# The First Generation Gypsy Compiler

Lawrence Waldo Hunter

ICSCA-CMP-23 June 1981

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

Institute For Computing Science And Computer Applications
Certifiable Minicomputer Project
The University of Texas at Austin
Austin, Texas 78712

Abstract

This report summarizes the significant results of implementing the first generation Gypsy compiler. Gypsy is a language for specifying, writing, and formally verifying computer software. The compiler implementation project gave us experience in compiling language features designed for verifiablility. It allowed us to observe programming styles used in the development of verified software, which in turn gave further insight into the design of compilers for such code. The Gypsy compiler also served as a testbed for the methodology of writing large, verifiable software systems. This report analyzes the strenghts and weaknesses in the first generation Gypsy compiler. It includes recommendations for the design of future compilers for verifiable languages.

# 1. INTRODUCTION

## 1.1 The Gypsy Language and CMP.

Gypsy is a high level language designed to support formal specification and verification of programs. Its syntax closely resembles that of Pascal, but it has several features which Pascal lacks and it has had some Pascal features removed. Gypsy has built in features for formal specification. Its operational features are all axiomatized to facilitate formal verification that Gypsy programs do in fact meet their formal specifications. [Good 78]

Gypsy is under development as part of the Certifiable Minicomputer Project (CMP) at the University of Texas. CMP is building a comprehensive methodology for development of reliable software based on formal specification and verification of programs written in Gypsy. Projects within CMP deal with both methods for developing reliable software and tools to support these methods. The tools are being integrated into the Gypsy Verification Environment (GVE). This is a prototype of a complete program development system including a file system, an editor, an interpreter, a formal verification system for Gypsy programs, and a compiler.

## 1.2 The Gypsy Compiler Project.

The Gypsy compiler project is an integral part of CMP. It served several goals within CMP in addition to translation of Gypsy code: it demonstrated the compilability of Gypsy and provided a vehicle to demonstrate Gypsy's usability as a general purpose language; it contributed to an investigation into hardware support for verifiable software; and it served as a testbed for the software development methodology.

The compiler itself is a component of the overall GVE. It operates from the program development data base. Syntax error analysis and diagnostics have already been handled by the parser which enters programs into the database. This makes the Gypsy compiler's job somewhat easier than that of a conventional compiler.

The role of the Gypsy compiler within the overall CMP, and the relatively limited manpower available for compiler development, dictated two priority decisions in the design and implementation of the compiler. First, the compiler development project should concentrate on new areas for which there did not already exist a body of knowledge about compilation methods. It was not intended to be a production quality compiler. Most of the effort should be spent on new language features, and less on already well-understood methods such as register allocation within the code generator. Second, optimization stages were made optional in several places in the compiler. It was expected that part-time student assistants could work productively on those stages without going through a long familiarization with Gypsy. Little effort was actually spent on optimization.

Development of the Gypsy compiler began in mid-1977. This was one of the first efforts to compile a language designed specifically to support formal specification and verification. The compiler became operational in late 1978, generating assembly code for the PDP-11. After analysis of the code produced by the compiler, several major improvements were made throughout 1979 and early 1980.

During late 1979 the compiler was reevaluated in light of the knowledge which had been gained about compiling verified code. Its performance was adequate: total compilation time of from 6 to 10 lines per second, generating from 7 to 20 words of code per line, for a language more difficult to compile than Pascal. But it was concluded that there were fundamental design and implementation flaws in the compiler, and that compiler development effort should be directed toward a completely redesigned second generation compiler. The first generation compiler was frozen except for maintenance and minor additions in mid-1980. The second generation Gypsy compiler is expected to replace the first generation compiler in 1981 or 1982 with significant increases in power and code quality.

This report is a summary of the significant results from the first generation Gypsy compiler development project. Much of this report deals with compilation of specific Gypsy features and the interaction of the compiler with the Gypsy Verification Environment. The Gypsy language and the entire program development system have properties which it is expected will be typical of verifiable languages and the manner in which they affect compiler design. So this report should apply to compilation of verifiable languages in general.

## 2. Verifiable Intermediate Language.

One of the overall CMP goals was an investigation into hardware support for verified software. This research was well under way when the compiler development project started. The result was the design of an abstract hardware architecture based on capabilities, called the "Capability Machine" (CM). The CM was designed to support verified software. It was also designed to be realizable through either microprogramming or hardware using existing technology. Its machine language, Capability Machine Language (CML), was designed to be a verifiable low level language. [Hoch 78]

One early design decision was to use CML as the intermediate language for the Gypsy compiler. The compiler was implemented in two separate stages: a CML Generator, which translated Gypsy into CML; and a Code Generator, which translated CML into target machine language. There were several reasons for this:

1. It would demonstrate the compilability of Gypsy into CML and also support the specification of the semantics of CML operations.
2. The semantic specifications of CML operations would then serve as specifications for target machine language generated by the Code Generator. This was expected to ease the design of the Code Generator.
3. The CML image of a Gypsy program would be a target machine independent representation of a program, reduced to relatively low level operations.
4. Since CML was designed to be target machine independent, it would be possible to implement Gypsy for different target machines by writing only the Code Generator stages.

The use of CML as an intermediate language was a limited success. It did demonstrate that Gypsy could be translated into a machine language which was designed for the purpose. And it did support rapid implementation of the CML Generator stage of the compiler. Neither result was surprising since CML was designed essentially as a transliteration of Gypsy with some reordering of operators.

There were some problems caused in the overall compiler project because of the nature of CML. First, the CML image of a program was deliberately self-contained. The CML Generator and the Code Generator had redundant but unsharable code for maintaining descriptors and other tables. Second, partially because the CML image of a program was self-contained, the Code Generator was designed to be independent of the Gypsy verification system. In practice this meant that the Code Generator lost access to the extensive cross reference information and program tree available in the verification system. Third, while CML was designed to be microprogrammable with existing technology, the CM is substantially different from existing machine architectures. This requires extensive software simulation of CM semantics, either through support packages or through large blocks of generated code. Both kinds of simulation occurred in Gypsy . Fourth, CML does not support optimization well. Since CML is straight line assembly language the program tree constructed by the verification system is lost and would have to be reconstructed. And since CML is target machine independent, none of the arithmetic involved in calculating offsets into aggregate objects appears in CML, so none of those operations would be considered during CML optimization.

# 3. Verified Support.

## 3.1 Self Support.

### 3.1-A Implementing Support Routines in Gypsy

One goal of the Gypsy compiler project was to minimize the amount of unverified support code necessary to run verified code. That is, we wanted to minimize the amount of code which was not itself the compiled image of Gypsy source code. There are two polarized approaches to this goal: to generate strictly self-contained code, or to use support code written and verified in a high level language. We chose the latter approach for several reasons:

1. Self contained code would include blocks of code which were hand written and then generated into the compiled image. This would just put the support code into the compiled code.
2. Formally specified support code would give formally specified interfaces to system resources.
3. It would be easier and faster to implement support code as high level language routines than by generating the code within the compiler.
4. It would provide another test situation for evaluating Gypsy as a general purpose programming language.

The support modules which were implemented in self support included dynamic storage management, message buffer operations, and concurrent process management.

It was necessary to extensively hand patch some of the support code written in Gypsy and compiled. All of the support modules needed to do some physical address processing, and there are no facilities for doing that in Gypsy. All of the decision logic was written in Gypsy. At points where address processing was needed we wrote function calls in the Gypsy source code, and in the compiled image of the support code we replaced these function calls with inline handwritten code to do the desired operations. (We adopted a voluntary convention not to do manual optimizing while we were making hand patches.) We provided headers for these support functions in the source code with specifications describing their effects in terms of user data. As a special case we simulated dereferencing of pointers by writing transfer functions for which we replaced the calls with one level of pointer following.

### 3.1-B Evaluation

Self support in Gypsy met several of its design goals. It did put policy decision logic into high level code, so no policy decisions were made by handwritten code. It did cut down on the amount of handwritten support code. And it did force us to specify exactly what the self support routines did.

There are several things wrong with self support as it was implemented. The most noticable is that the self support code is of poor quality. The first generation compiler spends little effort on optimization. The resulting self support code is both large and slow, which is particularly costly because several of the most frequently used functions are implemented in self support. It also inflates the amount of support code attributable to self support.

The self support effort demonstrated the need for additional facilities in Gypsy if it is to be used for general 'system' programming. There are no facilities for referencing, dereferencing, or address calculations. Such facilities would have to be restricted in order to retain axiomatizability, but they would have to be present. There are also no facilities for overloading. Overloaded support routines are necessary to implement operations defined on families of types defined by the same type constructor, such as buffer handling operations. An artificial "anybuffer" type was used to define access to the common components of buffers. This worked in Gypsy self-support only because the physical configuration of the common components is the same in all Gypsy buffer objects.

The handpatching required in Gypsy self support caused some of the design goals to be missed. The handpatching was supposed to be well-regulated, but in practice it was not. The large number of places which required hand patching meant that patching was tedious and error prone. Since these patches were being done in code generated by a compiler, there were no inline comments to guide the hand patching process. This made both the original patching and subsequent maintenance of patched code more difficult. The overall effort required to write, compile, and hand patch support routines was greater than what it would have taken to write them in assembly language originally. And the amount of hand patching necessary is so large that the goal of compiler-generated support code was essentially missed.

An alternative method of writing Gypsy support was much more productive. The support code was written in Gypsy as a pseudo-language. It was then recoded by hand into assembly language using the control and data flow of the original Gypsy, and including the Gypsy code as comments. The Gypsy entry and exit specifications are included as comments, as well as more low-level specifications such as locations of parameters, global objects used and modified, and register usage. The intent was to duplicate what an optimizing Gypsy compiler would have done if Gypsy had well regulated facilities for reference handling and overloading. This method loses the stamp of authenticity given by genuine machine compiling the self support, but it avoided many of the problems discussed above.

### 3.2 Predefined Modules.

Writing application programs in Gypsy demonstrated the need for high level language images of machine dependent hardware features. These included shifting and masking operations to process objects whose fields were defined by position within bit streams extending across storage unit boundaries. They also included I/O operations and access to the clock. Two possibilities were considered:

1. Introduce into the language new syntax classes of objects for the specific purpose of representation and control of hardware features.
2. Provide for entities of existing syntax classes to be bound to hardware features.

After much discussion and experimentation the latter approach was chosen. A Gypsy program is now provided with several predefined entities which have conventional Gypsy semantic properties but are associated with

hardware features. There are two kinds of predefined entities: I/0 buffers and data abstractions. These entities are defined in modules which are exported to programs which require access to those hardware features.

Predefined buffers are introduced into a program as actual parameters of the top level procedure. By convention the top level procedure must be named "Main". A predefined header for Main is provided in a module representing the machine environment. The formal parameters of Main are all buffers whose types are defined in the machine environment module. When Main is invoked to run the Gypsy program the actual parameters are the predefined I/O buffers. Buffer support operations detect predefined buffers and process them separately. Sending to a predefined output buffer results in queueing the transmitted object on a physical I/O buffer queue. Sending to a predefined input buffer is done by interrupt-driven routines which call the buffer self-support routines. Whether or not a buffer is predefined is transparent to support routines which receive from buffers and to all user routines.

A storage unit abstraction is a module containing a type definition for a storage unit (such as "byte" or "word") and specifications for operations on them. The operations include field insertion and extraction, and transfer functions between fields and primitive simple types. The code generator recognizes calls on these functions and provides the desired bit-manipulation code. The clock is also treated as a data abstraction module containing specifications for clock related operations such as putting the program to sleep for a specified time. The code generator recognizes calls on those operations and produces calls on appropriate clock support routines instead.

Predefined buffers transmit objects whose types are defined as aggregates of abstract storage units. This allows compiled Gypsy routines to directly process blocks transmitted through physical I/O devices.

## 4. Routine Call Expansion.

### 4.1 Necessity.

In code generated by early versions of the compiler it was found that over thirty percent of the instructions generated were from function and procedure call sites. There were three causes for this phenomenon:

1. Semantics of Gypsy routine calls. Gypsy routines may not access global variables, and the language definition requires the passing of condition parameters as well as data parameters, so Gypsy parameter lists tend to be long. In addition, actual parameters do not need to be of the same type as formal parameters. They only need to be compatible, which means that the type structures must be the same but scalar components of actual parameters' types only need to have a nonempty intersection with the corresponding component types of the formal parameters. So routine call sites must contain code to check the values of such components at execution time.

2. Gypsy programming style. Verified programs use two techniques which make inline expansion particularly valuable: data abstractions and single-call procedures. Data abstraction accessing functions are typically short, often having less code compiled from the routine body than from a routine call. Verification is frequently facilitated by writing major steps as procedures whose entry and exit specifications coincide with intermediate assertions. These procedures are only called in one place, so expanding them at that call site can save generation of routine calling linkage.

3. Inefficient call site implementation. All context and register contents saving was expanded inline at the call site instead of in the called routine.

Streamlining call sites was one approach to reducing the volume of call site code. But the Gypsy call site semantics and programming style noted above would still cause compiled Gypsy code to contain relatively frequent and relatively large routine call sites. So inline expansion of routine calls was also implemented.

### 4.2 Evaluation.

Inline expansion is not automated in the first generation compiler. The user supplies a list of routine names to be expanded. It would be possible to decide whether to expand or not by comparing the sizes of compiled routine bodies with the total size of all calls on them. This was not done because the user supplies his list to a routine in the (Lisp) GVE, and the size is not known until the (Pascal) code generator finishes.

The actual gain from doing inline expansion is difficult to assess since we currently have a very small sample size. We did observe 6 to 10 percent code compression on two examples of approximately 7K to 8K words of code.

## 5. Concurrency.

### 5.1 Gypsy Concurrent Control structures.

Gypsy contains two concurrent control structures: the cobegin and await statements. A cobegin creates a set of concurrent subprocesses, each in the form of a procedure invocation. The parent process is suspended until all of the concurrent offspring have terminated. Cobegin statements are intended to support decomposition into concurrent modules, with shared actual parameter buffers for intermodule communication. An await is a guarded selection command whose guards are message buffer operations. The process executing an await is suspended until one of its guard operations can be completed, either by sending to a nonfull buffer or receiving from a nonempty one. Await statements are intended to support I/O and interprocess message handling.

### 5.2 Implementation of Concurrent Subprocess Managment.

Primitive operations for activation and suspension are implemented in handwritten assembly language support. Activation has one parameter, a subprocess activation record, and is executed in the parent process environment. Suspension has no parameters, passing control back to the activating process. Matching activate-suspend and suspend-activate pairs are semantically no-ops to the Gypsy program except for changes in the contents of buffer parameters of the processes involved.

An abstract type "activation record" was defined in Gypsy so that process management information fields coincide with words in the header of actual activation records. A ready list is defined as a sequence of activation records. The selection routine for scheduling was specified as a Gypsy function on ready lists. It returns the index of the selected activation record or else raises a "none selected" exception condition. There is currently a single round-robin selector function, written in Gypsy, with its compiled image included in the support package unchanged except for its entry point label.

The compiled image of a cobegin creates and initializes a ready list of activation records for its subprocesses. It then enters a loop which alternately calls the selector routine and activates the selected process. This selection-activation loop terminates when all subprocesses on its ready list have terminated.

### 5.3 Exception Conditions Raised by Subprocesses.

The implementation of concurrent process management focused attention on the semantics of exception conditions raised by concurrent processes. A cobegin can terminate with either zero, one, or more than one exception conditions signaled by its subprocesses. The semantic definition of Gypsy 2.0 did not specify what happens if more than one subprocess of a cobegin signals an exception condition, so reasonable but arbitrary interpretations were implemented in the compiler:

1. If one or more subprocesses raise exception conditions but they all raise the same actual condition in the parent process environment, then that condition is raised in the parent process after all subprocesses have terminated.
2. If two or more subprocesses raise exceptions and raise different actual conditions in the parent environment then a "multiple condition" condition is raised in the parent process after all subprocesses have terminated. The multiple condition cannot be handled by any explicit condition handler. It can only be handled by an "else" arm in the exception condition handling part of a cobegin.

### 5.4 Localized and Nonpreemptive Process Management.

Concurrent process management techniques traditionally use a global preempting scheduler to allocate time slices among those processes which are then ready to execute. If a process is still executing at the end of its allocated time slice it is interrupted and the scheduler is invoked again to allocate the next time slice. The first generation Gypsy compiler implements localized nonpreemptive process management instead. The selection function is invoked within the local environment of the parent process and selection is done strictly among the processes spawned by that cobegin. If there are two sibling processes each executing a cobegin then each cobegin has its own ready list and selection-activation loop. Once selected and activated a process runs until it either terminates, becomes blocked for a buffer operation, or voluntarily suspends itself.

Localized nonpreemptive process management assumes that processes are mutually cooperative. No process holds the processor indefinitely by failing to communicate with its siblings and thus never being suspended. Priorities among processes must be built into the program by having low priority processes periodically receive "permission to proceed" through a buffer from a higher priority process, or by some equivalent means.

Localized nonpreemptive process management does not support active polling loops on user-defined buffers if the number of processors available is not greater than the largest number of concurrent active polling loops possible. The Gypsy await statement provides a well regulated way to achieve the same results in a cooperative manner. Localized nonpreemptive process management does not support concurrent execution of mutually competitive processes within one Gypsy program. It assumes that any program involving concurrent subprocesses should be specified and verified to be mutually cooperative.

Processes which are not mutually cooperative can circumvent localized nonpreemptive process management in two ways: one process can seize the processor and hold it away from that process’s siblings, or a set of processes in one cobegin can seize the processor collectively and hold it away from their "cousin" processes in other cobegins. Two measures were tried in the first generation Gypsy compiler to counteract such processor seizing:

1. The compiler can generate a suspension in the compiled image of every loop. This would prevent one process from holding the processor indefinitely.
2. The selection-activation loop can suspend itself periodically. This would prevent the set of processes in one cobegin from holding the processor indefinitely.

The selection-activation loop implements round robin selection, and the loop suspends itself after a fixed number of turns around its ready list. Suspending every loop would incur excessive overhead. So the compiler

inserts suspensions only into loops which test whether any buffer is full or empty. This was intended to prevent active polling loops from seizing the processor while avoiding the overhead of suspending all loops. It could be circumvented by a loop which called procedures which in turn did the polling of buffers.

### 5.5 The Semantics of Concurrency.

The use of localized nonpreemptive process management proved to be controversial. Cobegins were implemented before awaits, and in the absence of awaits, programmers wrote active polling loops over sets of user-defined buffers to simulate awaits. These active polling loops executed as infinite loops which seized the processor. This was unexpected but minimally consistent with the formal semantics of the Gypsy cobegin, which states only that sibling processes should run "concurrently" but does not specify the allocation of processors among processes.

There was disagreement over whether the semantics of Gypsy should implicitly require global preemptive scheduling of concurrent subprocesses. The language designers had assumed that processes would be mutually uncooperative, and that a global preempting scheduler would enforce an undefined "fair" means of allocating time slices among processes. That assumption was made before there was a body of experience in specifying program performance in terms of message buffer traffic, and without consideration of the difficulty of specifying "fair" scheduling.

The two methods of counteracting the seizure of processors described in the previous section were implemented to provide an environment in which mutually uncooperative source programs could be executed. They would enforce the letter but not the spirit of mutual cooperation.

A central issue in the definition of the semantics of concurrency is how to deal with mutually uncooperative processes. There are (at least) three positions possible:

1. Assume that concurrent processes in general may be mutually uncooperative, and require preemptive global scheduling among them. This has the apparent advantage of supporting familiar programming styles. There are two disadvantages: first, there is a lot of overhead associated with preemption at unpredictable and irreproducible times; second, it removes process priority and processor allocation from user specifiability and control.

2. Assume that a program written with mutually uncooperative processes is "erroneous". If a program is written in such a way that one of its processes can seize the only processor, then it is semantically correct to execute the program that way. Program specifications should include mutual cooperation among the program’s concurrent processes. There are three disadvantages to this position: first, it requires the definition and development of techniques for efficient implementation and use of mutually cooperative processes; second, it may incur substantial overhead in interprocess message handling; and third, it may make programs installation dependent in the sense that the programmer must know whether there are fewer processors than processes.

3. Assume that there may be mutually uncooperative processes but that global preemptive scheduling is not required. Have the compiler generate code which is mutually cooperative, which is possible if all executing code is generated by a compiler. This position has the disadvantage that it requires very high overhead since all loops must include suspensions. Gypsy processes can have no knowledge of whether they have sibling processes, so the compiler cannot generate suspensions in only those loops which are part of concurrent processes.

The semantics of mutually uncooperative concurrent processes in Gypsy has not been resolved. The first generation Gypsy compiler took the second and then the third of the above positions.

# 6. Data Structure Representation.

## 6.1 Activation Records for Dynamic Processes.

Both dynamic storage management and concurrent process management were implemented in self-support. As described above in section 3, ready lists were implemented as Gypsy sequences of activation records. This had the expected advantages of high-level language implementation: concurrency was relatively easy to implement using Gypsy sequence accessing operations, and it was consistent with the goal of writing as much of the support as possible in Gypsy itself.

It was immediately discovered that prototype versions of programs intended for verification frequently experienced severe memory fragmentation, while "toy" programs intended to test concurrency and dynamic memory management did not. Analysis of memory dumps revealed that the following chain of events led to the fragmentation:

1. The Gypsy cobegin provides fully dynamic creation and destruction of concurrent subprocesses. There is no limit on the number of existing processes or on the levels at which they may be created. Programmers frequently wrote programs whose natural organization called for several modules, each with several concurrent subprocesses. These were easily implemented with several levels of cobegin statements.
2. Activation record segments for concurrent processes were allocated in the same manner as elements of dynamic data objects.
3. In the initial stages of execution, programs allocated a mixture of activation record base segments and process synchronization messages. The activation record segments were large and had long lives, and the buffer messages were small with short lives. This resulted in spreading the activation record base segments across large amounts of memory.
4. In later stages of execution, programs began going deeply into levels of routine calls to process messages. This created a demand for large activation record segments with short lives. But the initial spreading of the base segments had severely limited the number of large contiguous blocks available. Programs often ran out of blocks large enough for activation record segments while there was still a sufficient volume of small but noncontiguous blocks available.

Allocating smaller segments for deep routine call local storage did not help because it introduced higher space overhead in the activation records. Programmers devised schemes to get around the fragmentation problem, but these did not follow natural program decompositions. Two observations made from the scenarios leading to fragmentation are being implemented in the second generation Gypsy compiler:

1. Activation record segments should not be allocated simply as large dynamic objects. They should probably be allocated from a separate area dedicated to large objects with long lives.
2. The implementation of concurrent subprocesses does not need to provide full dynamic creation and destruction. Many programs have several levels of cobegins, but the number of concurrent processes at steady state can be statically determined. Activation records for these processes could all be allocated at program initialization.

It was originally anticipated that implementing dynamic storage management in self-support would make it relatively easy to implement alternative allocation strategies. The system currently uses first fit for all storage allocation. Because of the difficulties encountered in the self support implementation of dynamic storage management we did not attempt to implement other methods.

### 6.2 Unit Sequences as Pointers.

Gypsy does not have pointer types. It does have dynamic types. But Gypsy type definitions must be statically nonrecursive, so Gypsy does not support the construction of linked data structures.

Programmers soon discovered that they could simulate the semantics of pointers with "unit sequences," sequences with a defined maximum length of 1. A null pointer is simulated by an empty sequence; a nonnull pointer by a nonempty sequence. Unit sequences were particularly inefficient in the internal representation used for sequences. That consists of contiguous blocks of pointers to sequence elements, allocated in groups of sixteen. Space was inefficiently used by unit sequences, and code size was unnecessarily large because the compiled code expected to operate on the general sequence representation.

When it was discovered that programmers were using unit sequences as pointers, the compiler was revised to treat unit sequences as separate type constructors which were incompatible with general sequences. Unit sequences were then compiled as pointers. It was still necessary to generate code to check for exception conditions involving sequences, but these all involved testing for either null or non-null pointers.

Representing pointers as unit sequences has a desirable side effect. To prevent generation of aliases, pointer assignment must be destructive. (This is achieved with unit sequences in Gypsy by implementing pointer assignment as sequence insertion, and immediately assigning an empty sequence to the previous right-side sequence.) Dangling references are caused by assigning reference values to pointers whose current values are not null. With pointers implemented as unit sequences, dangling references would be caused by insertion into a full unit sequence. Thus dangling references can be detected as a special case of the sequence error condition.

### 6.3 Execution Time Descriptors.

The code generator was initially designed as an interpreter for the intermediate language CML. It would generate code only for operations which it could not complete at compilation time. The intent of this design was twofold: to make Gypsy semantics explicit in the logic of the compiler itself, and to minimize the amount of code actually generated. As a consequence of this decision the compiler uses compile-time descriptors for all non-dynamic data objects. Parameterized types were not implemented in the first generation compiler.

Use of strictly compile-time descriptors proved very costly in assignments and formal-actual parameter bindings involving dynamic objects whose types were compatible but not identical. Those situations occur frequently in Gypsy programs. The compiler generates inline code to access and check all range-restricted components in which the destination field’s range is not a subset of that of the source field. On one occasion the compiler generated 4K words of code for an assignment statement!

The immediate cause of such spectacularly large code was found to be excessively complicated dynamic type definitions used by the programmers. (The 4K assignment statement involved a thirteen level structure of nested sequences.) But even with simplified types there was a lot of code generated for inline consistency checking. Gypsy is probably characteristic in requiring types to be compatible but not identical. It may be desirable to use execution time descriptors for some type consistency checking by support routines, even when complete descriptors are available to the compiler for generation of inline code.

# 7. The Compiler Role in Program Development.

## 7.1 The Compiler as a Program Development Tool.

During the initial design of the Gypsy Verification Environment it was assumed that compilation was the exit path out of the software development cycle. Once a program was verified it was ready for compilation and operation. Under this assumption the compiler was not considered to be a tool for ongoing program development in the same sense as the other GVE components, and it did not need to be integrated into the rest of the GVE. There were two significant design consequences of this assumption:

1. The compiler was developed separately from the GVE. It did not use the GVE facilities for cross referencing, data base information, or expression evaluation and simplification. (This assumed separation was one reason why the first generation compiler was written in Pascal while the rest of GVE is in Lisp.)
2. No provision was made for separate compilation of partial program units, and no attempt was made to integrate "compiled" code into the GVE database. It was planned to use the compiler only to compile entire programs which were already verified.

After considerable experience with the compiler and the rest of GVE it became evident that the compiler should be viewed as a step within the program development cycle. Verification shows only consistency between formal specifications and code. It does not show that the formal specifications really are consistent with the program's operating environment. Execution of parts of partially developed programs serves to test the formal specifications. Programmers may also want to test code before verifying it. Verification is still time-consuming and difficult, and the interpreter in the GVE can not yet simulate I/O through predefined buffers.

The verification system of the GVE maintains extensive cross referencing information and a program structure tree. GVE also has extensive facilities for expression evaluation and simplification. These could be used by the compiler to do extensive constant folding and code simplification before code generation. The second generation compiler is being implemented in Lisp, as part of the GVE, and operating from the overall GVE database.

## 7.2 Specification-driven Optimization.

The compiler was designed with two optional optimization stages, one on intermediate code (CML) and one on object code. Optimization was put off until after the CML Generator and Code Generator were running. That gave us the opportunity to examine code generated by strictly syntax-directed means with no optimization. A striking feature of that code is the amount of code generated for checking range restrictions on values of subscript expressions and values assigned to range-restricted variables.

If it could be determined from assertions and expression evaluation that values of particular subscript or other expressions would always be in the required ranges, and if the program including those assertions had been verified, then the compiler could safely avoid generating code to do the checking. Programs which are verified by inductive assertions can be expected to be rich in assertions involving bounds on the values of subscript and other expressions. From these observations we came to two conclusions:

1. The compiler should be implemented within the GVE, with access to assertions in programs and to the expression simplification and evaluation facilities used by the verification system.
2. Research should begin immediately on the use of verified assertions in particular, and specifications in general, to direct program optimization.

The second generation Gypsy compiler is being integrated into the GVE, and work has begun within CMP on specification-driven optimization.

# 8. Methodology Demonstration.

## 8.1 The Gypsy Methodology.

One of the products of CMP is a software development methodology based on formal specification and verification in Gypsy. The methodology requires the use of programming techniques which facilitate formal verification - in particular, top down modular design and careful use of data and procedural abstractions. Successful verification depends on construction of modules with clean boundaries. Thus the "Gypsy methodology" is essentially a form of modular top down design employing ease of formal specification and verification to direct the definition of module boundaries.

There were several reasons for using the Gypsy methodology in writing the Gypsy compiler. First, it would serve as a testbed for the methodology, and it would be a documented demonstration that the methodology is practical for the development of large software systems. Second, the constraints on our resources required that we use a methodology which would deliver reliable, running software in a relatively short time using relatively little manpower. Third, while we did not expect to formally specify and verify the entire compiler, we could employ the design methods and programming techniques with the expectation of verifying at least part of the compiler.

## 8.2 Implementation Language.

Despite the decision to use the Gypsy methodology, the first generation compiler was not written in Gypsy. The initial target machine was a PDP-11, but a Dec-10 (and later Dec-20) was available for implementing a cross-compiler. We considered two possible choices for implementation language:

1. We could specify and write the entire compiler in Gypsy and then translate it into an implementation language. Lisp was the obvious candidate since the verification system was being developed in Lisp.
2. Gypsy is a Pascal derivative. We could write the compiler in Pascal, avoiding Pascal features which had been eliminated in Gypsy, and simulating Gypsy features which were not in Pascal.

We chose the latter approach for two reasons. First, we felt at that time that the compiler could be logically separated from the verification system, so it was not critical that it be implemented in Lisp as part of the overall verification environment. Second, Pascal was considered a more appropriate language for compiler development than Lisp.

The implementation language is Pascal with several self imposed programming conventions. The intent of these conventions was to work in the intersection of Pascal and Gypsy to the extent that this was well understood. The conventions used were:

1. No global variables were to be used. All information was to be passed in parameter lists.
2. Only the high level control structures were to be used.
3. All major data objects such as tables and descriptors were to be implemented as data abstractions. All access to them was to be via access functions and procedures. Abstract objects themselves as aggregate objects were passed as parameters.
4. Pointers were not used. Gypsy has several dynamic type constructors but they do not appear in Pascal, so we used static data structures whose sizes could be tuned.
5. Every procedure was to begin with an extended specification comment. This comment gave the entry and exit specifications in formal Gypsy notation. The comment was to include a complete dated trace of changes in the specifications and the code.

In practice the language proved to be readily usable. The conventions noted above are guidelines to good

programming practice rather than restrictions. We did find the absence of both pointers and dynamic structures to be inconvenient, but this was a result of not setting up a convention to allow pointers in a regulated manner to simulate dynamic structures.

The compiler is not very efficient as a Pascal program since Pascal is not oriented toward the style of programming which facilitates verification. Specifically, the code is more dense in calls on small data access routines than traditional code. As a concession to the Pascal convention of passing non-var parameters as call by value, we passed all data objects as var parameters, even when they were intended for value extraction only.

### 8.3 Extent of the Methodology Demonstration.

The Code Generator stage of the compiler served as the methodology demonstration. The conventions were followed strictly through voluntary compliance, and much of the design and coding of the intermediate language (CML) generator part was finished before the conventions were adopted.

The Code Generator contains about 12000 lines of Pascal source code including comment lines. We do not have a separate count of non-comment lines but we estimate that approximately 30 percent of the source code lines are comments. In some modules about half of the source lines are comment lines, especially in the later modules written after the value of the methodology conventions was becoming evident. Most of the comment lines come from the specification comments in each routine. This is consistent with the observation that in verified Gypsy code about half of the code is specifications.

### 8.4 Overall Successful Evaluation.

The methodology demonstration was objectively successful. The result is a large program which was delivered with a relatively small cost in manpower; it has proved to be maintainable, and it has been transported. The Code Generator is a relatively small task as a compiler since it does not include intermediate code generation, syntax error handling, or much lexical analysis. But the entire Gypsy compiler including the separate intermediate code (CML) generation stage required approximately four and one-half man-years, a low figure for a compiler. The compiler was moved from a Dec-10 (Tenex) to a Dec-20 (Tops-20) with a different Pascal compiler for a total cost of about one man-week. The modularity enforced by the methodology had isolated system-dependent I/O operations for easy modification.

The methodology demonstration was also subjectively successful. The formal specification comments served as a framework for code structure, a phenomenon already known. Maintenance and modification was relatively easy because the formal specifications and the highly modularized style of programming made it relatively easy for programmers to analyze and modify code - both their own and each other’s.

We were especially impressed by the usefulness of formal specifications as maintenance documentation. We were also impressed by the difficulty of writing specifications well. Our immediate goal was to write specification comments which would be easy to code to and verify. We found that experience in writing verifiable specifications made it easy to read each other’s specifications for doing maintenance and modification.

### 8.5 Problems Encountered.

The simulation of a Gypsy program development environment was incomplete and inefficient. Since the methodology conventions were not enforced, there were occasional lapses in their use, usually involving the use of global data objects. Since the formal specifications were only comments they were occasionally only informal; this caused some problems in debugging when it was assumed that the informal and incomplete specifications were formal and complete. As noted above the result is not efficient as a Pascal program since Pascal is not oriented toward verified programming.

Some inefficiency and programming difficulty was caused by poorly designed data abstractions, especially for assembly language output. This was due to initial inexperience on the part of the programmers at designing data abstractions. It was also due to the lack of an automatic program development system which would isolate the effects of subsequently proposed changes in a data abstraction. It is important to note that this was a problem of programmer inexperience and support, and not of the methodology itself.

## 9. Summary.

The first generation Gypsy compiler project showed that it was possible to write and compile nontrivial, useful verified software. In fact, much of the support code was written in Gypsy itself. The compiler project also showed that compilation of verified software is different from compilation of more conventional code. In some cases this was unexpected, as in the apparent desirability of run-time descriptors for dynamic objects for which complete compile-time descriptors could be constructed. In other cases it was traceable to characteristics of verifiable code, as in the critical need for compact routine call sites.

Working on the compiler also gave us a better understanding of the semantics of concurrent invocation of subprocesses. We were forced to address the details of concurrent exception condition handling. And the implementation of process management in self-support revealed subtleties in trying to rigorously specify the behavior of a "scheduler".

The compiler also provided a laboratory for the analysis of verifiable code as it is actually written for execution. While our sample is small and our experience limited, it appears that programmers do not utilize the full generality of all Gypsy features which is necessary for axiomatizability. Thus a compiler could impose restrictions which would not really restrict the usability of the language. In particular, if unit sequences are not compatible with more general sequences, then they can easily be represented by pointers. And if static cobegin structure is required, a simpler and more efficient means of activation record management can be implemented.

Finally, the first generation Gypsy compiler revealed the need for a second generation compiler which was an integral part of the programming environment instead of a logically separated exit path. This would allow the compiler access to extensive and useful facilities which are already necessary for verification, such as simplification and partial expression evaluation. It would also open the way for the use of formal specifications to direct optimization efforts, an area of great promise.

)

[Good 78] Good, D.I., R.M. Cohen, C.G. Hoch, L.W. Hunter, D. F. Hare, "Report on the Language Gypsy: Version 2.0." ICSCA-CMP-10, September 1978.

[Hoch 78] Hoch, C. G. "Hardware Support for Modern Software Concepts." Ph. D. Thesis, August 1978; ICSCA-CMP-12.

# Table of Contents


1. INTRODUCTION . . . . . 2
  1.1. The Gypsy Language and CMP. . . . . 2
  1.2. The Gypsy Compiler Project. . . . . 2
2. Verifiable Intermediate Language. . . . . 3
3. Verified Support. . . . . 4
  3.1. Self Support. . . . . 4
    3.1-A. Implementing Support Routines in Gypsy . . . . . 4
    3.1-B. Evaluation . . . . . 5
  3.2. Predefined Modules. . . . . 5
4. Routine Call Expansion. . . . . 6
  4.1. Necessity. . . . . 6
  4.2. Evaluation. . . . . 7
5. Concurrency. . . . . 7
  5.1. Gypsy Concurrent Control structures. . . . . 7
  5.2. Implementation of Concurrent Subprocess Managment. . . . . 7
  5.3. Exception Conditions Raised by Subprocesses. . . . . 8
  5.4. Localized and Nonpreemptive Process Management. . . . . 8
  5.5. The Semantics of Concurrency. . . . . 9
6. Data Structure Representation. . . . . 10
  6.1. Activation Records for Dynamic Processes. . . . . 10
  6.2. Unit Sequences as Pointers. . . . . 11
  6.3. Execution Time Descriptors. . . . . 11
7. The Compiler Role in Program Development. . . . . 12
  7.1. The Compiler as a Program Development Tool. . . . . 12
  7.2. Specification-driven Optimization. . . . . 12
8. Methodology Demonstration. . . . . 13
  8.1. The Gypsy Methodology. . . . . 13
  8.2. Implementation Language. . . . . 13
  8.3. Extent of the Methodology Demonstration. . . . . 14
  8.4. Overall Successful Evaluation. . . . . 14
  8.5. Problems Encountered. . . . . 15
9. Summary. . . . . 15