** always gives exactly the same result as **y+x**? This prediction is accurate not just for *some* cases, or even just for *most* cases. It is accurate for *every* pair of natural numbers, no matter what they are. The prediction is accurate because there is a *proof* that **x+y=y+x** logically follows from accepted definitions of "natural number", "=" and "+."

The Gypsy verification environment is a large, interactive computer program that supports the construction of formal, mathematical proofs about the behavior of software systems. These proofs make it possible to predict the behavior of a software system with the same degree of precision and accuracy that is possible for mathematical operations. These proofs can be constructed *before* a software system can be run; and therefore, they can provide an objective, scientific basis for making predictions about system behavior throughout the software life cycle. This makes it possible for the proofs actually to guide the construction of the system. In theory, these proof methods make possible a new approach to software engineering that can produce systems whose predictability far exceeds that which can be attained with conventional methods.

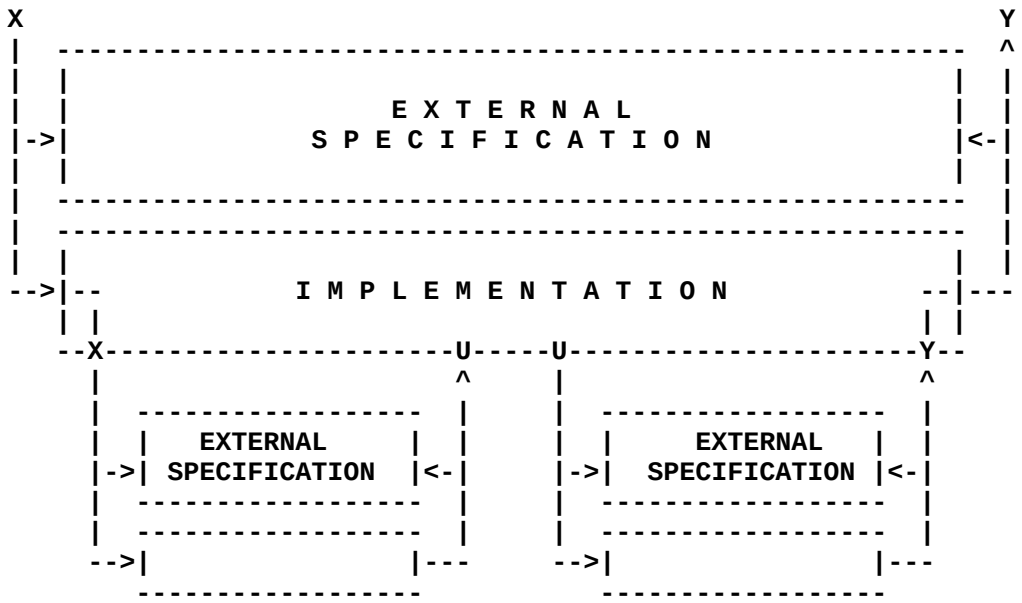
In practice, using this mathematical approach to software engineering requires very careful management of large amounts of detailed information. The Gypsy environment is an experimental system that has been developed to explore the viability of applying these methods in actual practice. The purposes of the environment are to amplify the ability of the human software engineer to manage these details and to reduce the probability of human error. The environment, therefore, contains tools for supporting the normal software development process as well as tools for constructing formal proofs.

2. A Mathematical Approach

The Gypsy verification environment is based on the Gypsy language [Good 78]. Rather than being based on an extension of the hardware architecture of some particular computer, the Gypsy language is based on rigorous, mathematical foundations for specifying and implementing computer programs. The specification describes *what* effect is desired when the program runs, and the implementation defines *how* the effect is caused. The mathematical foundation provided by the Gypsy language makes it possible to construct rigorous proofs about both the specifications and the implementations of software systems. The language, which is modeled after Pascal [Jensen 74], also is designed so that the implementations of programs can be compiled and executed on a computer with a conventional von Neumann architecture.

The basic structure of a Gypsy software system is shown in Figure 1. The purpose of a software system is to cause some effect on its external environment. The external environment of a Gypsy software system consists of data objects (and exception conditions). Every Gypsy data object has a name and a value. The implementation of a program causes an effect by changing the values of the data objects in its external environment (or by signalling a condition). To accomplish its effect, an implementation may create and use internal (local) data objects (and conditions). In Figure 1, **X** and **Y** represent external objects, and **U** represents an internal object.

Figure 1: Gypsy Software System Structure



The specifications of a program define constraints on its implementation. In parallel with the structure of implementations, Gypsy provides a means of stating both internal and external specifications. The external specifications constrain the externally visible effects of an implementation. Internal specifications constrain its internal behavior.

The external specifications of a program consist of two parts, a (mandatory) environment specification and an (optional) operational specification. The environment specification describes *all* of the external data objects that are accessible to the procedure. The specification also states the type of each of these objects and whether it is a variable or a constant object. A program may change the value of a data object *only* if it is a variable object.

The type of an object specifies the kind of values it may have. The mathematical foundations of Gypsy begin with its types. The Gypsy types are all well known mathematical objects (integers, rational numbers, the boolean values **`true`** and **`false`**, sets, sequences, mappings) or they can be easily derived from such objects (types character, record, array, buffer). For example, in Gypsy, type **`integer`** represents the full, unbounded set of mathematical objects. It is not restricted only to the integers that can be represented on a particular machine. For each of these pre-defined types, the Gypsy language also provides a set of primitive, pre-defined functions with known (and provable) mathematical properties.

The operational specification for an implementation is a relation (a boolean-valued function) that describes what effect is to be caused on the objects of the external environment. These relations are defined by ordinary functional composition from the Gypsy pre-defined functions.

The implementation of a Gypsy program is defined by a procedure. Running a Gypsy procedure is what actually causes an effect to be produced in its external environment. For implementation, the Gypsy language provides a set of pre-defined procedures (assign a value to an object, send a value to a buffer, remove an object from a sequence, ...) that have precisely defined effects. It also provides a set of composition rules (**`if...then...else...end`**, **`loop...end`**, **`cobegin...end`**,...) for composing these pre-defined procedures into more complex ones. Thus, the implementation of every Gypsy software system is some composition of the pre-defined procedures.

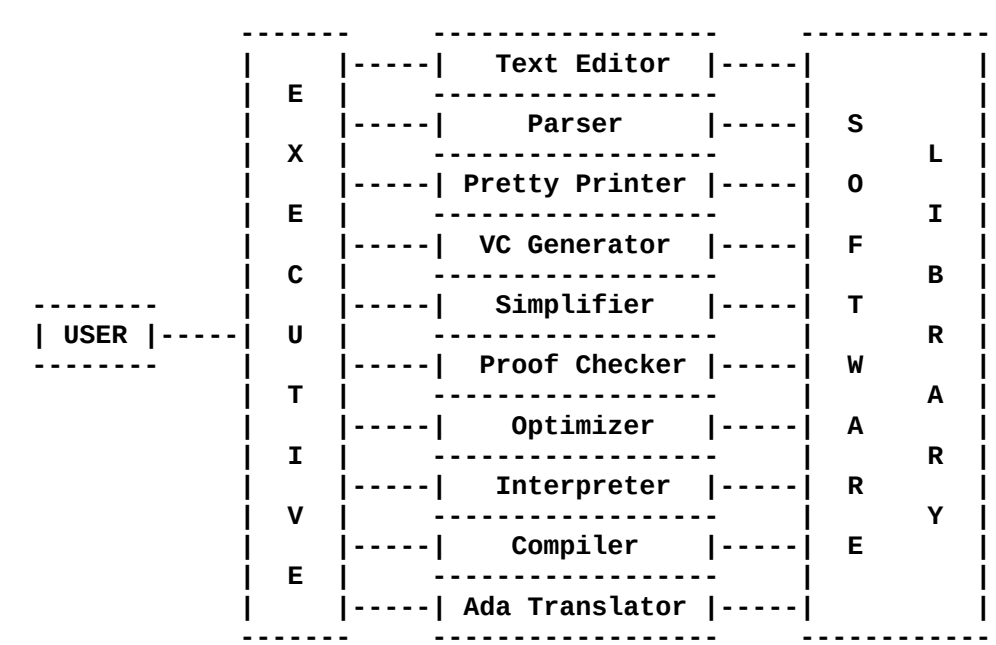
These composition rules are designed so that the effect that is caused by the composition can be deduced from the effects caused by its components. In particular, it is always possible to construct a set of formulas in the first order predicate calculus which are sufficient (but not always necessary) to show that the effect caused by a procedure satisfies its specifications. These formulas are called *verification conditions*. They are the logical conditions which are sufficient to verify that the implementation meets its specifications. By constructing them, the task of proving that an implementation always causes an effect that satisfies its specifications is reduced to a task of proving a set of formulas in the first order predicate calculus. The methods for constructing the verification conditions are based on the pioneering work of [Naur 66, Floyd 67, Dijkstra 68, Hoare 69, King 69, Good 70]. [Dijkstra 76, Jones 80, Gries 81, Hoare 82] provide more recent discussions of these basic ideas and their relation to software development.

One of the most important aspects of the Gypsy composition rules is illustrated in Figure 1. Only the external specifications of the components are required to construct the verification conditions for the composition. Neither the internal specifications nor the implementation of the components are required. The proof of the composition is completely independent of the internal operation of the components. Therefore, the proof of the composition can be done *before* the components are proved or even implemented. All that is required is that the components have external specifications. Because of this characteristic of the proof methods, a software system can be specified, implemented and proved by starting at the top and working downward rather than by building upward from the Gypsy pre-defined functions and procedures. Thus, when working from the top down, the proofs provide an sound, scientific basis for predicting how the system will behave, even long before it can be run. It is in these high levels of system design where proofs often can be most effective.

3. The Gypsy Environment

The Gypsy verification environment is an interactive program that supports a software engineer in specifying, implementing and proving Gypsy software systems. The specific goals of the environment are to increase the productivity of the software engineer and to reduce the probability of human error. To meet these goals, the Gypsy environment provides an integrated collection of conventional software development tools along with special tools for constructing formal, mathematical proofs. Figure 2 shows the logical structure of the environment.

Figure 2: Gypsy Environment Components



A single user interacts with the executive component of the environment to use a number of different software tools to build and evolve a software library. This library contains the various Gypsy components of the specification and implementation of a software system, as well as other supporting information such as verification conditions and proofs. The executive notes the changes that are made as the library evolves and marks components that need to be reconsidered in order to preserve the validity of the proofs [Moriconi 77].

The Emacs text editor [Stallman 80], parser and pretty printer are conventional tools for creating and modifying Gypsy text. The parser transforms Gypsy text into an internal form for storage in the library. The pretty printer transforms the internal form back into parsable Gypsy text. The interpreter, compiler [SmithL 80] and Ada translator [Akers 83] also are fairly conventional tools for running Gypsy programs. Although the interpreter would be a very useful debugging tool, it is not well developed and it is not presently available.

The tools that are involved in constructing proofs are the verification condition generator, the algebraic simplifier, the interactive proof checker and the optimizer. The verification condition generator automatically constructs verification conditions from the Gypsy text of a program. The algebraic simplifier automatically applies an *ad hoc* set of rewrite rules that reduce the complexity of the verification conditions and other logical formulas produced within the Gypsy environment. These rewrite

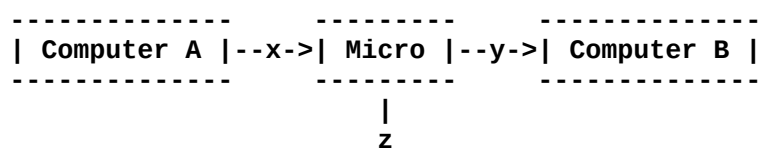
rules are based on equality (and other) relations that are implied by the definitions of the Gypsy pre-defined functions. The interactive proof checker has evolved from one described by [Bledsoe 74]. It provides a set of truth preserving transformations that can be performed on first order predicate calculus formulas. These transformations are selected interactively.

The optimizer [McHugh 83] is unique to the Gypsy environment. It produces logical formulas whose truth is sufficient to show that certain program optimizations are valid. The optimizer works in a manner similar to the verification condition generator. From the implementation of a program *and* its specifications, logical formulas called *optimization conditions* are constructed automatically. These conditions are proved, and then the compiler uses this knowledge to make various optimizations.

4. An Example

To illustrate the capabilities of the Gypsy language and environment, consider the design of a simple software system that filters a stream of messages. Two computers, A and B, are to be coupled by a transmission line so that A can send messages to B. These messages are strings of ASCII characters arranged in a certain format. However, certain kinds of these messages, even when properly formatted, cause machine B to crash. To solve this problem, a separate micro computer is to be installed between A and B as shown in Figure 3. The micro is to monitor the flow of messages from A to B, remove the undesirable messages and log them on an audit trail.

Figure 3: Micro Filter



4.1 Top Level Specification

The micro filter will be developed from the top down. The process begins by defining an abstract specification of its desired behavior. The Gypsy text for this top level specification is shown in Figure 4. When using the Gypsy environment, the first step would be to create this text and store it in the software library.

The Gypsy text defines a scope called **`message_stream_separator`** that contains six Gypsy units, procedure **`separator`**, functions **`msg_stream`** and **`separated`** and types **`a_char_seq`**, **`a_msg`** and **`a_msg_seq`**. (A Gypsy scope is just a name that identifies a particular collection of Gypsy units. The Gypsy units are procedures, functions, constants, lemmas and types. All Gypsy programs are implemented and specified in terms of these five kinds of units.)

Procedure **`separator`** is the program that will filter the messages going from computer A to B. The external environment specification of **`separator`** is **`(x:a_char_seq; var y, z:a_char_seq)`**. It states that **`separator`** has access to exactly three external data objects, **`x`**, **`y`** and **`z`** as illustrated in Figure 3. The object **`x`** is a constant, and **`y`** and **`z`** are variables. Each of the objects has a value that is a sequence of ASCII characters.

The operational specification is **`exit separated(msg_stream(x),y,z)`**. This defines a relation

Figure 4: Micro Filter Top Level Specification

```
scope message_stream_separator =
begin

  procedure separator(x:a_char_seq; var y, z:a_char_seq) =
  begin
  exit separated(msg_stream(x), y, z);
    pending;
  end;

  function msg_stream(x:a_char_seq):a_msg_seq = pending;

  function separated(s:a_msg_seq; y, z:a_char_seq):boolean =
  pending;

  type a_char_seq = sequence of character;
  type a_msg = a_char_seq;
  type a_msg_seq = sequence of a_msg;

end;
```

among **x**, **y**, and **z** that must be satisfied whenever **separator** halts (exits). The messages that arrive from computer A are supposed to be in a given format. However, there is no way to force A to deliver them properly, and, even if it does, there is the possibility of noise on the transmission line. Therefore, **separator** must designed to extract properly formatted messages from an arbitrary sequence of characters. **Msg_stream(x)** is the function that applies the formatting rules and determines the sequence of properly formatted messages that are contained in an arbitrary sequence of characters. **Separated(s,y,z)** defines what it means for a sequence of messages **s** to be separated into two character strings **y** and **z**.

This top level specification does not give precise definitions for **msg_stream** and **separated**. Only environment specifications for them are given. (The environment specifications for functions are interpreted in the same way as for procedures except that the additional type name immediately preceding the "=" identifies the type of value produced by the function.) The precise definitions of **msg_stream** and **separated**, as well as the implementation of **separator**, are left **pending** at this stage of development. At this stage, the interface between **separator** and its external environment has been defined, and it has been acknowledged that **separator** must be prepared to deal with an input sequence that may contain improperly formed messages. Formulating precise definitions for the pending items will be deferred to a later stage.

4.2 Specification Refinement

The next stage is to refine the operational specifications of **separator**. Figure 5 shows the actual Gypsy text that would be entered into the software library. This text extends scope **message_stream_separator** by replacing the old version of **separated** by the new one and by defining some new functions, types and lemmas.

Figure 5: Micro Filter Specification Refinement

```
$extending
scope message_stream_separator =
begin

  function separated(s:a_msg_seq; y, z:a_char_seq):boolean =
  begin
  exit [assume result iff y = passed(s) & z = rejected(s)];
  end;

  function passed(s:a_msg_seq):a_char_seq =
  begin
  exit [assume result =
     if s = null(a_msg_seq) then null(a_char_seq)
        else passed(nonlast(s)) @ image(last(s)).pass fi];
  end;

  function rejected(s:a_msg_seq):a_char_seq =
  begin
  exit [assume result =
     if s = null(a_msg_seq) then null(a_char_seq)
        else rejected(nonlast(s)) @ image(last(s)).reject fi];
  end;

  function image(m:a_msg):an_image = pending;

  type an_image = record(pass, reject:a_char_seq);

  lemma null_separation =
  separated(null(a_msg_seq), null(a_char_seq),
                      null(a_char_seq));

  lemma extend_separation(s:a_msg_seq; m:a_msg;
          y, z:a_char_seq) =
     separated(s, y, z)
     -> separated(s @ [seq: m], y @ image(m).pass,
                           z @ image(m).reject);

  lemma null_stream =
  msg_stream(null(a_char_seq)) = null(a_msg_seq);

end;
```

In this refinement, the **`separated`** specification is given a precise definition in terms of two new functions **`passed`** and **`rejected`**. The definition is given by the operational specification of **`separated`**. **`Result`** is the Gypsy convention for the name of the value returned by a function, and the specification states that **`result`** is to be true if and only if **`y=passed(s)`** and **`z=rejected(s)`**. The keyword **`assume`** indicates that this specification is to be assumed without proof. This is the normal Gypsy style for defining a function that is to be used just for specification.

Functions **passed** and **rejected** are defined in terms of pre-defined Gypsy functions and the function **image**. **Last** is a pre-defined function that gives the last element of a non-empty sequence, and **nonlast** gives all the other elements. The operator "@" denotes a pre-defined function that appends two sequences.

Image is a function that takes a message and produces a record of two parts, **pass** and **reject**. At a subsequent development stage, the definition of **image** will be refined to include the criterion for identifying a message that causes computer B to crash. **Image** also will define the actual output that is sent to computer B *and* to the audit trail for *each* message. If the message is of the form that will cause B to crash, the **pass** part of the record will contain a null sequence of characters and the **reject** part will contain the offending message and any other appropriate information. This record form for the result of **image** was chosen so that messages that are forwarded to B also can be audited if desired. This can be done by sending characters to both the **pass** and **reject** parts of the record. This design choice retains a large amount of flexibility for the subsequent design of the audit trail. The function **passed** applies the **image** function to each successive message **m** and appends the **pass** part of **image(m)** to **y**. Similarly, **rejected** applies **image** to each **m** and appends the **reject** part to **z**.

4.3 Specification Proof

The Gypsy text for the specification refinement also contains three lemmas. These are properties that can be proved to follow from the preceding definitions. These lemmas are the beginning of a simple problem domain theory of separating messages. The lemmas (theorems) of this theory serve several important purposes. First, to the extent that they are properties that the software designer intuitively believes *should* follow from the assumed definitions, proving that they *do* follow provides confidence in these assumptions. Second, these properties are the basis for the implementation in the next stage. They are used in the proof of the implementation to decompose the proof into manageable parts. Third, to the extent that the lemmas in this theory are reusable, they can significantly reduce the cost of other proofs that are based on the same theory [Good 82a].

The **null_separation** lemma is a rather trivial one that states that if a sequence of messages **s** is empty, then **separated(s,y,z)** is satisfied if **y** and **z** also are empty. Lemma **extend_separation** describes how to extend the **separated** relation to cover one more message **m**. If **separated(s,y,z)** is satisfied, then so is **separated(s@[seq:m], y@image(m).pass, z@image(m).reject)**.

A formal proof of both of these lemmas can be constructed with the assistance of the interactive proof checker in the Gypsy verification environment. The proof checker provides a fixed set of truth preserving transformations that can be performed on a logical formula. Although the proof checker has some very limited capability to make transformations without user direction, the primary means of constructing a proof is for the user to select each transformation. Expanding the definition of a function is one kind of transformation that can be made. The user directs the proof checker to expand the definition of a particular function, and then the expansion is done automatically. Other examples of transformations provided by the proof checker are instantiating a quantified variable, substituting equals for equal and using a particular lemma. A formula is proved to be a theorem by finding a sequence of transformations that transform the formula into **true**. This sequence constitutes a formal, mathematical proof.

A complete transcript of the interactive proof of **extend_separation** is given in Appendix A. The key steps in the proof are to expand the definition of the **separated** relation and the **passed** and **rejected** functions with the expand command. The theorem command shows the state of the formula at various intermediate stages of transformation. The **null_separation** lemma is proved in a similar

way.

Notice that both of these lemmas about message separation can be proved at this rather high level of abstraction without detailed knowledge of the specific format for incoming messages and without knowing the specific formatting details for the outputs **y** and **z**. These details are encapsulated in the functions **msg_stream** and **image** respectively. These definitions (which would need to be provided in subsequent refinement stages) might be quite simple or very complex. In either case, however, detailed definitions of these functions are *not* required at this stage. The use of abstraction in this way is what makes it possible to construct concise, intellectually manageable formal proofs about large complex specifications. The next section illustrates how similar techniques can be used in proofs about an implementation.

Finally, it is noted that the **null_stream** lemma can not be proved at this stage of refinement. However, it is required in the subsequent implementation proof, and therefore, it serves as a constraint on the refinement of the definition of **msg_stream**.

4.4 Implementation Refinement

An implementation of procedure **separator** that satisfies the preceding specifications is shown in Figure 6. The implementation contains two internal variable objects **m** and **p** of types **a_msg** and **integer** respectively. **Separator** causes its effect on its external variable objects, **y** and **z**, first by assigning each of them the value of the empty sequence of characters. Then, it enters a loop that separates the messages in **x** one by one, and for each message the appropriate output is appended to **y** and **z**.

The desired effect of the loop is described by the **assert** statement. It states that on each iteration of the loop, messages in the subsequence **x[1..p]** have been separated. (The Gypsy notation for element **i** of sequence **x** is **x[i]**, and **x[1..p]** is the notation for the subsequence **x[1],...,x[p]**.) This assertion is an *internal* specification about the operation of the procedure.

The loop operates by successively calling the procedures **get_msg** and **put_msg**. **Get_msg** assigns to **m** the next properly formatted message in **x** and increases **p** to be the number of the last character in **x** that has been examined. **Put_msg** appends to **y** and **z** the appropriate output for the new message **m**. These properties of **get_msg** and **put_msg** are stated precisely in the specifications that are given for them in Figure 6. (For the variable **p**, **p'** refers to its value at the time **get_msg** is started running, and **p** refers to its value when the procedure halts. The operator **<:** appends a single element to the end of a sequence.)

4.5 Implementation Proof

The remaining task for this level of the design of the micro filter is to prove that this abstract implementation of **separator** satisfies its specifications (both internal and external). This proof is possible without any further refinement of the specifications or the implementation. The current form is an instance of the one shown in Figure 1. Specifications and an implementation for **separator** have been constructed, but there is no implementation of either **get_msg** or **put_msg**. This level of proof simply assumes that these procedures eventually will be implemented and proved to satisfy their specifications. However, at this level, only their external specifications are required.

It is easy to see that the **exit** specification of **separator** logically follows from the assert statement in the loop whenever the procedure leaves the loop. This follows simply from the facts that, when the loop halts, **p=size(x)** and that for every Gypsy sequence, **x[1..size(x)]=x**. It also is easy to see that

Figure 6: Micro Filter Implementation Refinement

```
$extending
scope message_stream_separator =
begin

procedure separator(x:a_char_seq; var y, z:a_char_seq) =
begin
exit separated(msg_stream(x), y, z);
  var m:a_msg;
  var p:integer := 0;
  y := null(a_char_seq);
  z := null(a_char_seq);
  loop assert separated(msg_stream(x[1..p]), y, z)
       & p le size(x);
    if p = size(x) then leave;
    else get_msg(x, m, p);
         put_msg(m, y, z);
    end;
  end;
end;

procedure get_msg(x:a_char_seq; var m:a_msg; var p:integer)
begin
exit msg_stream(x[1..p]) = msg_stream(x[1..p']) <: m
     & p > p' & p le size(x);
  pending
end;

procedure put_msg(m:a_msg; var y, z:a_char_seq) =
begin
exit y = y' @ image(m).pass & z = z' @ image(m).reject;
  pending
 end;

end;
```

the assert statement is true the first time the loop is entered. This is because the local variable **p** is zero, and **y** and **z** are both equal to the empty sequence. The assertion then follows from the **null_stream** and **null_separation** lemmas because in Gypsy **x[1..0]** is the empty sequence and the size of a sequence is always non-negative. Finally, the **extend_separation** lemma can be used to prove that if the loop assertion is true on one iteration of the loop, then it also is true on the next. These steps comprise an inductive proof that the loop assertion is true on every iteration of the loop (even if it never halts). The loop, however, does halt because, according to the specifications of **get_msg**, **p** is an integer that increases on each iteration and yet never increases beyond the number of characters in the constant **x**. Therefore, the loop must halt; and when it does, the **exit** specification follows from the loop assertion.

The Gypsy verification environment automates all of this argument (except the argument about the loop

halting). From the Gypsy text shown in Figure 6, the verification conditions generator automatically constructs the formulas shown in Figure 7.

Figure 7: Separator Verification Conditions

```
Verification condition separator#2
separated (msg_stream (null (#seqtype#)),
           null (a_char_seq), null (a_char_seq))

Verification condition separator#3
  H1: msg_stream (x[1..p]) @ [seq: m#1]
        = msg_stream (x[1..p#1])
  H2: y @ image (m#1).pass = y#1
  H3: z @ image (m#1).reject = z#1
  H4: separated (msg_stream (x[1..p]), y, z)
  H5: p le size (x)
  H6: p + 1 le p#1
  H7: p#1 le size (x)
  H8: size (x) ne p
 -->
  C1: separated (msg_stream (x[1..p#1]), y#1, z#1)
```

Verification condition **separator#2** is the formula that states that the loop assertion is true the first time the loop is entered. **Separator#3** is the one that states that if the assertion is true on one iteration of the loop, it also is true on the next. Lines labelled **Hi** are the hypotheses of an implication, and lines labelled **Ci** are conclusions. Both the hypotheses and the conclusions are connected implicitly by logical conjunction. The notation **m#1** denotes a value of **m** upon completing the next cycle of the loop, and similarly for **p, y** and **z**. The notation **[seq: m#1]** means the sequence consisting of the single element **m#1**. The verification condition generator also has constructed a **separator#4** for the case when the loop terminates. The generator, however, does not present this one because the formula has been proved automatically by the algebraic simplifier. The best way to see the effect of the simplifier is to see what the verification conditions look like without it. The unsimplified formulas are shown in Figure 8. (There also is a **separator#1** which is so trivial that the generator does not even bother to use the algebraic simplifier.)

A complete transcript of the interactive proof of **separator#3** is given in Appendix A. The key steps are to do equality substitutions based on hypotheses H1, H2 and H3 with the eqsub command and then use the **extend_separation** lemma. **Separator#2** is proved by use of the lemmas **null_stream** and **null_separation**.

Once **separator** has been proved, the process of refinement can be resumed. In general, the refinement of both specifications and implementations is repeated until all specifications and procedures are implemented in terms of Gypsy primitives.

It is important to observe that the proof of **separator** has identified formal specifications for **get_msg** and **put_msg** that are adequate for the subsequent refinements of these procedures. It has been proved that **separator** will run according its specification if **get_msg** and **put_msg** run according to theirs. Therefore, these specifications are completely adequate constraints for the subsequent refinements. Some

Figure 8: Unsimplified Verification Conditions

```
------------------------------------------------------------------
|                                                                |
|   Verification condition separator#2                           |
|     H1: true                                                   |
|    -->                                                         |
|     C1: separated (msg_stream (x[1..0]), null (a_char_seq),    |
|                    null (a_char_seq))                          |
|     C2: 0 le size (x)                                          |
|                                                                |
|   Verification condition separator#3                           |
|     H1: separated (msg_stream (x[1..p]), y, z)                 |
|         & p le size (x)                                        |
|     H2: not p = size (x)                                       |
|     H3: msg_stream (x[1..p#1]) = msg_stream (x[1..p]) <: m#1   |
|         & p#1 > p                                              |
|         & p#1 le size (x)                                      |
|     H4: y#1 = y @ image (m#1).pass                             |
|         & z#1 = z @ image (m#1).reject                         |
|    -->                                                         |
|     C1: separated (msg_stream (x[1..p#1]), y#1, z#1)           |
|     C2: p#1 le size (x)                                        |
|                                                                |
|   Verification condition separator#4                           |
|     H1: separated (msg_stream (x[1..p]), y, z)                 |
|         & p le size (x)                                        |
|     H2: p = size (x)                                           |
|    -->                                                         |
|     C1: true                                                   |
|     C2: separated (msg_stream (x), y, z)                       |
|                                                                |
------------------------------------------------------------------
```

of the specifications may not be necessary, but they are sufficient to ensure that **`separator`** will satisfy its specification.

5. Trial Applications

The Gypsy environment has been developed to explore the practicality of constructing formal proofs about software systems that are intended to be used in actual operation. Throughout its development, the environment has been tested on a number of trial applications. The two major ones are summarized below.

5.1 Message Flow Modulator

The most recent application of Gypsy is the message flow modulator [Good 82b]. The micro filter that has been specified, designed and proved in the Section 4 is a very close approximation of the modulator. The micro filter example was chosen deliberately to show how it is possible to construct concise, formal proofs about much larger software systems. The modulator consists of 556 lines of implementation, and the proofs in the preceding sections apply, with only very minor alteration, to the design of the modulator. The lower level details that are unique to the modulator are encapsulated in the **`msg_stream`** and

image functions.

The message flow modulator is a filter that is applied continuously to a stream of messages flowing from one computer system to another. As in the micro filter, messages that pass the filter are passed on to their destination with a very minor modification. Messages that do not are rejected and logged on an audit trail. A properly formatted message consists of a sequence of at most 7200 ASCII characters that are opened and closed by a specific sequence.

The filter consists of a list of patterns. Each pattern defines a sequence of letters and digits that may be interspersed with various arrangements of delimiters. A delimiter is any character other than a letter or digit. If a message contains any phrase that matches any pattern, it is rejected to the audit trail along with a description of the offending pattern. Messages that do not contain any occurrence of any pattern are forwarded on to their destination.

In essence, the formal specifications of the modulator have the form **y=f(x,r) & z=g(x,r)** where **r** is the list of rejection patterns. The specification describes the exact sequences of characters that must flow out of the modulator for every possible input sequence. This includes handling both properly and improperly formatted messages in the input stream, detecting phrases that match the rejection patterns, and formatting both output sequences. The Gypsy formulation of these specifications is described in further detail in [Good 82b].

The modulator was developed within the Gypsy environment as a converging sequence of prototypes. First, Gypsy specifications and proofs were constructed for the top levels of the modulator design. This design covered the basic separation of messages into the two output streams. Then, a sequence of running prototypes was implemented. The purpose of these prototypes was to help decide what some of the detailed behavior of the modulator *should* be. These prototypes were used to investigate various approaches to handling improperly formed messages and to formatting the audit trail. Specifications for these aspects of the modulator were decided upon only after considerable experimentation with the prototypes. Next, another sequence of performance prototypes was built to evaluate the performance of various pattern matching implementations. Once adequate performance was attained, the Gypsy specifications and proofs were completed for the entire modulator.

As the final step, the proved modulator was tested in a live, operational environment on test scenarios developed by an independent, external group. Without any modification, the proved modulator passed all of these tests on the first attempt.

5.2 Network Interface

The first major application of Gypsy, and the most complex one to date, was a special interface for the ARPANET. Each ARPANET host has message traffic which needs to be transported over the network according to the standard Transmission Control Protocol (Version 4.0). The ARPANET, however, is assumed to be an untrustworthy courier. The special interfaces are to ensure proper message delivery across this potentially unreliable network.

Normally, each host is connected directly to the network by a bi-directional cable. Each cable is cut and an interface unit is installed at the cut (Figure 9). This turns the "dumb" cable into a "smart" one. When the smart cable receives a message from the host, the message is checked to see that it is return-addressed to the sending host. If it is not, the message is dropped. If it is properly return-addressed, then, in effect, the smart cable seals the message in a plain brown envelope that can not be opened by the network, addresses and return-addresses the envelope and sends it to the ARPANET for delivery. In the other