A REPORT ON THE DEVELOPMENT OF GYPSY

Donald I. Good
Richard M. Cohen
Lawrence W. Hunter

October, 1978 ICSCA-CMP-13

CERTIFIABLE MINICOMPUTER PROJECT
Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712
ACKNOWLEDGEMENTS

The development of Gypsy has been the cooperative effort of many people with diverse experiences and interests. Allen L. Ambler led the development of Gypsy 1.0, and the major concepts and directions set forth then have remained unchanged. James C. Browne, Wilhelm F. Burger, Carol A. David, Dwight F. Hare, Charles G. Hoch, John H. Howard, James Keeton-Williams, Judith S. Merriam, Mark S. Moriconi, Larry M. Smith, and Robert E. Wells also have contributed significantly to the design, development, and implementation of Gypsy. Kathryn S. Richardson and Laura K. Metheny have prepared numerous versions of several manuscripts.

The development of Gypsy also has drawn significantly from a number of other languages. Gypsy was developed primarily from Pascal and still bears a considerable resemblance to it. The ideas and features of Algol 60, Algol 68, Alphard, CLU, Concurrent Pascal, Euclid, Fortran, Nucleus, Simula, and Special also have influenced the design of Gypsy.
ABSTRACT

The first version of Gypsy was introduced in 1976 to support the specification and construction of verified programs. A second version has evolved based on the experiences of the last two years. The changes introduced in the second version are described. Some experiences with the specification and proof methodology are discussed, and the status of the implementation of the Gypsy compiler and verification system is summarized.

Keywords and Phrases:

Gypsy, programming language, specification language, correctness, program verification, systems programming language, reliable software, scope of names, dynamic storage management, concurrency, compiler design, programming language design

CR Categories: 4.12, 4.2, 4.22, 4.34, 4.43, 5.24
### TABLE OF CONTENTS

1.0 INTRODUCTION ........................................... 1

2.0 HISTORICAL PERSPECTIVE ............................... 1

3.0 LANGUAGE DESIGN ....................................... 4
   3.1 Scopes ............................................... 5
   3.2 Types ............................................... 5
   3.3 Exception Handling ................................. 7
   3.4 Lemmas ............................................. 8
   3.5 Concurrency ....................................... 9
   3.6 Dynamic Storage Management ..................... 12

4.0 INCREMENTAL VERIFICATION SYSTEM ................. 13

5.0 COMPILER ............................................... 14
   5.1 Use of Verification-Compatible Intermediate
       Language ............................................ 14
   5.2 Achieving Self Support ............................ 15
   5.3 Informal Application .............................. 15

6.0 CONCLUSION ............................................. 16

BIBLIOGRAPHY ............................................... 17
A REPORT ON THE DEVELOPMENT OF GYSY

Donald I. Good
Richard M. Cohen
Lawrence W. Hunter

1.0 INTRODUCTION

The design and development of Gypsy has resulted from the development of an integrated system of methods for building operational software systems of 1000-2000 lines of code. The bases of the methodology are formal methods for specifying and programming small-scale systems and for verifying the consistency of the system programs and their formal specifications. The Gypsy language is based completely on these formal methods, and it provides a way of applying these methods in actual practice. Gypsy is a single, unified language for expressing both programs and formal specifications. The language contains both a specification component and a programming component which strongly intersect. Gypsy is designed so that it may be used as either a formal specification language or a programming language. However, the language is used most effectively as a verifiable program description language, the term "program description" referring collectively to both a program and its specifications. Gypsy is designed so that effective, formal proof methods exist for verifying the consistency of all program descriptions in the language. This verification often also can be done by validating the specifications at run-time. Formal and effective provability has been an absolute requirement of the design and development of Gypsy.

The methodology upon which Gypsy is based has been developed over a four year period with the first report on Gypsy 1.0 appearing in [Ambler, 76]. Successive experiences using and implementing Gypsy have led to a second report on Gypsy 2.0 [Good, 78-1]. This paper summarizes the significant developments that have been made in progressing from Gypsy 1.0 to Gypsy 2.0; it also summarizes the current status of the implementation of the incremental verification system and compiler for Gypsy.

2.0 HISTORICAL PERSPECTIVE

The development of the methodology upon which Gypsy is based began in September, 1974 motivated by a clear and increasingly pressing need to build small-scale systems that perform critical functions with very high reliability. The general approach of the methodology was to build formally verified software that runs on fail-secure hardware. One of the specific goals has been the development of formally verified communications processing systems of 1000-2000 lines of code. Communications processing was selected as the particular applications area because these
applications typically are small-scale systems that contain many problems common to more general systems. A specific application area also was desired in order to keep the development of the methodology clearly directed at solutions to the problems of real systems. By systems standards, systems of 1000–2000 lines are small; but by formal verification standards, they are very large. Verifications of this scale required that the methodology be developed with equally strong emphasis on both the theoretical and the practical problems of verification.

The development of the methodology was begun with no intention of defining a new language. It was recognized that a proper choice of programming language would be of critical importance, but it was expected that Pascal [Jensen, 74] would provide an adequate and established basis for verification. As both the theoretical and practical problems of verifying systems programs became better defined, several important requirements of the programming language became clear. In order to support systems programming, the language would need to have concurrency and exception handling facilities. Also, as a critical practical matter, the language would have to be structured so that program modifications (during either development or maintenance) would cause minimal effects on a formal verification. Recompiling a 1000–2000 line program to accommodate a modification of a few lines might be quite feasible, but reverifying the entire program to accommodate these same small changes would be totally impractical. This pragmatic necessity indicated a need for independent and incremental verification of program components and a need for a data abstraction mechanism.

It also became clear very early that a well-defined language for formal specification would be required. Approaches to specification methods that could be used as bases for formal verification were only in the very early research stages, and none of these had yet been developed into a well-defined specification language. Three basic approaches were identified: the "assertion", the "state machine", and the "algebraic" approaches. The "assertion" approach, which consists of annotating a program with various kinds of logical assertions, was developing as a matter of necessity along with formal verification methods. This approach was typified by the work of [Hoare, 73] and [Tgarashi, 73] in defining proof rules and generating verification conditions for Pascal. This approach had been used in specifying a verification condition generator for a simple programming language [Ragland, 73] and has since been used in specifying parts of a prototype security kernel for Unix [Walker, 77]. The second approach was the "state machine" approach based on [Parnas, 72]. This approach has since been developed into a formal specification language, Special [Roubine,76], and used to specify the design of a provably secure operating system [Neumann, 75]. The third approach was the "algebraic" approach [Liskov, 73].
No one of these approaches had (or has) demonstrated a clear superiority over the others. Therefore, in building a general methodology for verifying real systems, it seemed advisable not to rule out any one of the three. Since our primary purpose was verification, it was decided to adopt primarily the assertion approach and expand it to incorporate both of the other two to the extent possible, although retaining a unified approach to specification.

With this assessment of the requirements of the methodology, Pascal was investigated as a language to support application of the methodology. Pascal was based on sound verification principles. However, several substantial modifications would be needed to provide an effective language for fully verified systems. The remaining verification trouble spots would have to be revised or eliminated. Some significant restrictions would have to be imposed to enhance incremental verification. Facilities for data abstraction, concurrency, exception handling, and formal specification would need to be added. The choice was to attempt to retrofit these substantial modifications to a language designed without formal verification as a primary goal, or to design a new verifiable language which could be adjusted freely to suit the needs of the newly developing methodology. The decision was made to develop a new language that was firmly based on the well-established verification principles of Pascal. This would allow us to readily incorporate the benefits of previous experience with Pascal and yet not have the development of the underlying methodology constrained by a language designed previously and primarily for other purposes.

The actual development of the Gypsy language was begun in the spring of 1975 and has been developed in parallel with the integrated methodology for specifying, programming, and verifying systems. A major design decision was to use message buffers [Brinch Hansen, 73] rather than monitors ([Hoare, 74], [Brinch Hansen, 75], [Howard, 76]) as the basis for concurrency in Gypsy. This is the only part of the language design that has been strongly influenced by the choice of communications processing as a target application area. Early work was done with monitors, but it was found that the monitors almost invariably were used to implement simple bounded buffers. The buffers also supported a less complex method for proving concurrent processes.

The first syntax analyzer for Gypsy was completed in the spring of 1976 as a major component of the incremental verification system that was being implemented in order to apply the methodology effectively in practice. The first report on the language Gypsy was issued in August, 1976 [Ambler, 76]. During 1976, Gypsy 1.0 was also being used in its first major trial application, the Network Communications System (NCS) [Wells, 76]. NCS is a three-layer message switching system that resembles the IMP subnet of the ARPANET. Messages are delivered among a fixed set of users over a five node packet switching network. The complete NCS consisted of approximately 1500 lines of formal
Gypsy specifications and 1000 lines of code involving over 20 concurrent processes. These processes were formally verified (manually), but NCS was not actually run because work on the Gypsy compiler had not yet begun. The results of the NCS experiment were given in the first report on the complete methodology [Good, 77-1]. ([Good, 77-2] is a summary of this report.)

The NCS experiment pointed out the need for several revisions in the methodology and in Gypsy (see [Wells, 76]). These experiences, coupled with a compiler implementation effort begun in the spring of 1977, have led to a new report on the language Gypsy 2.0. Gypsy 2.0 and the improved methodology currently are being applied to produce a small-scale, verified subsystem that will be tested as part of a network security experiment on the ARPANET. Initial top-level specifications for this program have been completed [Horn, 77]. These 920 lines of specifications include specifications for the relevant parts of the functional behavior of the ARPANET. Work on implementing and verifying this program is currently in progress.

3.0 LANGUAGE DESIGN

Several major modifications in the language definition have been made in progressing from Gypsy 1.0 to Gypsy 2.0. These revisions are primarily ones of consolidation and clarification and, in some cases, extension. An explicit "scope" declaration has been introduced to provide a hierarchical name space control and yet retain the flat program component definition structure that significantly enhances independent verification and compilation of routines. A simple and rigorous definition of type consistency for parameter passage has been formulated. Exception condition parameters have been separated from the data parameters in both the actual and formal parameter lists. A "lemma" specification has been introduced that is a generalization of the previous "axioms" for data abstraction. Lemmas also provide an explicit and effective way of constructing well-structured proofs. The concurrency mechanisms have been revised to make a procedure call a degenerate cobegin, and to support more effective methods for specification and proof. A unique dynamic storage management facility has been introduced, which is very convenient for verification, yet does not involve explicit pointers.

Several major features of Gypsy 1.0 have been eliminated completely. Macros were eliminated due to lack of use, and because their effect could largely be absorbed by proper use of functions. Functions have the additional advantage that specifications can be written for them. Axioms were eliminated in favor of lemmas. Process declarations and the bag (multiset) structure were eliminated as part of the concurrency revisions. Ordinary procedures are now used where processes were required,
and bags are no longer required in proofs of cobegins. Parameterized types and all timing facilities have been eliminated pending further study.

3.1 Scopes

Gypsy is a "flat" language, without the hierarchical, tree-structured scope of names introduced by Algol-60 and preserved in Pascal. Most of this mechanism is unnecessary in Gypsy because of the absence of global variables. Because function, procedure, and type definitions are not nested, Gypsy 1.0 programs required a great many unique names, and the language offered little help in organizing the program text to reflect the program structure. To alleviate this situation, Gypsy 2.0 provides an explicit "scope" declaration. The scope declaration contains definitions of program components (procedures, functions, types, constants) local to that scope. It may also contain "name" declarations, which declare local names that refer to components defined in other scopes. The scope declaration provides a means for associating related components in the program text. In the following example,

```
scope A =
    begin
    procedure P1 (...) = ... ;
    procedure P2 (...) = ... ;
    end;

scope B =
    begin
    name P1 from A;
    name P2A = P2 from A;
    procedure P2 (...) = ... ;
    procedure P3 (...) = ... ;
    end;
```

Scope A defines procedures P1 and P2. Scope B declares the name P1 to refer to the procedure P1 in scope A, and P2A to refer to the procedure P2 in scope A. The renaming of procedure P2 in the context of scope B allows scope B to define its own local procedure P2.

3.2 Types

Two major revisions have been made concerning types. First, a rigorous and simple definition of the type consistency of actual and formal parameters of routines has been formulated. Second, the data abstraction mechanism has been revised to permit extension of the "=" operator to abstract types and to permit concrete invariants. Extending the "=" operator is a necessary
part of the proof methodology.

Each data type in Gypsy defines a value set. The primitive types (integer, character, boolean, and rational) each define a specific predefined value set. Scalar type declarations explicitly define their value set. The value set for a structured type (array, record, set, sequence, or mapping) is defined by an appropriate cartesian product on the value sets of each of the component types. Arrays and records are of fixed size, while the size of sets, mappings, and sequences may vary dynamically. As in Gypsy 1.0, new types may be defined by placing restrictions on other types. The most common restriction is a range restriction. (E.g., integer[1..10]) A size restriction declares the maximum permitted size for a dynamic structure. The base type of a type is the type having the same structure, but with any restrictions removed at each level.

Types in Gypsy provide the means of checking the consistency between actual and formal parameters in routine calls. The type consistency rules are defined so that every routine can be analyzed completely based on the types of its formal parameters, without any reference to its actual parameters. The type consistency requirements between actual and formal parameters are very simple:

1. the value of the actual parameter must be an element of the value set of the type of the corresponding formal parameter in each routine call; and

2. if the formal parameter is a var parameter, then the value set of the type of the formal parameter must be contained in the value set of type of the actual.

In most cases, type consistency can be determined statically. In some cases, however, run time checks will be required. All parameters are passed by reference, and the rules assure that all manipulations done on a formal var parameter are valid on the actual parameter.

The concrete representation of abstract data types in Gypsy are encapsulated by a "composition access list". The composition access list acts as a gate and enumerates those components that may use the concrete representation. For example,

type stack
  <emptystack, push, pop, top, stackeq> =
  record (a:array[integer[0..10]]; p:integer[1..10])

defines the abstract type "stack". Type "stack" is considered primitive (i.e., its own base type) by all routines except "emptystack", "push", "pop", "top", and "stackeq". Only those routines may "see" that the body of "stack" is a record declaration.
Equality is defined for Gypsy primitive types, and for structured types by testing equality of each component of the structured value. Routines not named in the composition access list do not have access to the concrete representation, and may not apply the equality operator unless there is an explicit equality function defined for that abstract type. For example,

```lisp
function stackeq extends "="
   (s1,s2:stack):boolean = ...
```
defines "stack" equality. It partitions the value set of the concrete record structure into equivalence classes appropriate to the stack abstraction. The usual axioms of an equality relation are automatically added to any external specifications of a declared equality extension, and must be verified along with the user supplied specifications. Within routines that have concrete access to the type stack, the equality operator of stack is defined to be record equality, as it would be for a similar non-abstract structured type.

An abstract data type may have a concrete data invariant. The invariant must hold for all concrete objects representing objects of the abstract type. (See [Hoare, 72].) The addition of concrete data invariants, which requires all abstract objects to be initialized, has led to the definition of default initialization values for all types.

### 3.3 Exception Handling

Exception handling in Gypsy 2.0 is based on the work of Goodenough [Goodenough, 75], as it was in Gypsy 1.0. An important aspect of Gypsy "condition" handling is the propagation of conditions across routine boundaries. Condition names may be passed as parameters to a function or procedure. If the called routine terminates abnormally, it signals an appropriate condition in the calling environment. The terminated routine can only signal one of its condition parameters or the predefined condition "routineerror" into the calling environment. Any other conditions that are signalled, but not handled within the called routine are mapped to "routineerror" by the terminating routine.

The condition name parameter list has been separated from the data parameter list, both in the declaration and at the call site. The function sqrt declared
function sqrt(x:integer) unless (cond complexroot) = pending

could be called by

sqrt(y) unless (cannot).

If the condition "complexroot" were signalled in sqrt, and not handled within the function body, the function would terminate abnormally (without returning a value) and signal the condition "cannot" in the calling environment.

When calling a routine with a formal condition parameter list, the actual condition parameter list is optional. Thus "sqrt(y)" would also be a legal call to the function sqrt. In this case, the actual condition parameter list is filled in automatically with the names from the formal condition parameter list. The formal condition parameter list is part of the routine header, which is an external specification of the routine. Thus routine modularity is not violated. This is the normal mechanism for signalling standard errors at run time without using the "unless" part - e.g., "a[i]" may give an "indexerror".

3.4 Lemmas

Gypsy 2.0 lemmas are a generalization of Gypsy 1.0 type axioms. They are separate program components, rather than being embedded in type declarations. Lemmas support specification and proof. They may specify relationships between functions in the program, or may provide problem domain specific facts. The boolean expression forming the body of the lemma may be labelled as an assumption, or as a fact derived from the rest of the program and to be proven.

The body of a lemma may not refer to the concrete representations of abstract data types. (Thus use of a lemma in a proof cannot violate the principle of data hiding.) If the lemma has composition access to an abstract type, then knowledge of the concrete representation may be used in the proof of the lemma.

The lemma declaration takes a formal parameter list, and the parameters are used as free variables within the lemma body. A formal condition parameter list is not allowed. Thus no condition names are known within the scope of the lemma body, so no actual condition parameters may be passed in any function calls in the lemma. Since condition parameters cannot affect the value of a function reference, this is not a restriction on the power of the lemma. The boolean expression in the lemma body must be satisfied so long as the entry conditions of all referenced functions are satisfied. Thus partial functions may be used freely within lemmas. For example,
function push(s: stack; x: element): stack =
begin
  entry not isfull(s);
  
end;

function pop(s: stack): stack =
begin
  entry not isempty(s);
  
end;

lemma Pop_of_Push (s: stack; x: element) =
(prove pop(push(s,x)) = s);

Proof of this lemma requires proof of the theorem

not isfull(s)
& not isempty(push(s,x))
-> pop(push(s,x)) = s,

where the hypotheses are derived from the entry conditions for
the functions referenced in the body of the lemma.

3.5 Concurrency

Two revisions of major importance have been made in the
concurrency facilities. First, the "cobegin" statement, which is
the major source of concurrency, is now a strict generalization
of the sequential procedure call. Second, the specification
notation for buffer histories has been modified to clearly
distinguish multiple activations of the same procedure. These
changes also support a substantially improved proof method that
permits a much more effective use of automatic algebraic
simplification in constructing and proving verification
conditions. The important property of independent verification
of concurrent processes has been preserved. These proof methods
are described in [Good, 79].

The cobegin is the statement for initiating concurrent
execution of procedures as in the following single
producer_consumer example:

procedure p_c(var x: in_buf; var y: out_buf) =
begin
  block not full(y)
    -> empty(x) and outto(y, myid) = infrom(x, myid);
  var b: obj_buf;
  cobegin
    move_obj(x,b);
    move_obj(b,y);
  end;
end;

procedure move_obj(u:in_buf; v:out_buf) =

begin
  block not full(v) -> empty (u) and outto (v,myid) =
  infrom (u,myid);
  var obj:object;
  loop
    assert outto(v,myid) = infrom(u,myid);
    receive obj from u;
    send obj to v;
  end;
end;

type obj_buf = buffer (2) of object;
type in_buf = obj_buf <input>;
type out_buf = obj_buf <output>;

The cobegin in the producer_consumer, "p_c", suspends execution of
"p_c" and initiates concurrent execution of two different
activations of procedure "move_obj". One moves "objects" from
buffer "x" to buffer "b", and the other moves "objects" from "b" to "y". The execution of "p_c" is resumed when (and if) all
activations initiated by the cobegin terminate. (In this example, they do not.) Thus, the cobegin exhibits the same basic control
flow as a sequential procedure call except that several
procedures may be activated rather than just one. If the cobegin
activates only a single procedure, its effect is identical to a
procedure call.

Interpreting the cobegin as a generalization of the
procedure call also involves a generalization of the
"non-aliasing" requirement. The normal Gypsy requirement for all
procedures is

Whenever a procedure is called, no change in the value
of any actual var parameter may affect the value of the
object referred to by any other actual parameter.

In Gypsy, all parameters (both "var" and "const") are passed by
reference. The potential satisfaction of the non-aliasing
requirement is enforced statically, and any parts that cannot be
enforced statically are checked at run time. Within a cobegin
each individual procedure call must satisfy the non-aliasing
require and the following:

The non-aliasing requirement applies to all actual var
parameters across all procedures invoked by the cobegin
except that fully buffered objects may appear as actual
parameters to more than one activation. (An object is
fully buffered if it is of a buffer type or of a type
consisting solely of fully buffered components.)

These requirements allow non-buffered objects to be used as
actual parameters within a cobegin in well-controlled ways. A non-buffered object may be a const parameter of any number of concurrent activations so long as it is not a var parameter of any one of them. Also, a non-buffer object may be a var parameter to at most one activation.

The second major change in concurrency is in expressing specifications. There are two important changes. One is the ability to express operation restrictions on buffers, and the other is a change to a functional notation for buffer histories.

The use of operation restrictions on buffers is illustrated in the preceding example by the types "in_buf" and "out_buf". Gypsy 2.0 has three special statements for manipulating buffers: "send", "receive", and "give". A buffer, such as "in_buf", that is "<input>" restricted may be used only as an input buffer; "send" and "give" operations may not be performed on it. Similarly, "receive" operations may not be performed on an "<output>" restricted buffer. These restrictions are checked statically.

The effects of these restrictions on specifications are that the input (output) history of an "<output>" ("<input>") buffer is known always to be null. This information normally is essential in proofs, and the buffer restrictions allow it to be expressed and checked statically without using the more complex formal specification and verification methods. For example, without the buffer restrictions, the "move_obj" procedure above would need to be written as

```
procedure move_obj(var u, v:obj_buf) =
begin
    block not full(v)
        -> empty(u) and outto(v,myid) = infrom(u,myid)
            and infrom(v,myid) = null(obj_seq)
            and outto(u,myid) = null(obj_seq);
    ...{same as above}...
end;
type obj_seq = sequence of object;
```

The buffer restrictions also provide for increased power in automatic verification, because they can be applied directly without going through several intermediate deductions to determine the same information.

The buffer restrictions are operation restrictions in the spirit of [Jones, 78]. Gypsy 2.0 allows these restrictions to be used on buffers because they have a strong impact on the specification and proof methods for concurrency. A similar impact on proofs involving other types has not yet been discovered, and therefore, a more general mechanism has not been developed at this time. Operation restrictions are allowed only on buffers.
The other major change in the specifications for concurrent processes is the use of a functional notation for buffer histories. This change appears to be primarily syntactic, but it was motivated by a need to clearly identify several different activations of the same procedure that may be running concurrently. This situation is illustrated in the cobegin of the producer consumer above. It initiates two activations of procedure "move_obj". These different activations must be clearly distinguished in proving the cobegin. This problem does not arise in proving sequential calls because one always deals with just one activation at a time.

To distinguish these activations, an implicit "const myid : activationid" formal parameter is defined automatically for every procedure. (Only procedures can be invoked by a cobegin.) An "activationid" is a predefined type that uniquely identifies procedure activations, and "myid" is the unique name of the arbitrary activation of the procedure for which it is an implicit formal parameter. For example, "infrom(x,myid)" in procedure "p_c" refers to the history of "objects" received into buffer "x" by the "myid" activation of "p_c". Activationids are useful only in specifications and proofs. The actual parameters corresponding to the implicit "myid" parameters are assigned as parts of the proof rule for the cobegin.

A more extensive multiplexor example of the Gypsy 2.0 approach to concurrency appears in [Good, 78-2].

3.6 Dynamic Storage Management

One of the major extensions to Gypsy 2.0 is the addition of a unique dynamic storage management facility based on ideas developed largely by Charles G. Hoch. The Pascal pointer mechanism was discarded because of verification difficulties and replaced with a mechanism based on "dynamic" structures. The dynamic structures are predefined structured types whose number of components can vary at run time. The basic operations on these types are component addition and deletion and the ability to move a component from one dynamic structure to another. The operations on the types are defined so that they can be implemented efficiently by a hidden pointer manipulation, and in such a way that every accessible object is pointed to by at most one hidden pointer. This approach eliminates the problem of dangling references, the aliasing problem normally associated with explicit pointers, and the need for garbage collection. The operations on the dynamic types also have well-defined algebraic properties that are similar to those for arrays.

The dynamic structures are sets, sequences, mappings, and buffers. Sets and sequences are modelled after their mathematical counterparts. The notation for sequences has been extended to allow subscripting, "s[i]", for element selection,
and to allow "s[i..j]" for selection of a subsequence. Buffers are, in essence, queues that are protected by mutual exclusion. Mappings correspond to discrete mappings of mathematics. A mapping has a domain type and a range type, and "m[x]" denotes the range element "y" associated with a given domain element "x" in a mapping "m".

The basic dynamic operations are done through the normal assignment statement and two special statements, "remove" and "move", as illustrated in the following simple stack example. Suppose we represent a stack by a sequence (in which elements can be allocated dynamically) rather than as a (statically allocated) record, as in the previous example:

\[ \text{type stack = sequence of element} \]

Let the top of the stack be "s[1]". Pushing "x" onto stack "s" can be written as "before s[1] := x". Popping "s" is "remove s[1]" and "move s[1] to y" pops "s" and assigns the (previous) top to "y". The assignment statement (push) creates a new sequence element, the remove statement (pop) deletes the first sequence element, and the move statement deletes the first element from "s" and assigns it to "y". In general, "move A to B" is functionally equivalent to "B := A; remove A". (Remove can only be used on components of dynamic structures.)

The "give" statement is similar to the "move" statement except that it is used strictly on buffers. "Give A to B" is functionally equivalent to "send A to B; remove A". This permits large objects to be passed through buffers without the copying expense of the "send" statement.

4.0 INCREMENTAL VERIFICATION SYSTEM

The first major "implementation" of Gypsy has been an experimental verification system. (This contrasts with most implementation efforts, which normally begin with a compiler.) The need for a verification system to support applications of the size expected for small-scale systems was absolutely clear, and therefore a verification system was implemented in parallel with the development of Gypsy and its underlying methodology. The mutual development of the basic methods, the language in which they are realized, and the verification system that implements these methods has led to many mutual benefits. The basic methods have been developed to be automatable, and the implementation has helped to raise and to clarify many language and methodological issues.

The verification system implements most of Gypsy 2.0. It is implemented in LISP on a PDP-10 and contains conventional components: syntax analyzer, verification condition generator, algebraic simplifier, and interactive theorem prover. The Gypsy
system, however, is unique in that these components are integrated under the control of a "design-verification manager" (DVM) [Moriconi, 77]. The syntax analyzer translates Gypsy program descriptions into a functional (prefix) form and the DVM stores the program description components in a data base that supports incremental verification. As the specifications of various components change during program development, the DVM keeps track of the effects of the changes, noting which routines are no longer type consistent and which verification conditions may have to be reproven.

The verification system has successfully verified a small message switching system containing "n" concurrent processes. Each process receives messages from a single source and sends them directly to the designated destination. The system also enforces a simple security constraint. The total system, including specifications, consists of about 200 lines of Gypsy (see [Moriconi, 77]).

5.0 COMPILER

The Gypsy compiler is a cross compiler from a PDP-10 to a PDP-11/03. (This choice was dictated by external constraints.) The compiler works strictly in conjunction with the incremental verification system. A program is developed and verified using the verification system, and then the prefix representation of the program, which is internal to the verification system, is written on a file for input to the compiler. This method of program development requires relatively heavy use of the verification system and relatively light use of the compiler. Therefore, the compiler is designed to produce efficient, optimized code at the cost of relatively slow compilation. To date, the significant aspects of the compiler development have been use of a verification-compatible intermediate language, achieving a high level of self support, and informal application of the Gypsy program development methodology.

5.1 Use Of Verification-Compatible Intermediate Language

The Gypsy compiler is a two stage translator. It translates from prefix, which is internal to the verification system, into object code through an intermediate language, Capability Machine Language (CML). CML is the assembly language of a hypothetical Capability Machine (CM). The CML Generator stage of the compiler translates prefix into CML; the Code Generator stage translates CML into object code. The CML Generator algorithm is universal and independent of the target machine. The Code Generator is target machine dependent.
The Capability Machine was designed as part of an investigation into architecture features for the support of verifiable software [Hoch, 78]. Its assembly language, CML, was designed to be compatible with verified software written in Gypsy. It is a low level language in that its operations are primitive with explicit direction of control flow. It is by design compatible with verified software written in Gypsy in that the CML image of a Gypsy source program has execution properties which are necessary for verification of the source program. In particular, CML enforces type consistency and non-aliasing.

5.2 Achieving Self Support

Gypsy is implemented as a cross compiler, so it is not amenable to conventional bootstrapping methods. Instead the compiler makes extensive use of "self support" to implement parts of Gypsy in a subset of itself. The "support" for a language implementation is the object code which must be present in order to execute a program in the language, but which is not part of the object code image of program statements. This includes "system" routines such as schedulers and interrupt handlers; it also includes interpreter routines which are called in the object code image of the program. "External support" is written in languages other than that being implemented; and "self support" is written in the language itself. The object code images of self support routines can use external support and any other support facilities already implemented.

The Gypsy compiler has been designed to minimize external support, to implement as much decision making logic as possible in self support, and to allow formal specification of all support routines. Self support for Gypsy includes scheduling and operations on structures of dynamic types.

5.3 Informal Application

The Code Generator stage of the Gypsy compiler was designed and written using the Gypsy formal specification and programming methods as much as possible in the absence of a suitable Gypsy implementation. The Code Generator is written in Pascal using only features common to both Pascal and Gypsy. This subset of Pascal was selected with the eventual goal of formally verifying the compiler in Gypsy. This restriction is imposed by informal requirements on the way Pascal code is written. Much of the self support software is being written directly in Gypsy.

Even though the Gypsy methodology is being used only informally, several benefits have been observed. Most noticeably, the strictly regulated routine interface requirements resulted in surprisingly few interfacing problems among routines.
written at different times and by different authors. Unexpected side effects were virtually eliminated. Interfacing problems consisted mostly of the need for transfer functions between types containing the same essential information, but in different layouts. The logic of many major routines was partially worked out as a result of writing specifications as comments, with exit specifications frequently serving as the top level design for routines.

6.0 CONCLUSION

Gypsy is a verifiable program description language that has been developed in conjunction with an integrated methodology for specifying, programming, and verifying software systems. The language and the methodology are being implemented through an incremental verification system and a compiler. The development of methods, language, and implementation has been guided throughout by significant experimental applications to real problems.

The objectives of a verifiable program description language are to increase program quality and, in particular, program reliability. A verification demonstrates that a program always executes according to specifications. This demonstration of consistency between a program and its specifications does not guarantee perfect program quality or reliability. It can, however, provide program quality that is significantly higher than that usually associated with unverified programs. Formal verifications can be performed with the same degree of rigor and precision that is used in mathematics. Therefore, verification can, in principle, raise program quality to a level comparable to mathematics. Though not infallible, the quality of mathematical proofs and analyses has far exceeded that of unverified programs. Attaining this level of program quality would be a substantial step forward. The main objective of Gypsy is to provide a language in which this step can be attained in practice as well as in principle.
BIBLIOGRAPHY


