Proving Gypsy Programs

Richard M. Cohen

12 May 1987

Technical Report #51 May 1986

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

Abstract

Program verification applies formal understanding of programming language semantics to the practical problem of constructing reliable software. The programming language Gypsy and the Gypsy Verification Environment, a programming environment that supports development of programs, specifications, and proofs, represent significant steps in making the program verification technology available for practical use.

The purpose of this dissertation is to specify clearly the meaning of Gypsy programs, and the means by which they are proven. The semantics are presented in a "semi-formal" way, so that they are more accessible to the verification practitioner, than would be more formal, mathematical semantics. First an operational semantics for a subset of Gypsy is presented. Then this model is extended to cover more complex features of the language. Using the execution semantics as the underlying conceptual base, we proceed to present the mechanisms used to generate verification conditions for the full Gypsy language, including data abstraction, concurrent programming, and exception handling.

Acknowledgments

I owe many debts to my friends and colleagues for their support and encouragement throughout this work. None of this work would have been done without the interest and enthusiasm of Don Good, the primary force behind the creation of Gypsy. Bob Boyer was instrumental in helping me avoid several blind alleys, and convincing me I was done. Dan Craigen read an early draft of my dissertation, and his comments helped me improve upon it.

My past and present colleagues at the Institute for Computing Science have contributed to my efforts in many ways. They have provided a good environment for discussion, elaboration, and refinement of my understanding of Gypsy. They have also provided friendship and intellectual interest that has helped me through these efforts.

Many friends have encouraged me throughout my graduate career, and without them I surely would not have persevered. For this I acknowledge special gratitude to my dear friends Dave and Paula Matuszek, John McHugh, Ruth Baldwin, Paul Reynolds, and Bernice Borak. The members of C.E.R.F. offered encouragement, and I trust they are satisfied with the result. Many others have been helpful, and I do not mean to slight them by not mentioning their names. It only indicates that I have a poor memory.

And lastly, I must acknowledge the support of my wife, Dorene. She not only put up with me when I was tired, depressed, excited, and frequently absent, but remained supportive and enthusiastic. We have supported each other through our studies financially, emotionally, and spiritually. Neither of us would have finished without the other’s support.

Chapter 1
INTRODUCTION

Gypsy is a program description language. That is, Gypsy is a language for describing computer programs and their specifications. This report defines the proof methods for the language Gypsy. A simple semantic model of a language feature is first presented and then extended as required to encompass the additional complexities of the full language. A verification condition generator based on this model is then developed. The elaboration is pedagogical and is meant to provide an understanding of the meaning of Gypsy programs. The goal of this report is to provide the reader with this understanding.

The presentation describes the unique proof methods for handling concurrency and data abstraction in Gypsy. Gypsy supports concurrent programming using message buffers and supports specification of concurrent programs by defining buffer histories, records of the messages sent to and received from a buffer.

The reader is expected to be familiar with the Gypsy language, as described by the Gypsy 2.1 language report [Good 85].[1] Footnotes generally discuss details that relate the current topic to topics that are discussed in later sections. They may be skipped on first reading.

1.1 Why Gypsy is Important and What I Have Done

I have defined the semantics of the Gypsy language and the formal methods used to prove the consistency of Gypsy programs and specifications. Gypsy and the Gypsy Verification Environment (GVE) mark a significant step in the development of practical methods for program verification. The development of these proof methods form the underlying basis for Gypsy verification. Moreover, the definition style used here (mapping an operational definition into a verification condition generator) offers an intuitive basis for the verification by clearly relating an operational understanding of the program to the verification conditions (VCs) required for the proof. This reinforces in the programmer's mind the relation between a procedural understanding of the program (as is common to most programmers) and the requirements of the proof methods encountered during the verification.

Gypsy proof methods include two areas often left out of formal semantic definitions:

- concurrency, and
- exception handling.

The proof methods developed for concurrent programs in Gypsy were the first formalization of the semantics of message buffers. Earlier descriptions and examples of the Gypsy concurrency have already been published in

[1]The reader who is familiar with the Gypsy 2.0 language report [Good 78a] should not find the differences particularly bothersome.

[Good 78b] and [Good 79].

1.2 Semantic Definition Method

The programming language Gypsy and informal proof methods for Gypsy programs have evolved over several years. An important goal in this evolutionary design has been modularity of programs and their proofs. Modularity in programs is traditionally enforced by abstraction and information hiding. An acceptable semantic definition of Gypsy must reflect those mechanisms into the proof methods.

Normally denotational semantic definitions embody no abstraction. The denotation of a function declaration is the mathematical function it computes. References to the function from other points in the program evoke that denotation and hence, complete knowledge of its definition. This violates the requirement of information hiding.

To solve this we take a two-step approach, presenting:

1. an execution model which lacks abstraction, and then
2. methods (derived from the execution model) to generate and prove the verification conditions (VCs) derived from the program. The VCs and their proofs will be constructed to permit proof to reflect abstraction in the program.

The primitive execution model is presented mainly to provide an intuitive base on which to develop the VC generator. The simple execution model is unacceptable in our proofs, as it violates our requirement of modularity. The VC generator supports our notions of modular and abstract proofs, but masks the full semantic interpretation of program units by reducing the usable knowledge of a routine to its (possibly incomplete) external specifications.

In practice, the formal specifications of verified software often specify less than full functionality, stating only the critical program properties. Thus, programs that meet their formal specifications may not exhibit other required (but not formally specified) behavior. (E.g., real-time behavior is ignored in most formal specification languages.)

1.3 Nondeterminism

Nondeterminism arises for many reasons and is necessary for most distributed computing systems. This nondeterminism causes significant complications in programming. Thus, program verification is of particular interest for distributed or parallel systems, as they are much harder to understand than sequential programs. This same nondeterminism also complicates programming language semantics, causes significant complications in program verification, and interferes with the use of most mechanical theorem provers. Most theorem proving work is done in theories with deterministic functions. If F is a nondeterministic function, then $F(X) = F(X)$ may not be a valid formula.

Our semantics allow us to manage nondeterministic program constructs. Basically, the extensions add contextual knowledge about otherwise uninterpreted function symbols. This allows us to reason about the function symbols without revealing complete details. Use of the "context" mechanism is sufficient to provide a weak definition of concurrent Gypsy constructs. We use the abstraction facility to hide many of the nondeterministic details of how the concurrent processes interact. The Gypsy specification language does not allow us to describe the full behavior of Gypsy concurrent processes. Certain aspects of concurrent program behavior that are evident in the intuitive operational model cannot be expressed in the specification language.

The semantics of concurrency given here are incomplete in exactly the same way. The modularity and abstraction that form the basis of the *Independence Principle* [Good 85, p. 2] are enforced equally in the programming language and in the specification language.

Gypsy buffers resemble shared variables. There is no mechanism to pass a buffer "exclusively" to a procedure, indicating that no other active procedure has access to that buffer. Therefore, the independence principle forces us to treat the buffer parameter as a shared buffer, which other procedures may manipulate while this procedure is active. With sufficient global knowledge of the calling structure program, it might be possible to determine that no other procedures could in fact access the buffer while this procedure was active; such knowledge cannot be used in a proof of the Gypsy procedure, as it would violate the independence principle.

Gypsy procedures may be nondeterministic. For example, in a concurrent procedure call, the result may depend on the order in which events occur in the parallel procedure invocations. The semantics of the concurrent procedure call do not specify the order of the events. Any ordering information regarding the events must be gained from the specification abstraction of the called procedures.

1.4 Incremental Development of Programs and Proofs

Programs are usually developed by iterative refinement and rewriting. The direction may be top-down or bottom-up, but some backtracking is almost always necessary. Later, programs are often revised and extended (an activity called "maintenance"). Modifying a program may require recompilation. It may also require modifications to the proof. Constructing formal specifications and proofs is very time consuming and labor intensive. Redoing the proof of an entire program would be extremely expensive, and if verification is to be practical, small modifications in a program must require only small modifications in the proof. The same notions of abstraction that allow us to modify programs easily also allow us to construct modular proofs. The portion of a proof that depends on a particular piece of the program should be comparable to the portion of the program that depends on that piece. The VC generation semantics allow us to reflect the abstraction mechanisms built into the language into the proof methods, and hence, to support modularity in proofs just as the language supports modularity in the program. This allows the GVE’s proof manager [Moriconi 77] to handle incremental changes during program development.

1.5 Negotiable Semantics

The language definer has a choice between defining rigorous, inflexible semantics, leaving no aspect of program behavior undefined, or defining looser, flexible semantics, leaving some aspects of program behavior unspecified. (The appearance of the word "undefined" in language definitions usually indicates presence of the latter approach.) Some have argued as to whether the presence of "undefined" in a language definition is good or bad [Palme 74]. One main reason for introducing these unspecified portions into the language definition is to provide some leeway for the implementor. Some details of a language feature may be irrelevant or tangential to the intended behavior of a program. Leaving these details of the language unspecified may allow the use of a variety of implementation techniques. Conversely, specifying every detail of the language may preclude some desirable implementation options.

Some semantic definition techniques lend themselves to leaving some things undefined, while others do not. Often the semantic definitional methods make it easier or cleaner for the language definer to completely specify program behavior. In the semantic definition of Gypsy, the order of argument evaluation is specified, and call-by-value/call-by-value-result semantics are used to describe program behavior.[2] This admits several reasonable operational models, which are simpler than ones involving call-by-reference. However, this also has the effect of requiring Gypsy compilers to preserve these semantics even when attempting to use the more efficient call-by-reference technique at run time. We propose a language extension that allows the programmer to "request" relaxation of some details of the language definition. The formal semantics do not reflect any leniency in the formally required program behavior, but the proof methods do recognize this relaxation request and relax the semantics used to verify that the program conforms to its specifications. Thus, the programmer can explicitly annotate the program to indicate that some details for the program behavior are not required. The proof methods will use the relaxed program semantics in verifying the program, and an optimizing compiler might be allowed to take advantage of the relaxed semantics to apply specific optimizations.

The "negotiation" allows the program writer to specify an abstract equality relation on program states, masking certain uninteresting distinctions. Based on this, weaker proof rules can be used and a possible explosion of cases can be avoided.

This "relaxation" of semantics is particularly helpful in the case of exception conditions in Gypsy programs. Gypsy programs typically handle many exception conditions with a single "condition handler", and it is often not necessary to deal with the detailed state information that distinguishes one particular condition from another. Thus, if the programmer could specify that the values of some variables were not of interest when an exception condition is raised, then the proofs relating to the exception condition would not have to distinguish among the multiple signalling states.

1.6 Outline of Chapters

1.6.1 Introduction

The introduction provides a brief overview of the work and presentation, and tries to gently introduce the reader to the issues at hand.

1.6.2 Related Work

This chapter discusses the nature of program verification, and identifies several tutorial works on verification. Other mechanical systems supporting program verification are described. Several other efforts have addressed the areas of specifying and proving concurrent programs and exception handling. These are contrasted with the approach presented here.

1.6.3 Some Remarks about Gypsy Programs

This chapter deals with several issues underlying the Gypsy proof methods. The first section outlines the properties of the Gypsy language that helped to make the semantic definition simple. Gypsy does not distinguish function declarations intended for use in specifications from those intended for execution. Thus, Gypsy functions have meaning in program proofs, as well as a possible interpretation as an executable function. The nature of this dual definition is discussed, as well as several aspects of Gypsy specifications and their

[2]The Gypsy 2.1 Report [Good 85] defines assignment and other basic operations of the language (including function invocation) in terms of procedures and procedure calls. The approach taken here is slightly different. We use assignment and function invocation as primitive operations, and reduce the other parts of the language (including procedure invocation) to these two primitives. This different approach was chosen to reduce the number of primitives required in the semantic description, but is logically equivalent to that taken in the language report.

analysis.

1.6.4 Normalizing Gypsy Programs

Many Gypsy constructs can be eliminated by treating them as abbreviations with expansions composed from a smaller base language. In order to simplify the semantic definition, the number of constructs in the base language is reduced. The normalization process reduces Gypsy programs to six executable statements and three specification statements. The details of normalization and expanding syntactic abbreviations are described.

1.6.5 A Simple Operational Model

A simple operational model for Gypsy programs is presented. The model is first developed without considering exception handling or concurrency. The model is then extended to include exception handling. Concurrency is dealt with in Chapter 8.

1.6.6 A Verification Condition Generator

Based on the operational model in Chapter 5, a verification condition generator (VC generator) is described. A forward VC generation technique is used to provide a close relation between the operational model and the VC generation process. The VC generator first analyzes the control flow of the routine to compute a set of linear control path segments, and then symbolically executes each path segment to generate the VCs. Methods for proving termination of execution of sequential Gypsy programs are also described.

1.6.7 Buffers and Activation_ids

The basic Gypsy concurrency mechanism is based on message buffers. Specification of concurrent programs is based on histories of the messages sent to and received from buffers. The history mechanism uses a unique activation_id to identify each dynamic routine invocation, in order to pair each message in the history with the specific routine activation that performed the corresponding send or receive operation on the buffer.

1.6.8 Concurrency

The Gypsy concurrent procedure call (**`COBEGIN`** statement) is a generalization of the sequential procedure call. The basis of specifying and verifying concurrent Gypsy programs is to describe the effect of the **`COBEGIN`** on buffer parameters. The **`BLOCK`** specification is a technique for describing behavior of non-terminating concurrent programs. The interpretation of the **`BLOCK`** specification is given, along with its proof methods.

1.6.9 Proof of Data Abstraction

Data abstraction does not affect the execution semantics of the program. Data abstraction merely imposes visibility restrictions on the program description and proof. Program code and specifications dealing with the abstract data type (not its implementation) must be independent of the details of the data representation. External specifications of a routine include both abstract specifications, which are visible everywhere, and concrete specifications, which are only visible in contexts in which the data representation is known. These two sets of specifications must be proven to be consistent. Gypsy allows the programmer to define equality on abstract data types. To allow the normal rules of substitution in proofs, it is necessary to show that the user has defined an equivalence relation, and that all functions defined on the abstract data type behave deterministically under this equivalence relation.

1.6.10 Proposed Language Changes

In pursuing this work, several relatively minor changes came to light that might improve the language. The most significant of these are the suggested extensions to support proof of well-definedness of specification functions and termination of (sequential) executable routines.

1.6.11 Future Work

The work presented suggests several lines for future research, such as the application of this style of semantic definition to other languages, and more formal models of VC generation. Several other topics for further work are discussed.

1.7 Lacuna: Topics Omitted

The precise version of the language addressed here lies partly between Gypsy 2.0 and Gypsy 2.1. The language is basically Gypsy 2.1 as described in the July 1985 draft language manual [Good 85]. This work proceeded in parallel with the evolution of Gypsy 2.1 from Gypsy 2.0, and some aspects of this "moving target" may yet be evident in the manuscript. Some of the language changes incorporated in Gypsy 2.1 were based on these efforts to clarify and formalize the semantics of Gypsy.

The static semantics or compile-time type consistency requirements of Gypsy are not discussed here, nor are the semantics of Gypsy data structures. The type consistency rules are adequately explained in the Gypsy 2.0 and Gypsy 2.1 language reports. Gypsy’s data structures are quite ordinary. They include integers, rationals, arrays, and records, all of which are common in many programming languages, and quite well understood. Gypsy also includes sets, sequences (indexed lists), and "mappings" (sets of ordered pairs), which are not present in most programming languages. But these less common data types are defined to behave as in common mathematical usage, and so they, too, are not of special interest here.

Chapter 2
RELATED WORK

2.1 Specification & Verification Techniques

There are several different aspects of "program correctness."

- Partial correctness deals with proving properties of the results of programs. The results are guaranteed to satisfy the properties *if the program terminates and produces any results.*
- Proof of termination addresses the question of whether a given program will terminate. Of course, we know that proof of termination is not possible for all programs, since this is precisely the Halting Problem [Hopcroft & Ullman 69].
- Total correctness is the combination of partial correctness plus a proof of termination.
- Safety properties are the analog of partial correctness in the world of concurrent programs. These deal with properties of partial results of running programs.
- Liveness properties and the absence of deadlock are the analogs of termination in the world of concurrent programs. These deal with the progress of a concurrent computation and its ultimate production of some result or action.

Many techniques have been proposed for specifying and proving partial correctness of programs (e.g., state machines, algebraic axioms, program assertions for specifications, and inductive assertions, structural induction, computational induction for proofs). The classic technique for proving partial correctness properties of sequential, procedural code is the Floyd-Hoare inductive assertion method [Floyd 67, Hoare 69]. This forms the basis for Gypsy verification, as it does for much of the practical verification work being done.

There are several introductions to the area of formal program verification, including [Manna 74, Hantler 76, Anderson 79, Boyer&Moore 81a]. [Berg 82] is particularly encyclopedic in its coverage of specification and verification techniques. Many progressive introductory programming texts, such as [Wirth 73], introduce the notions of formal specification and proof along with the more traditional notions of algorithmic programming. Some perspectives on program verification can be obtained by reading [London 75, Gries 78]. [Gordon 79] provides a very nice introduction to denotational semantics, and [Pagan 81] and [Tennent 81] provide a general introduction to programming language semantics.

2.2 Verification Systems

Interest in the formal correctness of programs dates from the 1960's [McCarthy 63, Floyd 67, Dijkstra 68, Hoare 69]. Elspas *et al.* suggested that mechanical theorem provers were required to make program verification practical as early as 1972 [Elspas 72]. Yet Lipton *et al.* seem to have misunderstood the importance of mechanical proof checkers in 1979 [De Millo 79].

King [King 69] was the first to implement a mechanical program verifier based on Floyd's work. His verifier used backward substitution to generate VCs for a simple Algol-like language manipulating integers.

The Stanford Pascal Verifier [von Henke 75, Luckham 79] is perhaps best known for Wolfgang Polak's proof of a Pascal compiler [Polak 80]. However, the user must supply the theorem prover with a large number of axioms in order to complete the proofs, and it is extremely difficult for humans to construct large sets of axioms without introducing a contradiction.

AFFIRM [Gerhart 80, Thompson 81, Erickson 81] started as a verifier for abstract data types based on the Knuth-Bendix algorithm [Knuth 69] for testing the completeness of a system of rewrite rules. The system was extended to support VC generation for Pascal-like programs, although the proofs about abstract data types (including inductive proofs) remains AFFIRM's strength.

The Hierarchical Development Methodology (HDM) was developed at Stanford Research Institute, later SRI International [Robinson 75, Robinson 77, Robinson 79, Silverberg 79, Levitt 79]. It centered around the program specification language SPECIAL [Roubine 77], which uses a state machine model to describe changes to a global machine state (i.e., global variables), based on the work of David Parnas [Parnas 72a, Parnas 72b]. The user then proves properties of the initial machine state and invariants about reachable states. Lower level machines may be defined, and mappings between the higher level and lower level machines may be shown to preserve the verified properties of the higher level machine. Ultimately, the state machine specification is used as a specification for a program written in a standard programming language. HDM adherents claim that this "programming language independence" is an advantage. However, HDM only proves properties of a non-procedural program specification, which must then be interpreted by a human programmer in order to produce an actual program. Thus, the relevance of the specification proof to the actual program depends on the integrity of the human programmer.

A new version of HDM is under development [Crow 85a]. Revised SPECIAL is based on Hoare-axioms [Crow 85b], and the new HDM system accepts statements about fragments of Pascal programs using Hoare's notation. Larger program fragments can then be built from smaller ones by the rule of composition, until full Pascal programs have been constructed.

Ina Jo [Locasso 80] is the specification language component of the Formal Development Methodology developed at Systems Development Corporation. Ina Jo is another state-machine oriented program specification language. It is mated with a low-level interactive theorem prover called ITP [Schorre 84]. As with HDM, this system only proves properties of non-procedural state-machine program specifications. An attempt is being made to extend Ina Jo to support specification of concurrent programs using temporal logic [Nixon 85].

Starting in the early 1970's, Boyer and Moore built one of the most impressive mechanical theorem provers in existence [Boyer&Moore 75, Boyer&Moore 79]. They use structural induction to prove properties of pure LISP programs, using LISP as the specification language as well as the "programming language." They implemented a VC generator for a large subset of FORTRAN in order to produce more theorems for their prover to prove [Boyer&Moore 81b].

The Gypsy Verification Environment has evolved over a period of ten years. It's predecessor, jointly developed by USC Information Sciences Institute and The University of Texas, is described in [Good 75]. Various aspects of the GVE are described in [Moriconi 77, Hare 79, Smith 80, Akers 83]. Some of the notable successes of the Gypsy effort are described in [DiVito 81, DiVito 82, Good 82, Siebert 84, Good 84].

2.3 Proof of Concurrent Programs

Owicki and Gries [Owicki 75, Owicki 76] handle concurrent programs with globally shared variables. The only constraint is that the underlying machine provide the uninterruptable memory operations read-a-word and write-a-word. Their method first proves properties of the individual, sequential procedures, and then shows that there is no "interference" between the operation of the concurrent procedures and the assumptions of the individual proofs. Thus, the sequential proofs remain valid for the concurrent program. While this is a very powerful technique, the non-interference proofs can grow very large.

Apt *et al.* [Apt 80] developed an axiomatic proof technique for Hoare's CSP. They deal with both safety properties and freedom from deadlock. This requires proving "cooperation" of the component, sequential proofs, which corresponds to global reasoning about the cooperating processes, in addition to the sequential proofs. The proofs of cooperation can grow as $O(n^2)$ in the length of the program, as is the case with Owicki and Gries.

Levin has also developed an axiomatic proof technique for CSP [Levin 81]. He uses shared auxiliary variables, rather than global invariants as Apt *et al.* Thus, in constructing the proof of a program from those of its sequential processes, he must prove both a "satisfaction" proof, that all possible message exchanges justify the post-condition following a send or receive, and a non-interference proof (similar to Owicki and Gries). He views communication between processes as a means to an end, rather than as the purpose of the computation. Thus, it is natural to use auxiliary variables to reflect as much or as little of the communication history as needed.

Monitors [Hoare 74] provide a mechanism for structuring control of and access to shared objects. Proof methods for monitors were introduced [Howard 76] as Gypsy 1.0 was being designed. Indeed, monitors were originally envisioned as the concurrency mechanism to be included in Gypsy. However, the examples that were driving the initial concurrent verification work at that time were mainly communication and networking problems, and early examples of Gypsy programs tended to use monitors to build bounded buffers. This observation led to the adoption of message buffers as a simpler, more constrained concurrency mechanism, and ultimately led to simpler proof methods.

2.4 Exception Handling

The Gypsy mechanisms for handling exception conditions is based on the work of Goodenough [Goodenough 75]. A significant extension added in Gypsy is the mapping of condition names at routine boundaries, in order to preserve the independence principle.

A condition signalled in a routine is propagated into the calling environment as **ROUTINEERROR** if the condition signalled is neither handled within the routine nor is it a condition parameter of the routine.[3] Propagating the original condition unchanged would leave unconstrained the possible exception conditions that

[3]The one exception to this rule is that **SPACEERROR** propagates unchanged. This is because **SPACEERROR** represents violation of an implementation defined constraint on computing resources, a resource limit not directly under the control of the invoked routine.

could arise at a call site, depending on all of the routines in the possible call-tree below that point. There would be no way to analyze the possible control paths due to exceptions without global knowledge of all of the routines involved. Mapping the condition names at routine boundaries allows analysis to be done based only on the local procedure and the external specifications of called routines. Luckham's work on exception handling in Ada™ has used the Gypsy **`EXIT CASE`** style of specification in Anna. [Luckham 80]

Optimizing compilers prove theorems about programs in order to justify that optimizing transformations do not alter the semantics of a program. Typically, either the optimizers simply fail to do the proofs, or apply optimizations for which the proofs can be done very simply (perhaps due to the language design). McHugh used clean termination proofs of Gypsy statements to support optimization of Gypsy programs [McHugh 83]. Typical optimizations involving code motion and the suppression of run-time tests preserve program semantics only in the absence of run-time errors. By using a description of the run-time environment, McHugh was able to prove the absence of certain run-time errors, and hence apply program optimizations safely.

Several attempts have been made to extend algebraic specifications to describe operations that may encounter run-time error conditions. The notion of an extended algebra including error terms and undefined terms is discussed in [Majster 79] and [Musser 77].

Euclid [Lampson 77]was initially designed at the same time as Gypsy, but was intended more to support practical programming efforts than program verification research. Still, verifiability was an important goal of the Euclid work. Following the style of the axiomatic definition of Pascal done by Hoare and Wirth [Hoare 73], proof rules were developed for Euclid [London 78], and a verifier for a close descendent of Euclid [Crowe 82] is currently under construction [Bonyun 82, Craigen 84]. Euclid took an interesting approach to exception handling. No legal Euclid program can signal an exception. In order to be a legal Euclid program, various "legality assertions" must be proven, showing that run-time errors will not occur. Thus, one cannot determine whether a program is a legal Euclid program without doing the proofs! Early work on proving absense of run-time errors was done by Richard Sites [Sites 74]. Sites developed techniques for proving clean termination of flow-graph programs. Later, Steve German worked on proving absence of run-time errors in Pascal programs [German 81]. German produced a working implementation based on his formalism.

2.5 Key Results of the Gypsy Project

The Gypsy Project has addressed the breadth of program verification from language design and formal semantics, to mechanical tools for supporting proofs about real programs, to compilers for the language. Here are some of the key results of these efforts.

- Integration of specifications & program into a single language.
- Use of message buffers as concurrency mechanism, and first proof methods for buffers. The simple structure of concurrent programs has allowed proofs of significant, real-world, distributed programs -- verified Gypsy programs have actually run on distributed hardware!
- The Independence Principle -- an enforcement of modularity in program code, specifications, and proof -- allows incremental program development (top-down, or bottom-up) with manageable consequences to the verification.
- Incremental development manager
    - manages ripple effects of changes during program development and verification,
    - only invalidates proofs that use changed elements of the program,
    - provides a verification *environment* that integrates program development, specification, and verification, along with editing and compilation.

- Interactive theorem prover to decouple automatic proving technology from practice of verification. The prover helps in discovering proofs (the hard part, the intellectual challenge), but should not stand in the way of the user completing the rest of the verification if a single formula cannot be proven.
- Forward VC generation & natural deduction prover help the programmer/verifier maintain context during development.[4]

[4] [Good 70] is one of the first descriptions of "forward accumulation" in VC generation, although this was not the major reason for choosing forward VC generation for the GVE.

Chapter 3
SOME REMARKS ABOUT GYPSY PROGRAMS

3.1 Why the Semantics of Gypsy are Easier than Many Other Languages

3.1.1 Parameter Passing Mechanisms

Gypsy outlaws the uses of variables that commonly cause problems in programming and are awkward in semantic definitions. The following common programming language sore spots are illegal:

- dangerous aliasing (overlapping parameter references to **VAR** parameters)
- general shared variables (shared between concurrent processes)
- undetermined values resulting from routines that exit abnormally (e.g., after a run-time arithmetic error).

In Gypsy programs the call-by-value-result and call-by-reference mechanisms for passing data parameters are equivalent.[5] Banning dangerous aliasing for data parameters and returning well-defined values when routines terminate abnormally eliminate the circumstances in which these parameter passing mechanisms are distinguishable for data parameters.[6] Buffer parameters represent a restricted case of shared variables, and the no-aliasing rules are slightly relaxed. Thus, slightly more complexity is required to deal with passing buffer parameters within a concurrent procedure call.

3.1.2 No Dangerous Aliasing

No dangerous aliasing is *ever* allowed within the argument list of a single procedure. This means there is no need for explicit use of *locations* (see, for example, [Stoy 77], or [Tennent 81]) in the semantics of Gypsy.

Locations are an elegant mechanism for modeling:

- explicit storage allocation and deallocation,
- call-by-reference parameter passing,
- aliasing in parameter passage, and
- dangling references.

Locations allow this by introducing a level of indirection in resolving all identifier references. This mechanism

[5]This is not the case for buffer parameters.

[6]The Gypsy 2.0 condition **VARERROR** has been eliminated in Gypsy 2.1. Type consistency restrictions in Gypsy 2.1 assure that the assignment to the formal parameters required by "call by value" can be performed properly if the actual data parameters yield proper values.

is useful when we want to include a certain level of implementation details in the semantic definition, or when the programming language in question requires this more cumbersome mechanism. We choose not to use this mechanism, and Gypsy does not require it.

3.1.3 Simple Control Structures

Gypsy does not have a **GO TO** statement. Further, all control structures must always be exited "properly." It is impossible for a statement to abort the execution of an encompassing control structure. (Signalling a condition does not abort encompassing control structures -- the encompassing control structures may handle the condition, or, in the case of routines, may map the condition name into some other condition name.) Thus, evaluation of Gypsy programs follows the conventional stack model of expression evaluation (augmented to handle conditions).

Consequently, there is no need to use the cumbersome (but fully general and powerful) mechanism of *continuations* to describe Gypsy control structures. (See, for example, [Milne 76], or [Gordon 79].) Simple function composition is sufficient. We do not need to introduce the explicit notions of an interpreter and control stack. The basic notion of function evaluation is enough.

3.1.4 Procedure Calls

Since no non-local variable references exist within Gypsy procedures, procedures cannot refer to global variables. Similarly, since no **GO TO** statements exist in Gypsy, no global labels can be referenced. Further, there are no non-local exits from the procedure body during execution of a procedure call. In addition, the signalling of error conditions involves the normal (proper) unwinding of the calling stack. Thus, the proper "post-lude" of the procedure is always executed, and no "pre-mature block exit" is possible.

3.1.5 Run-Time Error Conditions

Gypsy defines a mechanism for handling exception conditions that arise during execution of a program. There is no exception handling defined for specifications, and the remainder of this section deals only with executable functions. During execution, all routine invocations must either:

- terminate normally,
- terminate abnormally (i.e., signalling an exception condition), or
- fail to terminate.

In effect, all executable functions and procedures that terminate, either normally or abnormally, under all circumstances, are total functions. There is no need to worry about clean termination of executable functions. (Routines are defined to return an exception condition if they do terminate abnormally. Thus, all routines return values composed of a condition name -- perhaps ***NORMAL** -- and the function result or procedure output parameters. The execution-time notion which occurs in some programming languages, of a function yielding an undefined result, does not occur in Gypsy. A function that terminates abnormally simply yields a value with the condition name being something other than ***NORMAL**.)

Restrictions on the proper domain of a function can appear as "type restrictions" on the formal parameters, or as predicates in the entry specification. Routine invocations need not terminate normally simply because the actual parameters are type consistent with the formal parameter list, and the routine's entry specification is satisfied. In fact, functions that never terminate normally are perfectly acceptable, as for example,

```
function Never (x:integer) : boolean unless (cond C) =
  begin
    entry x in [1..10];

    signal C;
  end;
```

This function always terminates abnormally, signalling its condition parameter. This declaration denotes a function that is defined over all integers if we acknowledge that functions return an ordered pair (a condition and a normal value).[7] In this case, **Never** would return the pair **(C, FALSE)**, where **C** stands for the function's actual condition parameter, and **FALSE** is the default initial value for type boolean.

The only way that a sequential routine may fail to terminate and yield a value is to execute a non-terminating **LOOP** statement, or invoke a routine that does not terminate. Gypsy **SEND** and **RECEIVE** statements are modeled as procedure calls, so blocking for input or output also fits this model.

3.2 Data Abstraction

Data abstraction is merely an enforced programmer discipline and has no effect on dynamic semantics. The discipline forces the programmer to avoid having Gypsy programs and proofs depend on the concrete representation of data types.

In Gypsy, abstract data type declarations contain a list of routine names, identifying the routines inside which the concrete data representation is visible. The visibility restrictions affect type checking and accessing of the concrete representation in specifications and executable code. Within routines that have concrete access to an abstract type, the abstract type is considered type equivalent to the concrete type. Outside those routines, the two types are not considered equivalent.

Visibility restrictions affect the proof methods implemented in the Gypsy Verification Environment. Thus, proof dependencies are restricted to reflect the dependencies permitted in routines. The proof of a routine lacking concrete access to an abstract data type, just as the executable code of that routine, cannot directly depend on the concrete structure of that data type.

Later in the discussion on generation of verification conditions, we see that function names are left uninterpreted. Information about the behavior of the function is introduced when external specifications of the routine are brought into the proof.

3.3 Duality of Function Definitions

3.3.1 Specification Functions and Executable Functions

Function declarations in Gypsy include the following aspects:

- executable code (i.e., input to the compiler to generate machine executable code),
- internal and external specifications (i.e., input intended for purposes of formally verifying properties of the program)
- type specifications (which are used for both compilation and proof purposes).

[7]It would be totally undefined if we only considered the normal value portion of the result.

Within this Gypsy function declaration there are actually *two* function definitions -- one executable, the other a specification. These are distinct and separate (though related) function definitions. *The main goal of verification of Gypsy programs is to prove that the executable function conforms to the specification function.* That is, both yield the same value whenever both are well-defined. We say that the executable function is a *restriction* of the specification function. For convenience, we consider a function to be a restriction of itself.

The entry and exit specifications[8] define a function (in a purely mathematical domain). Conjuncts in function exit specifications can be divided into two classes: function definitions (i.e., conjuncts of the form **`RESULT = ...`**, and similar forms), and everything else. Any conjunct appearing in a function exit specification that is not a function definition can be written as a Gypsy lemma.[9] Hence, we will assume that exit specifications contain only function definitions.

The executable body of the function declaration defines the function that is to be evaluated when the program runs.[10] These two definitions are intended to be closely related to one another -- that is precisely the role of the Gypsy proof methods.

> *It is a key concept in composing and verifying Gypsy programs that these two function definitions are distinct, and that they serve very different roles within the Gypsy proof methods.*

Specification function refers to the function defined in the exit specification. ***Executable function*** refers to the function defined in the executable body of the declaration.

3.3.2 Specification Functions and Specification Abstraction

Specification functions are the basic Gypsy mechanism for proof and specification abstraction. Abstraction plays a parallel role in program implementation, specification, and proof. Just as procedural abstraction is used to suppress implementation details in the executable program, the abstraction of the specification function is used to suppress details of the implementation and specification in the proof of the program. In both cases, the object is to build an implementation or proof structure that is modular; an internal modification to one function should have only minor, local effects through the rest of the program or proof. This property has been recognized as a key in program implementation (e.g., see the literature on structured programming, top-down design, and step-wise refinement) and has been identified as a key step toward practical program verification.

3.3.3 The Relation Between the Two Functions

In our proofs and specifications, we use the specification function defined by the function declaration *in place of* the executable function. All Gypsy specifications (e.g., **`ASSERT`** statements, **`ENTRY`** and **`EXIT`** specifications) use the function name to refer to the specification function. Thus, by requiring that the executable function be a restriction of the specification function, properties proven using the specification function will hold for the executable function when it terminates normally.

---

[8]The following remarks discuss only abstract specifications. These comments mostly apply to concrete specifications, except that Gypsy lemmas can only refer to abstract properties.

[9]Actually, definitions could be written as lemmas, too, but the proof methods for proving that the executable function conforms to the specification function is keyed to the exit specification.

[10]A particular Gypsy implementation may have constraints that cause its execution of a given Gypsy program to evaluate some function that is a restriction of the function defined by the executable body. That is, the implemented executable function may signal **`ROUTINEERROR`** or **`SPACEERROR`** under some circumstances under which the executable body would not.

Specifications may describe program behavior on a set larger than the proper domain[11] of the executable function. If so, our proofs will include consideration of cases that do not correspond to any possible program execution, as well as all the cases that do correspond to possible program executions. Thus, it is important that the specification function be a suitable stand-in for our implementation in each reference. To assure this, we must prove the following:

- that the executable function conforms to the specification function (i.e., the executable function, which may be a partial function, is a restriction of the specification function),
- that the specification function is a well-defined function over some specified domain, and
- that in our specifications and proof each use of the specification function applies the function to actual parameters in its proper domain.[12]

An additional property is desirable (in terms of keeping our specifications and proofs closely related to the expected behavior of our program):

- that the executable function be well-defined (i.e., terminate normally) over some identified domain.

This last property allows us to restrict use of the specification function to instances that correspond to possible uses of the executable function. If we omit this restriction, then we are free to extend our specification to describe extensions to program behavior (and hence prove theorems about behavior) that cannot be realized by the executable program. Nothing is wrong with proving these possibly interesting theorems, but it is desirable that the proof methods help us to understand when we have drifted from proving things about our executable programs and proceed to prove properties of our specification abstractions.

3.3.4 Domains of the Two Functions

The ***proper domain*** of a function refers to a set of values over which the function terminates normally and returns a proper value (i.e., not bottom, the undefined element, and -- in the case of an executable function -- not an abnormal condition).

The specification function definition given in the exit specification should probably be a total function. For the time being we will relax this view, and only require that the specification function be well-defined when the entry condition for the function declaration is satisfied. Thus, the proper domain for the specification function is identified by the entry specification.

The proper domain for the executable function may be smaller than that of the specification function.[13] In this case, the executable body will signal an abnormal termination condition for some input values that satisfy the entry specification. Our proof methods must assure that whenever the executable function terminates normally and yields a value, the value is precisely that of the specification function definition. In other words, the executable function must be a restriction of the specification function (i.e., they must agree wherever both are defined, and the proper domain of the executable function must be a subset of the proper domain of the specification function). Further, if we prove the proper domain of the executable function is a subset of the proper domain of the exit specification, then we can assume the well-definedness of the exit specification in any

[11]The proper domain of an executable function is the set of input parameter values for which the function terminates normally and returns a proper value.

[12]The current GVE only proves these last two points for the Gypsy predefined functions, not for user defined functions. A future enhancement of the GVE will remove this potential for constructing unsound proofs.

[13]It can even be larger, but we are not interested in any input values not admitted by the entry specification, because they are neither relevant to our proof, nor to the expected program behavior at execution time.

context corresponding to normal termination of the executable function.

3.4 Prescriptive versus Descriptive Specifications

We divide specifications of routines into two classes: *prescriptive* specifications and *descriptive* specifications. A prescriptive specification lays down a rule, and dictates that certain actions must be taken when computing the function. A descriptive specification expresses qualities or properties of the function, serving to describe necessary conditions when certain actions are taken. This is a particularly important distinction when defining functions that may terminate abnormally. Prescriptive specifications require that the routine signal an exception condition when a certain precondition is met (or not met). Descriptive specifications specify what the normal result of the computation should be (without regard to the preconditions of any particular implementation of the computation), and describe conditions that hold if an exception condition is signalled. Thus, the descriptive specifications permit an implementation to compute a correct result whenever possible, specifying a predicate that holds when an exception condition is signalled. A prescriptive specification gives a precondition that defines a set of values outside the proper domain of the function; a descriptive specification gives a characterization of a superset of values outside the domain of the function.

Below we define **foo** prescriptively, and **bar** descriptively.

```
         {Prescriptive version:  }

function foo (x:T) unless (cond abnormal_cond)
   begin
     entry prescriptive_predicate(x)
              otherwise abnormal_cond;
     exit  foo(x) = alter_x(x);
     result:=alter_x(x) unless (abnormal_cond);
   end;

         {Descriptive version:  }

function bar (x:T) unless (cond abnormal_cond)
   begin
     entry true;
     exit case (is normal:        bar(x) = alter_x(x);
                is abnormal_cond:
                            descriptive_predicate(x));
     result:=alter_x(x) unless (abnormal_cond);
   end;

function alter_x (x:T) unless (abnormal_cond) = pending;
```

Foo is not allowed to return a normal result outside the set characterized by **prescriptive_predicate**, while **BAR** is not allowed to signal an exception condition outside the set characterized by **descriptive_predicate**. Thus, in verifying **BAR**, we would have to establish that **descriptive_predicate** holds whenever **Alter_x** terminates abnormally. This is obviously required, since **BAR** terminates abnormally precisely when **Alter_x** does.[14]

[14]Implementations can signal implementation error conditions, such as memory size limits, which can further restrict the actual domain of a function. In this case, exception conditions, such as stack overflow, can arise at execution time, causing **BAR** to terminate abnormally, even though **Alter_x** is normally defined for those arguments. See section 3.5, page 18.

3.5 Relation of Proofs to Real Implementations

All implementations of Gypsy implement only subsets of full Gypsy language; they are required to compute restrictions of the functions defined or described by the formal meaning of the program (i.e., implemented functions can signal conditions more often than the formally-defined executable function, but must always agree when both are well defined). In practice, implementations are likely to "punt" and signal an abnormal condition when computations require arithmetic precision greater than the hardware word size, or when a program requires more memory than is physically present. Such behavior is permitted by this semantic definition, if the appropriate abnormal condition is signalled.

The mechanism for proving absence of run-time errors discussed in [McHugh 83] allows us to talk about clean termination of a Gypsy program on real hardware, rather than only on an ideal Gypsy machine.

3.6 Nondeterminism in Gypsy

3.6.1 What is nondeterminism, and why do we have it?

A routine is nondeterministic if its result (or output) is not a well-defined mathematical function of its input. A mathematical equality function is required to be symmetric, transitive, and reflexive. Since we do not know that `f(x) = f(x)` when `f` is a nondeterministic function, we would have to restrict very severely our use of equality, and much of our ability to apply normal mathematical theorem proving and algebraic techniques. To avoid this, Gypsy specifications functions are constrained to be deterministic.[15]

Nondeterminism in programming languages can arise for several reasons.

- imprecise language definitions
- precise language definitions that intentionally leave open some implementation choices (usually in the interest of efficiency or ease of implementation)
- languages intended for writing algorithms at a "high level" -- not allowing the programmer to specify "low level" (and presumably unimportant) details of the program
- data abstraction (i.e., functions may appear deterministic at the abstract level, but not at the concrete level. Consider a function that performs a table look up, and as a side effect reorders the table to speed up future queries, preserving abstract equality on the table, but not concrete equality.)
- concurrency (either due to nondeterministic parallel computations within a routine, or due to interaction with external concurrent computation).

3.6.2 Sources of Nondeterminism in Gypsy

Gypsy procedures are permitted to behave nondeterministically. The Gypsy 2.1 language has three sources of

[15]And the Gypsy verification system requires that specification functions be proven to be deterministic.

nondeterministic behavior.[16]

a. Explicit Concurrency. The **COBEGIN** statement, which supports possible parallel execution of program parts, does not specify any timing constraints on the order of events in different arms of the **COBEGIN**. Three functions defined on buffers (**FULL**, **EMPTY**, and **CONTENT**) can reveal properties of buffers as truly shared objects. Thus, these functions may behave nondeterministically.

b. Nondeterministic Buffer Polling. The **AWAIT** statement allows a sequential Gypsy procedure to wait until any one of several buffer operations can complete. For example, a procedure could thus respond to input from any one of several input buffers, using the **AWAIT** to suspend its execution until any one of the buffers became non-empty. This is essentially a nondeterministic **CASE** statement in which the case arms are labeled with buffer operations. The nondeterminism is at the statement level, and does not threaten the determinacy of our expression language.

c. Data Abstraction. Procedures in which the concrete representation of an abstract data type is visible can reveal information about that concrete representation. Thus, the procedures may not behave deterministically from an abstract point of view. Functions with concrete access to the data type are not permitted to violate the visibility rules of the abstract data type.[17]

3.6.2-A Apparently Nondeterministic Procedure Calls

Procedure invocation (as opposed to function invocation) need not appear deterministic. Thus, results are permitted to depend on external events or scheduling decisions. Procedure results can differ by violating data abstraction rules (e.g., a procedure can produce different results when called with two abstract objects that appear to be equal at the abstract level).

3.6.2-B Resource Errors and Determinacy

Any routine invocation can terminate abnormally if for some reason the processor cannot complete the computation properly. Abnormal termination is reflected in the calling environment by signalling a condition. When a function invocation terminates abnormally, it does not yield a normal result value to be used in further computations. Our requirement for determinacy is restricted to the function invocations that yield normal values, since only these are in our specifications and proofs. Signalling an abnormal condition is reflected in the procedural execution of a routine, and the process of generating verification conditions maps this procedural behavior into the functional domain of verification conditions.

---

[16]Gypsy 2.0 had an additional source of nondeterminacy, expression evaluation. Within a Gypsy 2.0 expression, the language specifies that arguments (or nested expressions) are evaluated before higher level (or containing) expressions. There are two degrees of freedom in the order of evaluation:

i. of evaluation of actual parameters in a routine call, and

ii. of evaluation of arguments to an infix operator (e.g., "+").

(Actually, these are the same, as infix operators are merely syntactic abbreviations for function calls to predefined functions. Thus, the arguments are actual parameters in a routine call.) The flexibility permitted the implementor is *only* in the choice of which of several possible exception conditions to signal. If the expression yields a normal value, then all Gypsy implementations must produce the same result. *For this reason Gypsy 2.1 specifies the order of evaluation of expressions and function arguments.*

[17]All concrete access functions must be shown to behave deterministically under substitution using abstract equality. That is, the function must yield results that are equal under an appropriate equality function when it is applied to two sets of actual parameters that satisfy the abstract equality function associated with the abstract data type. See Chapter 9 for more details.

3.6.2-C Concurrent Routine Calls

The order of execution in a concurrent routine call (**`COBEGIN`** statement) is unspecified to allow the parallel routines to operate asynchronously. Sometimes their execution order will be determined by explicit external communication (i.e., outside the routine and its descendants); sometimes it will be determined by implicit external communication (e.g., a clock-driven scheduler); sometimes it will be determined by explicit internal communication between the concurrent routines. Often the system behavior will be determined by some combination of these circumstances.

3.6.2-D Expression Evaluation

Expression evaluation always yields a well-defined value, or a well-defined exception condition. Actual parameters in routine call statements are evaluated from left to right. If evaluation of one of the actual parameters terminates abnormally, then that condition is propagated.

The normal result of a Gypsy function must be determined by the actual parameters of the function call. The determinacy of the normal exit case may require proof that concurrent routine calls evaluated within the evaluation of the function are themselves deterministic. Abnormal termination conditions need not be deterministic, since abnormal termination can depend (for example) on implementation resource constraints.

The language constraints on expression evaluation derive from the following two goals:

- assuring that program execution can be accurately modeled by the algebra of the proof system,

and within that,

- allowing the implementor maximum flexibility.

In general, specifications of the predefined routines in Gypsy are of the form

If the routine exits normally, then the result is exactly ....

The language does not specify exactly when a routine will exit normally. Often the specification will not even specify when a routine must not exit normally. The main and pervasive constraint is that *if* the routine exits normally, then the result *must* be algebraically correct.

Further, every attempt is made in the language definition to be as machine independent as possible. Thus, while implementation constraints (such as integer overflow) are acknowledged as possible exit conditions for predefined routines (such as add), the circumstances under which these error conditions will be signalled are defined in terms of function definitions *supplied by the implementor*. The language definition is independent of what a particular implementation can properly evaluate. Any implementation is a restriction of the full language. A completely degenerate implementation, which cannot properly evaluate anything, might reduce all programs to signalling **`ROUTINEERROR`**. (That is, an implementor can always punt on any feature. Of course, most implementations attempt to support at least *some* useful programs.)

The language definition, particularly with the constraints on order of evaluation, requires the implementation to signal an appropriate error condition at a relatively well-defined time. The time at which the condition must be signalled is constrained by the order of observable side effects. Normally, this would mean that the signal must occur during expression evaluation, before the next side effect is performed. (In fact, the requirement is that the program state be consistent with the statically observable state from the original program, given the point of the signal. The implementor is free to evaluate pieces of the program in arbitrary order, only if the observable program state, order of observable side effects, and routine results are correct. The implementor is quite free to let incorrect computations proceed, only if the program state is rolled back appropriately before the error