Grammar-based
Rapid Application Development
(GRAD)

Charlie Fly

United Space Alliance
600 Gemini Ave.
Houston, Texas 77058

Abstract

The *Grammar-based Rapid Application Development* (GRAD) system is an interface generator between Python and legacy languages. To allow the interface generator to work with multiple languages, the interface is based upon a grammar for a given language. Currently, the C++ language has had its interface compiler created and successfully emits the necessary binding code to make C++ code accessible from Python.

The GRAD process is non-intrusive so the legacy code does not have to be modified in order for the interface to work. No superclasses or include files are added to the existing code and changes in the interface generator system itself do not require changes in the legacy code. Changes in the target legacy code naturally requires the regeneration of the interface but no manual editing of the generated interface is needed, so the interface generation can be made part of the build process.

Using GRAD, the promise of entire new applications written rapidly in the high level language Python is attainable while still preserving much of the operational advantages of, and investments in, existing compiled and tested code.

SECTION 1 - INTRODUCTION

1.1 I/O and Control Schemes

Building large, complex software to model simulations runs the risk of creating monolithic programs which are very hard to tailor once established and tested. Great effort is placed on getting the system to work fast and properly, pushing considerations such as flexibility and generality onto the back burner. Very often little attention is paid to the techniques for getting data into and out of the application. Individual programmers will usually just adopt whatever approach seems most direct and expedient at the time. Unfortunately, this tends to result in software systems which have tight coupling between its core functionality and its I/O needs. Once built, such systems do not lend themselves to changes in the control mechanisms or data visibility. If the user wishes to explore the effects of particular parameter or algorithm changes she must often resort to software changes in the native implementation language. This is inefficient as it requires the user to understand many details of the implementation at a low level, and moreover it voids the validity of whatever testing was done on the baseline software.

Methods to address this problem has been dealt with in numerous computer science texts. Most revolve around good practices such as object oriented design or modularization of code. The GRAD developers believe decoupling I/O and overall control logic from the core functionality reduces the likelihood of unbounded complexity growth while preserving operational flexibility needed for adapting to new requirements.

1.2 Early Executive Attempts

In 1992 a project embarked on a plan to reengineer and develop a replacement for older software written in a patchwork manner over many years. For a number of reasons not discussed here C++ was chosen as the implimentation language. A domain design effort commenced to identify and code core reusable components which would be used to design and build modular applications. To interface directly with the C++ domain classes being written, an executive was prototyped in C++. This executive would be able to create and interrogate objects directly through pointer mechanisms. Both the public and private members could be exposed allowing not only inspection but dynamic alteration of the C++ objects at runtime. The intent was to provide a flexible control mechanism for C++ applications, but was less than successful for a number of reasons. Not only was the implementation difficult but it required yet another mini-language be developed for the user to define his control logic. This approach would only work with C++. Moreover, the wisdom of exposing all the members of a C++ object was suspect, as subtle errors can be easily introduced with such an implementation.

1.3 The Python Option

The existence of the Python language interpreter became known to us in early 1994. At first, it was seen as a fine scripting language. It was put to use in system administration scripting but it's power as an extensible object-oriented language was not used and consequently under-appreciated. The C++ Executive effort mentioned above was already underway on it's own track. It took about a year to realize that this effort wasn't going to yield useful results. At that time, early 1995, the option of using Python as the front end glue environment (executive) for evolving C++ applications was seriously explored. Python's dynamic extension facility would be key in this role.

Three members of the team attended the 2nd Python Workshop in Menlo Park, California in May 1995. Discussions there with Jim Roskind sparked an idea on how to address our C++ executive problem in a general fashion using Python. Fresh with enthusiasm from the workshop, the developers returned to Houston prepared to redo the executive project in Python. The Grammar-based Rapid Application Development (GRAD) project began shortly thereafter.

1.4 Acknowledgments

The author would like to acknowledge the contributions of his co-developers Greg Boes, Mark Guerra and Robin Friedrich. Without their insight and skill GRAD could not exist.

Discussions with Jim Roskind and his C/C++ grammars have been immensely beneficial.

Finally, the attendees of the 2nd Python Workshop showed the GRAD developers that Python was an effective means to attack the executive problem.

SECTION 2 - DESIGN DECISIONS

Based on the lessons learned from the earlier executive and the new insights gained at the workshop, the following design guidelines were adopted:

1. A true parser based on a grammar for the legacy language is required.
2. The system should work with multiple languages, so the parser had to be driven using the results of a compiler generator.
3. Only public interfaces are supported.
4. The legacy code should not be dependent upon the GRAD interface.
5. Generation of the interface should be automatic from the source code.
6. The interface generator should produce an intermediate language from which code is emitted.
7. Python becomes the application development language utilizing core software written in multiple languages.

Each of these decisions will be discussed below.

Overlying the whole executive concept is the desire to supplant application-specific Input/Output schemes where possible and reasonable. Application specific I/O schemes are seen by the GRAD developers as a major source of complexity growth in software. Due to changing requirements levied on applications after initial delivery, the coupling of I/O operations and core functionality get progressively more tangled. Without a clean division between input-output features and core domain functionality it becomes more difficult to maintain the code and even less likely to reuse the core domain components in other applications. Other applications rarely have the same input-output needs as the original program. The GRAD interface, with it's definite division of responsibilities between Python and the legacy code, is seen as a natural way to limit such complexity growth.

Finally, writing in Python is markedly faster than in C or C++. This means applications can be written and tested in less time.

The primary results of the last year's work are two programs:

- A Compiler Generator written in Python which produces state and grammar token files for an LALR(1) parser and a Python file for import containing methods to implement the reductions.
- An Interface Generator also written in Python which uses the products of the compiler generator to compile the C++ code into Python and C++ interface modules.

Thus far, an interface to C++ has been created. Grammar files have been located for C and FORTRAN, but the work of adding reductions to them has not yet started.

Just recently, the GRAD project has expanded to a joint collaboration between the United Space Alliance and the Lawrence Livermore National Labs.

2.1 A true parser based on a grammar for the legacy language is required.

GRAD arose from earlier work on an executive system for the Reusable Object Simulation Environment (ROSE). ROSE was tasked with reengineering approximately 3.5 million lines of FORTRAN code used for Space Shuttle simulation into about 600,000 lines of C++.

The early executive was coded in C++ to interface with C++ code. The developers parsed the header files to create a Schema which provided access to all parts of a class: public, private and protected. The parsing was accomplished with a set of regular expressions. This proved adequate for a subset of the ROSE classes. But when attempts were made to expand this to the full set of classes, the system became woefully complex.

This led to the decision to handle the parsing via a grammar specification for C++. Due to the avoidance of precedence and associativity in its rules, Jim Roskind's C++ grammar was selected. Also there is an associated C grammar specification that is as identical as possible to the C++ grammar specification. So much of the work done on the actions in the C++ grammar is directly portable to the C grammar.

2.2 The system should work with multiple languages.

The fundamental driver for this decision is the number of languages used in the Space Shuttle program including C, C++, FORTRAN, HAL/S and ADA. The majority of the code within the Space Shuttle program is written in FORTRAN.

Since the parser for GRAD is derived from a grammar file, it is a simple matter to switch languages by switching grammars.

GRAD produces interfaces that are independent of each other, so it is possible to run two or more languages simultaneously from Python. For example, a FORTRAN application being rehosted in C could be run from Python simultaneously with its C equivalent providing a unique validation technique.

Inter-Language Unification (ILU) from Xerox PARC can perform this same feat in a shared address space but not without modification of the native codes and addition of ISL files. We felt this level of intrusiveness was unacceptable for our dynamic application domain.

2.3 Only public interfaces are supported.

The GRAD developers have decided to only provide an interface to the public interfaces of a class or code module. While there are good reasons (debugging for example) to provide access to the protected and private components of C++ classes, it is too easy to write applications that violate the fundamental tenet of data hiding (encapsulation).

If access to private data is accomplished via a Python script such that the application writer can alter the behavior of a class in use, inverse coupling is achieved between the application and the class. While the core code has not been coupled directly, its behavior is now dependent upon external code. The core code cannot now be reused without including the external code. Over time, the problem we aim to solve, coupling, is just reinstated in another way.

2.4 The legacy code should not be dependent upon the GRAD interface.

The predecessor executive project was dependent upon inheritance from a superclass to provide the executive with access to the attributes and methods of any class. This had two implications:

1. A change to either the executive or the legacy code required a recompilation of both.
2. All legacy code that predated the executive had to be retrofitted with the code to implement the inheritance scheme.

The GRAD developers take the position that the Legacy Code should remain as pristine as possible. The GRAD approach is therefore non-intrusive.

2.5 Generation of the interface should be automatic from the source code.

This point is obvious, but not necessarily easily achieved. If it is necessary to manually create inputs for the interface generator or manually edit the generated code, then the tool is significantly less usable. GRAD's premise is that the input is source code and the output is ready to compile and link.

There are two underlying assumptions:

1. The code being processed by GRAD has been successfully compiled and linked by the system compiler.
2. A dynamic shared library of the source code can be created on the target platform.

A consequence of the first assumption is that any error in the source code detected by GRAD is not a problem with the source code, but is assumed to be a problem with GRAD.

When the interface generation is automatic and the legacy code is not dependent upon the interface, then it becomes a simple matter to regenerate the Python/Legacy code interface.

When the public interface of legacy code (e.g., public methods and attribute types in C++ code) changes, the Python scripts based upon it require changing. The changed interface can be represented as an object by GRAD that has knowledge of the previous interface and the new interface. A utility can then be implemented to search for Python scripts that use the old interface and notify the owner of the change. This has not been implemented yet, but is in the future plans for GRAD.

2.6 The interface generator should produce an intermediate language from which code is emitted.

The intermediate language is an abstract representation of the constructs found in most languages:

- Classes, Structs, Unions
- Attributes
- Methods and
- Types

The intermediate language is represented in two sets of classes:

- Metaclass and
- MetaTask

Both are abstractions of the constructs found in the legacy code. Metaclass supports some special properties allowing Metaclass objects to be simply ‘added’ to each other to support the reduction process. MetaTask is directly focused on output generation. After the source code is translated into Metaclass objects, the Metaclass objects transform themselves into MetaTask objects.

This allowed the developers to simultaneously code the parser and the code emission using separate treatments of the intermediate language as the interface.

The intermediate language maps across multiple legacy languages.

The output generation is not limited to the Python/Legacy language interface. In particular, the PATHS classes can be generated from the intermediate language. (See section 5 on page 16 for a discussion of PATHS.) These classes assist manual test generation as a minimum and, in many cases, can provide automated test generation and tracking. Interfaces to languages other than Python could also be handled, but the GRAD project has no plans to do so.

2.7 Python becomes the application development language utilizing core software written in multiple languages.

In the earlier executive efforts, a simple control language was devised to provide the end user with tools necessary to analyze intermediate results of calculations and make decisions regarding further processing. Developing such an application specific language was found to be disproportionately difficult when compared to the rest of the system.

At the same time as the early executive was being developed, Python was being explored by other engineers on the ROSE project. It's virtues as a highly flexible, extensible programming language were just being appreciated. It became obvious that Python could function very well as the glue language for future executive work.

SECTION 3 - SUPPLANTING APPLICATION SPECIFIC I/O SCHEMES

Many applications, particularly in the NASA simulation world, use some form of external data to drive the program. This data is, in effect, an equivalent representation of an internal state of the application. If the external data is altered, then the internal state will be different if the program is run with this external data. The internal state is simply the values stored in all the variables at a given point in time.

An I/O scheme, as defined here, is the part of the application which transforms between the internal and external forms. This is really only an I/O scheme if the process is bidirectional. (Otherwise it might be more correctly called an Input Scheme.)

The I/O scheme provides associativity between user specified names and internal data. To accomplish this, the I/O scheme "parses" external data creating the association and setting the data value.

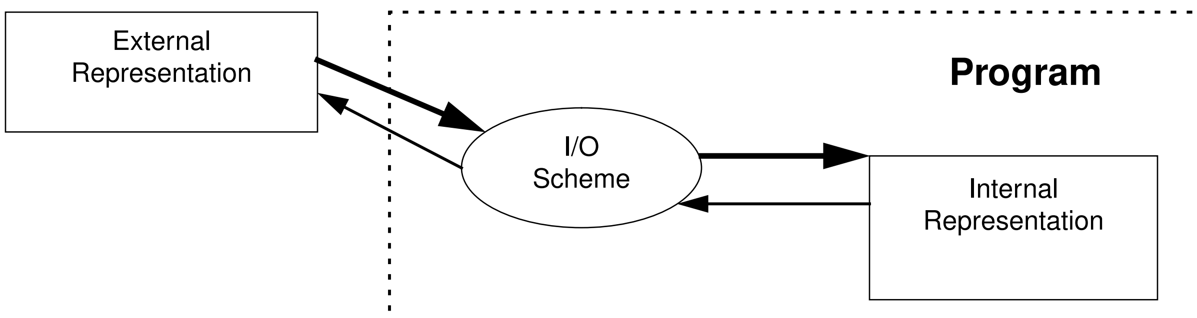

Figure 1. Simple I/O Scheme

The form of the external representation is dictated by the developer(s) of the application. The key point is that the form can be unique to each application. So each application has a unique I/O scheme. Depending upon the complexity of the application, the I/O scheme can account for a modest or a significant amount of the application code.

It is our experience that these I/O schemes are a major source of complexity growth over time. Much of our software has a history back to the early days of the space program. This makes physical sense. The algorithms to simulate and accomplish space flight do not change. Navigation is still navigation. So code written 30 years ago has been rehosted multiple times. The languages have pretty much stayed the same and the platforms have changed.

In simulations, the external representation corresponds to the sequence of internal states the program goes through during execution. If it is necessary to restart the program at one of these intermediate states, then the I/O scheme needs to output the data needed to initialize the application at the intermediate state.

The restart data is just another copy of the external representation. When fed back to the application through the I/O scheme, the application initializes to the state and continues execution. The subsequent states are a subset of the original sequence of states.

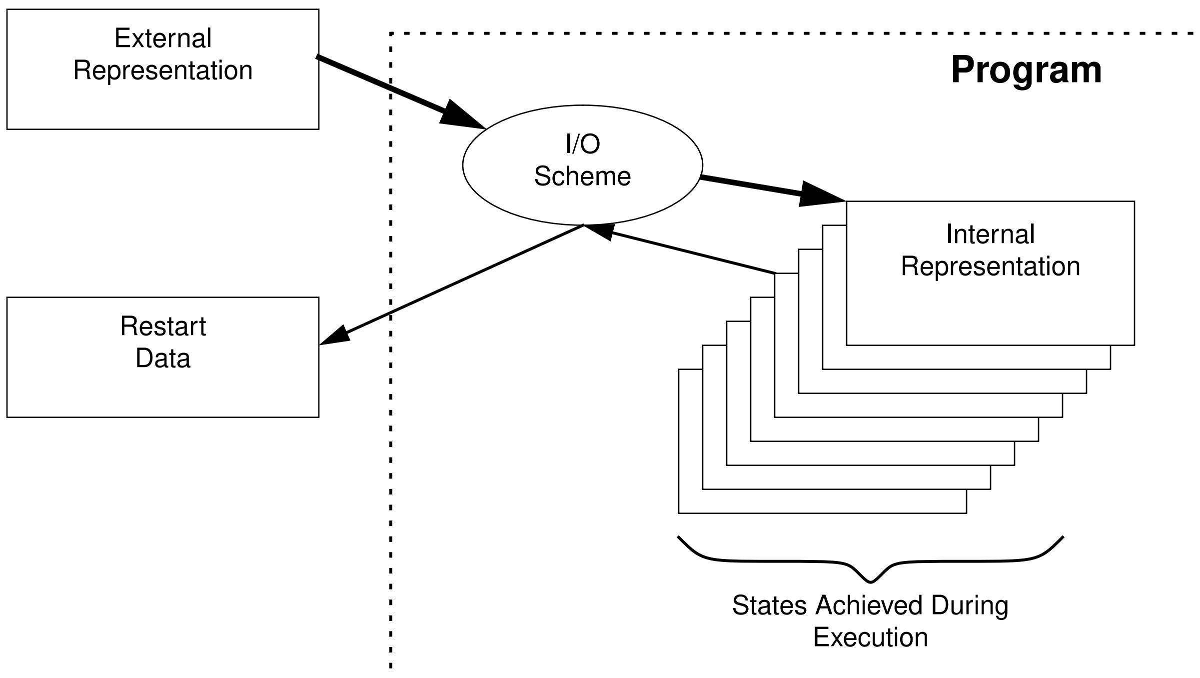

Figure 2. I/O Scheme with Restart

In and of itself, none of this would seem to be a problem. It's just part of writing a program for this environment. However, programs evolve over time. And that evolution alters one or more of the three components:

- External Representation
- I/O Scheme
- Internal Representation

None of the above can be altered without a corresponding alteration of the other two. And any of these changes will force the modification of preexisting data sets to obey the new format.

A second consequence is more insidious. In general, the evolution of a program is naturally driven by the need to make it do something it didn't do originally. So the format of the I/O scheme often doesn't conceptually support the new requirements. Hence, the I/O scheme gets reorganized so it can accommodate both the original problem and the new problem. The destination data structures for the new requirement often do not exist at the application level but at some deeper level. So the I/O scheme is extended into code that has no requirement for it other than to support the higher level enhancement. Coupling between the I/O scheme and core code modules is introduced.

Over time there is a tendency for each of these I/O schemes to evolve from a relatively simple form until they become effectively high level debuggers as each programmer adds new ways to access the internal state of the program. The I/O scheme gets embedded in the low-level code. Once the I/O scheme is embedded, the low-level code is effectively barred from reuse by another application which, almost by definition, will use a different I/O scheme. In the end there are nearly as many copies of the low-level components as there are applications.

The motivation driving this problem is not trivial. The engineers doing flight planning need to be able to get at the lower level components to isolate problems. Doing this from a conventional debugger is not really the solution either. The problems are not code problems, but data and data relationship problems.

GRAD eliminates much of this coupling because GRAD becomes the I/O scheme. It accomplishes the associativity between names and instances. The external representation are just expressed in Python statements. GRAD thus becomes a consistent I/O scheme across many applications.

While there will still be a need to write specialized I/O schemes, the number of these will decrease. Further, these will be written more generally in Python. The motivation and ability to couple the I/O scheme into the core code is dramatically reduced. A much higher level of code reuse is the result.

SECTION 4 - GRAD SYSTEM OVERVIEW

Figure 3 shows the three distinct phases of data flow within the GRAD system:

1. Compiler Generation
2. Interface Generation
3. Usage

The Compiler Generator creates the tables used by the parser in the Interface Generator.

The Interface Generator processes the Legacy Code source files creating a binding code module in the legacy language, a global Python module, a module specific Python module and a Python module for each class found in the source code.

The user imports the interface into Python either from the command line or in a Python program. Objects in the legacy language are created, manipulated and destroyed by Python statements.

4.1 The Compiler Generator

The GRAD Compiler Generator gives GRAD the ability to work with multiple languages. The GRAD Interface Generator contains a generic parser which is driven by a set of State objects and grammar token ids. These are created by the Compiler Generator.

The Compiler Generator builds an LALR(1) parser. Rather than build yet another parser generator, the GRAD Compiler Generator uses the output of GNU Bison to build a parser in Python. There was definitely a lesson learned here. Much of the early work on the parser side of the project was directed at creating a compiler generator in Python. While the Python tool worked, it was very slow. This impacted the development of the parser. Once the decision was made to build the state objects from output of bison, work sped up significantly.

The GRAD grammar file is a modified Bison grammar file. Lexical correspondence between tokens in the rules and the actual character(s) (e.g. SEMICOLON and ';') is provided within specialized comments. The actions are written in Python.

The Bison output is used to create a list of State objects. Each state contains shifts, reductions and gotos which provide the parsing action. The actions from the grammar rules are translated into a Python file that is imported by the GRAD parser. Since the actions are Python code, it is much easier to debug them while the parser is being developed.

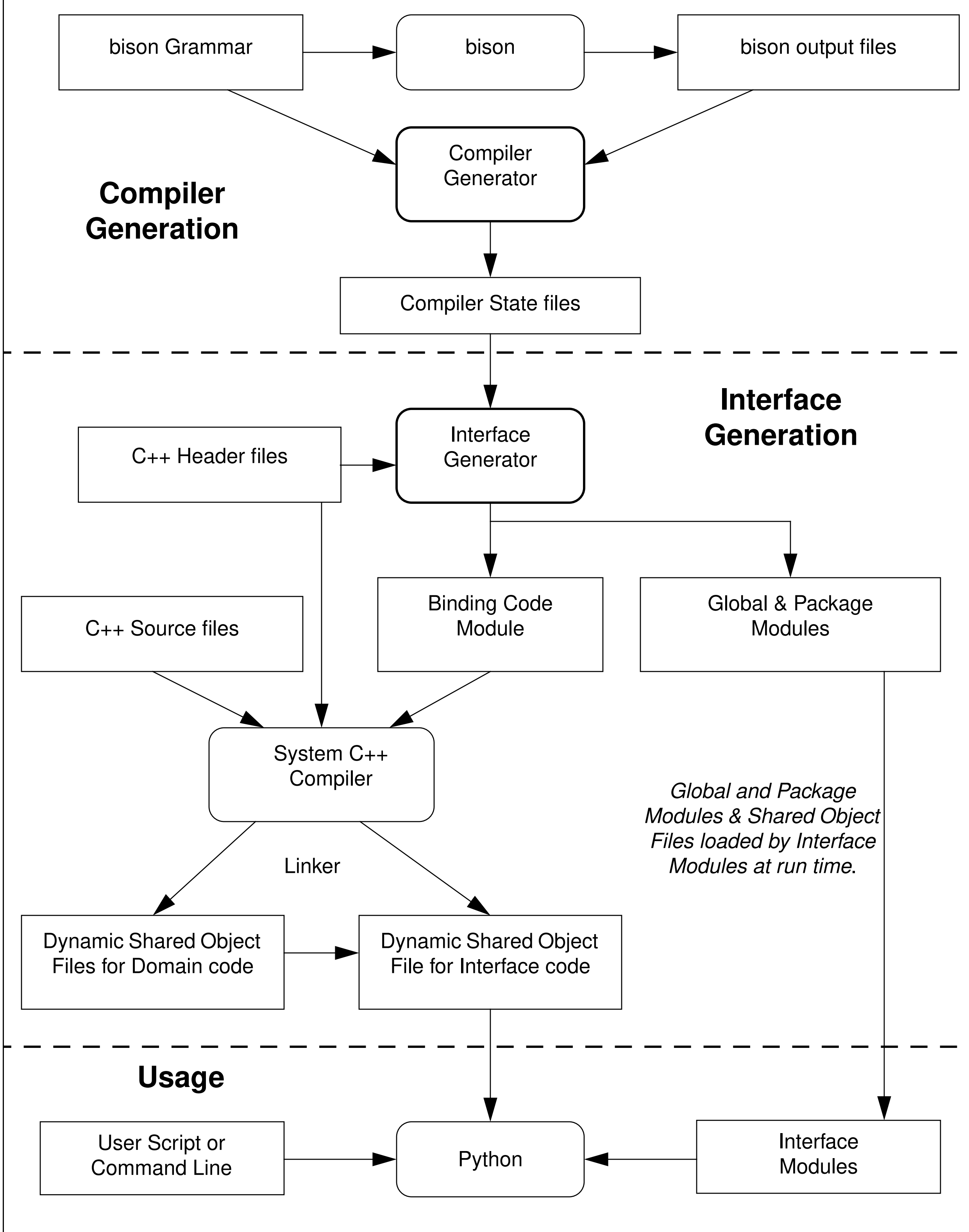

Figure 3. GRAD Generation and Data Flows

4.2 The Interface Generator

The Interface Generator is an LALR(1) parser for the given legacy language. The operation of the parser proceeds in four steps:

- Lexical Analysis - all code is translated into tokens before parsing
- Parsing - translation of the Legacy Code into Metaclass Objects
- Code Generation Preparation - Metaclass Objects are translated to MetaTask Objects
- Code Generation - MetaTask Objects emit Python and Legacy Language interface code

The final step of the process is the compilation and linkage of the interface code modules.

Interfaces are not developed for operators not supported by Python. So '+=', '-=', '*=', '/=' and '()' are not supported. However, in general, any class that has a '+=' operator will also have a '+' operator (which is supported by Python); therefore, an interface to the '+' is generated by GRAD.

Intrinsics (`int`, `long`, `void`, etc.) are handled in Python as values. So no shadow class is created. An intrinsic created in Python does not have a legacy language equivalent. However, it is necessary in C and C++ to represent intrinsic pointers (`void*` for example). Intrinsic shadow classes are created in this case. It is possible to create a pointer in C or C++ from Python or attach to a pointer attribute.

Templates are processed by the GRAD parser, but no interface is generated to attributes or methods that have a templated type, argument or return value. The developers plan to address this deficiency in the near future.

Operator and method overloading is handled by signature type checking in the Python shadow method. Invalid signature types raise an exception before the C++ code is called. The split between Metaclass and MetaTask turns out to be an effective way to handle the output generation problems induced by polymorphism. The Metaclass objects are created ignoring polymorphism. Before MetaTask generation, polymorphic methods are collected into a `Polymorphic_list` object. Thus Python's lack of polymorphism can be bypassed through signature matching within the method interface code generated by MetaTask.

Function and C++ class method pointers are handled by a two stage process. Functions which will be passed by pointer are registered from Python with a `Function_pointer` class. This provides linkage between the method name and its address. The method name is then supplied as an argument to a method or assigned to an attribute of type function pointer.

The GRAD developers actually came up with two ways to address function pointers. The above and an automatically generated process through which the address of any function can be obtained from the binding code through a function call. Since function pointers are not used extensively in most programming, the former method was deemed sufficient.

Enumerations are treated as Python integers (with the same name as the source code definition) with either module or class scope.

Macros in the source code as a named entity in Python are not currently supported. The GRAD Interface Generator works on preprocessed code so all macros have been expanded. The developers are looking into this problem, but to date no adequate solution has been identified.

The Compiler Generator processes one or more source code files into a package. A package is a directory which contains

- The binding code module in the legacy language
- A package module to contain global attributes and methods
- A global code file that contains the interface to global attributes and methods
- One Python shadow module for each file in the source code files.

The shadow module contains the shadow classes corresponding to classes as well as structs and unions. The shadow classes contain the interface to the methods and attributes of the class. Everything in a non-object-oriented language ends up in the global modules.

4.3 Level of Interface Abstraction

An important consideration for the runtime performance of applications written with a generated interface is the level of abstraction provided by the source classes / methods. Obviously if the source code design assumes a native link edited runtime environment, (usually the case), then the Python binding may result in an API which forces the Python program to make high frequency calls, (e.g., tight loops). This can easily lead to unsatisfactory runtime performance. Binding to low level routines will yield workable results only in the simplest of processors.

If this proves to be the case, it will be necessary to provide higher level C++ classes which combine the necessary components such that high frequency calls are not needed by the Python application.

4.4 Usage

The interface is activated by importing the shadow module into Python. For example, an interface to a C++ Vector class is used in the following Python commands

```
>>> import Vector
>>> vars()
{'Vector': <module 'Vector'>, '__doc__': None, '__builtins__': <module '__builtin__'>,
'__name__': '__main__'}
>>> a = Vector.Vector( 1., 2., 3. )
>>> a
<Vector instance at 72590>
>>> print a
#(1 2 3)
>>> x = a.unit_x
>>> y = a.unit_y
>>> z = a.unit_z
>>> print x.cross( y ), y.cross( z ), z.cross( x )
#(0 0 1) #(1 0 0) #(0 1 0)
```

The `unit_x`, `unit_y` and `unit_z` are class attributes defined as `static const Vector`. They represent unit vectors in the base coordinate system. The print statement catches the Vector `ostream` operator output and displays it on the screen.

The C++ Vector class and the generated interface modules are presented in Appendix A.

SECTION 5 - TESTING AUTOMATION

Since the GRAD Compiler and Interface Generators are intended for utilization across multiple projects, they need to be thoroughly tested. Two types of testing are required:

- Code Testing and
- Parser Testing

A set of classes called Paths has been written to aid in coding test development. Using the parser from the Interface Generator, the Paths classes translate a source code module into a tree of Paths objects corresponding to the execution paths in the source code. The Paths classes can be used in two ways:

1. The source code can be instrumented allowing tracking of path coverage against programs and test cases. It is possible to say that test X hits paths A, B and C. It is also possible to say path A is hit by tests 1, 5 and 32. An underlying data base is developed to record the information over time.
2. The tree of Paths objects can emit a set of test prototypes that cover all paths possible in the program. This suite is a near minimum set of tests. The prototypes have to be hand coded at this point into tests.

A follow-on project to add Value Dependency Graphs to the Paths classes is underway. This will allow, in most cases, the specification of the dependency of a decision in the code to the external driving data. With this information the GRAD developers expect to automatically generate a test suite for most of the source code. The remaining tests will have to be manually generated using the test prototypes and Value Dependency results.

Thus far, the only parser that works with Paths is a parser for Python. The C++ parser developed for the Interface Generator ignores executable code since executable code knowledge is not needed for interface generation.

Expanding on the concepts of path coverage, a Parser Test Generator has been written in Python. This tool uses the grammar file input to the Compiler Generator to build a set of legacy language statements that exercise every state transition in the parser. This causes every reduction to be exercised with all possible types of stack states that it can see. In turn, the developers can write more robust actions.

SECTION 6 - COMPARABLE APPROACHES

As the reader may be familiar with other tools used to extend Python, this section will attempt to illustrate GRAD's features by comparing and contrasting other methods with ours. It is not intended as a critique.

6.1 SWIG

SWIG (Simplified Wrapper and Interface Generator) for Python was developed by David Beazley as a follow-on to his automatic wrapper tool used for other scripting languages such as PERL.

GRAD was developed specifically to be a non-invasive hands off method of enabling C++ code in Python. Non-invasive means that the developer of the C++ code does not have to place anything in the source code for GRAD. SWIG does relies upon interface file directives to create the interface code. SWIG is a more generic tool but it does require more work for C++ to Python integration.

Both implementations employ the use of a shadow class. SWIG's shadow class is still in development but is quite similar to that created in GRAD. Data members for classes in both SWIG and GRAD use the `__getattr__` and `__setattr__` methods. GRAD does checking in the `__setattr__` method to assure that the C++ data attribute is not declared `const`. If the data attribute is `const`, an exception is raised. Inheritance in SWIG is immature. All inherited members are wrapped as though they are part if the derived class. GRAD's approach is to let the inheritance mechanism in Python take care of this. Therefore multiple inheritance and Abstract Base Classes are supported in GRAD. Both systems have automatic documentation systems. GRAD's documentation is placed in the Python shadow class and is based solely on the information parsed from the declarations. This information includes function name, return types, signature types, signature identifiers, and default argument values. SWIG includes directives in the interface file for documentation and is output in numerous formats.

Since GRAD has an implementation specifically for C++ and Python, several key items are supported in GRAD that are absent in SWIG. This includes two very significant C++ features, overloaded functions and overloaded operators. To implement these features, signature checking is done in the Python shadow classes to enable the correct method call. Also it should be noted that GRAD only supports, (with one exception), the overloaded operators that are available in Python. GRAD enables overloaded assignment operators by use of a call to a method called `assignment`.

Another significant difference is how the interface is packaged. SWIG wraps the class code, the interface code, and some SWIG support code (pointer string manipulation, etc.) into one shared object. GRAD only creates a shared object for the interface code. By using this method, the C++ code being wrapped by GRAD can be made available via a shared object library. This honors the non-invasive approach that is a tenet of GRAD.

6.2 Header2Scheme

Header2Scheme interface generation is partially automatic. Several manually generated files are required:

- Typedefs - Used to distinguish between classes, typedefs and enumerations. Typedef and enumerations are not parsed by Header2Scheme.
- Opaques - Types of data for which no interface is built.
- Hints - Resolves pointer issues regarding pointers to intrinsics, objects and arrays as well as function pointers.
- Bases - Defines linkage for classes that inherit from one or more base classes.
- Macros - Header2Scheme does not use the system preprocessor. It uses `m4` instead. This file is the input to `m4`.
- Ignores - Files and directories that are not to be parsed.
  GRAD uses a similar concept to define header files that are not included during preprocessing. However, GRAD is not dependent upon the file to build an interface. The reason for GRAD's exclusion list is compiling speed; otherwise many system header files outside the desired Python interface would be parsed.

GRAD automatically handles these instructions.

Header2Scheme is non-intrusive.

Header2Scheme has been used to interface Scheme to a C++ 3-D graphics toolkit called *Open Inventor*. The resulting system is called *Ivy*.

Header2Scheme is specialized to C++. The compiler is directly coded and not generated from a grammar. So Header2Scheme does not offer multiple front-end language support. Header2Scheme, like GRAD, only provides interfaces to public components.

Like GRAD, Header2Scheme does type checking of arguments to methods and operators. Consequently, Header2Scheme support overloading. It also supports inheritance via the Bases input.

Header2Scheme outputs C++ code which is compiled and linked with the Scheme interpreter. Presumably, a dynamic shared library implementation is possible.

6.3 FFIGEN *(Foreign Function Interface GENerator)*

FFIGEN is a system to generate interfaces to C code for multiple back end languages. FFIGEN produces a target-independent representation of the C header file as s-expressions. These are processed by a back end program into the target interface. At present, only Scheme is support as a back end language. Work is underway or planned for STk, Scsh and ILU.

Just as the intermediate language has allowed GRAD to use multiple front-end languages, it has allowed FFIGEN to go to multiple back-end languages. GRAD does not exclude multiple back-end languages, but the developers feel that Python offers some unique properties due to its rich but simple features and easy extensibility and embedability. In fact, the use of the GRAD parser to create the PATHS classes for testing is an example of an alternate back-end for GRAD.

FFIGEN uses `lcc`, a freely available ANSI C compiler, for its front end. FFIGEN can be taken to other languages by substituting another compiler in the front end. The FFIGEN discusses a desire to develop a C++ interface, but no work toward that goal is indicated.

FFIGEN is automatic and non-intrusive.

SECTION 7 - CONCLUDING REMARKS

GRAD has been developed with the vision of long-term software development and maintenance. In particular, it is aimed at providing a mechanism that obviates most needs for application-specific I/O schemes. The GRAD team view these as a major source of complexity growth in legacy code over time. By eliminating the motivation, and somewhat the ability, to couple the I/O scheme into core functionality; the core functionality can remain reusable for an extended period of time.

Since GRAD is grammar-based and produces an intermediate language representation, it can be extended to multiple languages. A Python script can simultaneously access code from multiple languages transferring information bidirectionally between the multiple language code sets.

GRAD has achieved these results through the early definition of these design decisions:

1. A true parser based on a grammar for the legacy language is required.
2. The system should work with multiple languages, so the parser had to be driven using the results of a compiler generator.
3. Only public interfaces are supported.
4. The legacy code should not be dependent upon the GRAD interface.
5. Generation of the interface should be automatic from the source code.
6. The interface generator should produce an intermediate language from which code is emitted.
7. Python becomes the application development language utilizing core software written in multiple languages.

Using GRAD, the promise of entire new applications written rapidly in the high level language Python is attainable while still preserving much of the operational advantage of, and investment in, existing compiled and tested code.

SECTION 8 - REFERENCES

[1] GRAD Concept Document (The Automobile Example). Available from the author.

[2] Python Tutorial, Release 1.2, Guido van Rossum, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, April 10, 1995.

[3] Python Reference Manual, Release 1.2, Guido van Rossum, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, April 4, 1995.

[4] Extending and Embedding the Python Interpreter, Release 1.2, Guido van Rossum, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, April 4, 1995.

[5] Ellis, Margaret A. and Stroustrup, Bjarne: The Annotated C++ Reference Manual ANSI Base Document, Addison-Wesley Publishing Company, Reading Mass.

[6] The C++ Programming Language (second edition), B. Stroustrup, Addison-Wesley Publishing Company, Reading Mass.

[7] Holub, Allen I.: Compiler Design in C, Prentice Hall Software Series, Englewood Cliffs, New Jersey 07632

[8] Aho, Alfred J. and Ullman, Jeffery D., Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Massachusetts

[9] Simplified Wrapper and Interface Generator (SWIG) web page: http://www.cs.utah.edu/~beazley/SWIG/swig.html

[10] Header2Scheme web page: http://www-white.media.mit.edu:80/~kbrussel/Header2Scheme/

[11] FFIGEN (Foreign Function Interface GENerator) web page: http://www.cs.uoregon.edu/~lth/ffigen/

APPENDIX A - EXAMPLE INTERFACE FOR A C++ CLASS

This appendix will present a C++ class, Vector, which is part of the Shuttle simulation software at United Space Alliance. The class is a convenient example because

1. It is fairly small
2. It does not include any other classes nor does it inherit from another class
3. It has some static class attributes so that interface can be demonstrated

Vector has a header file `Vector.h` and a source file `Vector.C`.

The GRAD system translates `Vector.h` into 4 files:

- `Vector.py` - The Python shadow class
- `VECTOR_python.C` - The C++ binding code which is Python cognizant
- `_GLOBAL_.py` - Treatment of global attributes and methods
- `VECTOR.py` - A Python module to hold the name space of global attributes. Imports `_GLOBAL_.py`.

Since the class Vector has no global attributes, `VECTOR.py` is a file of zero length and `_GLOBAL_.py` is just a skeleton.

The Vector `istream` and `ostream` operators are global. But GRAD treats the `ostream` operator as a member of the class so the Python print statement can attach to the `ostream` output. GRAD does not provide an interface to the other stream operators whether part of the class interface or a friend.

GRAD produces a log of messages produced during translation.

Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H
static char* sccs_Vector = “%ccm_header: @(#) Vector.h2.5  15:19:31 95/09/29 (1) % %I%”;
#include <math.h>
class Vector
{
   public:
      static const Vector zero;
      static const Vector unit_x;
      static const Vector unit_y;
      static const Vector unit_z;
      inline Vector(double x = 0.0, double y = 0.0, double z = 0.0)
         { _data[0] = x;
           _data[1] = y;
```

```
      _data[2] = z; }
   inline Vector(const Vector& v)
      { _data[0] = v._data[0];
        _data[1] = v._data[1];
        _data[2] = v._data[2]; }
   inline ~Vector()
      {}
   inline Vector& operator =(double d)
      { _data[0] = _data[1] = _data[2] = d;
        return *this; }
   inline Vector& operator =(const Vector& v)
      { _data[0] = v._data[0];
        _data[1] = v._data[1];
        _data[2] = v._data[2];
        return *this; }
   int compare(const Vector& v, double tol=0.0) const;
   inline double& operator [](int i)
      { return _data[i]; }
   inline const double& operator [](int i) const
      { return _data[i]; }
   inline Vector& operator +=(const Vector& v)
      { _data[0] += v._data[0];
       _data[1] += v._data[1];
       _data[2] += v._data[2];
        return *this;}
   inline Vector& operator -=(const Vector& v)
      { _data[0] -= v._data[0];
       _data[1] -= v._data[1];
       _data[2] -= v._data[2];
        return *this; }
   inline Vector& operator *=(double d)
      { _data[0] *= d;
       _data[1] *= d;
       _data[2] *= d;
        return *this; }
   inline Vector& operator /=(double d)
      { double d_inv = 1.0 / d;
        _data[0] *= d_inv;
       _data[1] *= d_inv;
       _data[2] *= d_inv;
         return *this; }
   inline Vector operator -() const
     { return Vector( -_data[0], -_data[1],-_data[2] ); }
   inline friend Vector operator +(const Vector& v, const Vector& w)
      { return Vector( v._data[0] + w._data[0],
                       v._data[1] + w._data[1],
                       v._data[2] + w._data[2] ); }
   inline friend Vector operator -(const Vector& v, const Vector& w)
      { return Vector( v._data[0] - w._data[0],
                       v._data[1] - w._data[1],
                       v._data[2] - w._data[2] ); }
   inline friend Vector operator *(double d, const Vector& v)
      { return Vector( v._data[0] * d,
                       v._data[1] * d,
                       v._data[2] * d ); }
   inline friend Vector operator *(const Vector& v, double d)
      { return Vector( v._data[0] * d,
                       v._data[1] * d,
                       v._data[2] * d ); }
```

```
      inline friend Vector operator /(const Vector& v, double d)
         { double dinv = 1.0 / d;
           return Vector( v._data[0] * dinv,
                          v._data[1] * dinv,
                          v._data[2] * dinv ); }
      inline int length() const
        { return 3; }
      inline double norm() const
         { return sqrt ( _data[0] * _data[0] +
                         _data[1] * _data[1] +
                         _data[2] * _data[2] ); }
      inline double dot(const Vector& v) const
         { return ( _data[0] * v._data[0] +
                    _data[1] * v._data[1] +
                    _data[2] * v._data[2] ); }
      double angle(const Vector& v) const;
      inline Vector cross(const Vector& v) const
         { return Vector( _data[1] * v._data[2] - _data[2] * v._data[1],
                          _data[2] * v._data[0] - _data[0] * v._data[2],
                          _data[0] * v._data[1] - _data[1] * v._data[0] ); }
      inline Vector unit() const
         { double dinv = 1.0 / norm();
          return Vector( _data[0] * dinv,
                          _data[1] * dinv,
                          _data[2] * dinv ); }
   private:
      double _data[3];
};
//----------------------------------------------------------------------------
//
// Overloaded stream operators for Vector input and output.
//
//----------------------------------------------------------------------------
#include <fstream.h>
ostream& operator  <<( ostream& s,  const Vector& v );
istream& operator  >>( istream& s,  Vector& v );
ofstream& operator <<( ofstream& s, const Vector& v );
ifstream& operator >>( ifstream& s, Vector& v );
//----------------------------------------------------------------------------
//
// FILENAME:
//    Vector.h
//
// DESCRIPTION:
//    The Vector class provides the functionality of vectors in a
//    3-dimesional real vector space. For n-dimensional vectors, use the
//    N_vector class.
//
// AUTHOR:
//    Navigation TOT team.
//
// TO DO:
//    None.
//
// USAGE:
//
//    inline Vector(double x = 0.0, double y = 0.0, double z = 0.0)
//
//      Default constructor allows specification of three components.
```

```
//
//    inline Vector(const Vector& v)
//
//      Copy constructor.
//
//    inline ~Vector()
//
//      Default destructor.
//
//    inline Vector& operator =(double d)
//
//      Assignment from scalar operator.
//
//    inline Vector& operator =(const Vector& v)
//
//      Assignment from Vector operator.
//
//    int compare(const Vector& v, double tol=0.0)
//
//      Compare each component of this and v within component tolerance.
//
//    inline double& operator [](int i)
//
//      Component modifier.
//
//      Precondition: 0 <= i <= 2
//
//    inline const double& operator [](int i) const
//
//      Component accessor.
//
//      Precondition: 0 <= i <= 2
//
//    inline Vector& operator +=(const Vector& v)
//
//      Returns this = this + v.
//
//    inline Vector& operator -=(const Vector& v)
//
//      Returns this = this - v.
//
//    inline Vector& operator *=(double d)
//
//      Returns this = d * this.
//
//    inline Vector& operator /=(double d).
//
//      Returns this = 1/d * this.
//
//      Precondition: d != 0.0.
//
//    inline Vector operator -() const
//
//       Returns -this.
//
//    inline friend Vector operator +(const Vector& v, const Vector& w)
//
//      Returns v + w.
//
```

```
//    inline friend Vector operator -(const Vector& v, const Vector& w)
//
//      Returns v - w.
//
//    inline friend Vector operator *(double d, const Vector& v)
//
//      Returns d * v.
//
//    inline friend Vector operator *(const Vector& v, double d)
//
//      Returns d * v.
//
//    inline friend Vector operator /(const Vector& v, double d)
//
//      Returns 1/d * v.
//
//      Precondition: d != 0.0.
//
//    inline int length() const
//
//      Returns the length (number of components) of this.
//
//      Postcondition: length() = 3.
//
//    inline double norm() const
//
//      Returns the norm (magnitude) of this.
//
//    inline double dot(const Vector& v) const
//
//      Returns the dot (scalar) product of this and v.
//
//    double angle(const Vector& v) const
//
//      Returns the angle between this and v.
//
//      Precondition: this != zero, v != zero.
//
//    inline Vector cross(const Vector& v) const
//
//      Returns the cross (vector) product of this and v.
//
//    inline Vector unit() const
//
//      Returns the unit vector in direction of this.
//
//      Precondition: this != zero.
//
//    static const Vector zero
//
//      Zero vector (public data member).
//
//    static const Vector unit_x
//
//      Unit vector along x-axis (public data member).
//
//    static const Vector unit_y
//
//      Unit vector along y-axis (public data member).
```

```
//
//    static const Vector unit_z
//
//      Unit vector along z-axis (public data member).
//
// WARNINGS:
//    No error reporting or handling is provided.
//
// SEE ALSO:
//    None.
//
// KEYWORDS:
//    Vector
//
// REFERENCES:
//    None.
//
//------------------------------------------------------------------------------
#endif
```

Vector.C

```
static char* SCCS_Vector = "%ccm_header: @(#) Vector.C2.5  15:25:02 95/09/29 (1) % %I%";
//------------------------------------------------------------------------------
// Filename:
//    Vector.C
//------------------------------------------------------------------------------
#include <Vector.h>
//------------------------------------------------------------------------------
//
// Comparison function compares each component within the specified tolerance.
//
//------------------------------------------------------------------------------
int Vector::compare(const Vector& v, double tol) const
{
   return ( ( fabs( _data[0] - v._data[0] ) <= tol ) &&
            ( fabs( _data[1] - v._data[1] ) <= tol ) &&
            ( fabs( _data[2] - v._data[2] ) <= tol ) );
}
//------------------------------------------------------------------------------
//
// Returns the angle between this and v as the dot product of the unit
// vectors in the direction of this and v.
// assert ( this != Vector::zero && v != Vector::zero )
//
//------------------------------------------------------------------------------
double Vector::angle(const Vector& v) const
{
   return acos( ( _data[0] * v._data[0] + _data[1] * v._data[1] +_data[2] * v._data[2] ) / ( 
sqrt( _data[0] * _data[0] + _data[1] * _data[1] + _data[2] * _data[2] ) * sqrt( v._data[0] * 
v._data[0] + v._data[1] * v._data[1] + v._data[2] * v._data[2] ) ) );
}
//------------------------------------------------------------------------------
//
// Public constant Vectors: the zero vectors and unit component vectors.
//
//------------------------------------------------------------------------------
```

```
const Vector Vector::zero( Vector(0.0,0.0,0.0) ); // zero vector
const Vector Vector::unit_x( Vector(1.0,0.0,0.0) ); // unit vector along x-axis
const Vector Vector::unit_y( Vector(0.0,1.0,0.0) ); // unit vector along y-axis
const Vector Vector::unit_z( Vector(0.0,0.0,1.0) ); // unit vector along z-axis
//---------------------------------------------------------------------------
//
// Error handler for overloaded stream operators.
//
//---------------------------------------------------------------------------
#define ERROR_HANDLER {s.clear(ios::badbit);   \
                     cout << "error occured: math_spec  " << c << endl ;\
                     return(s);}
//---------------------------------------------------------------------------
//
// Overloaded stream operators for Vector input and output.
//
//---------------------------------------------------------------------------
ostream& operator<<(ostream& s, const Vector& x)
{
   s << "#(";
   s << x[0];
   for ( int i = 1; i < 3; i++ )
   {
      s << " " << x[i];
   }
   s << ")";
   return s;
}
istream& operator>>(istream&s, Vector& x)
{
   Vector y;
   char c;
   s >> c;
   if ( c != '#' ) ERROR_HANDLER;
   s >> c;
   if ( c != '(' ) ERROR_HANDLER;
   for ( int i = 0; i < 3; i++ )
   {
     s >> y[i];
   }
   s >> c;
   if ( c != ')' ) ERROR_HANDLER;
   if( s ) x = y;
   return s;
}
ofstream& operator<<(ofstream& s, const Vector& x)
{
   for ( int i = 0; i < 3; i++ )
   {
      s.write( (char*)&x[i], sizeof(double) );
   }
   return s;
}
ifstream& operator>>(ifstream& s, Vector& x)
{
   for ( int i = 0; i < 3; i++ )
   {
      s.read( (char*)&x[i], sizeof(double) );
   }
```

```
   return s;
```

Vector.py

Note the class attributes `zero`, `unit_x`, `unit_y` and `unit_z` being handled by the `__getattr__` and `__setattr__` methods.

```
import VECTOR_python
from Intrinsic import *
class Vector_cself:
   __inherit__ = ('Vector_cself',)
   def __init__(self, arg):
      self.cself = arg
   def __del__(self):
      del self
class Vector:
   __inherit__ = ('Vector',)
   __sof__ = 'VECTOR_python.so'
   _statics = {}
   def __getattr__(self, name):
      if name in ['zero', 'unit_x', 'unit_y', 'unit_z']:
         if name == 'zero':
            from Vector import *
            tmp_obj = Vector_cself(VECTOR_python.Vector_zero0(self.cself, ))
            Vector._statics[tmp_obj.cself] = 1
            return Vector(tmp_obj)
         if name == 'unit_x':
            from Vector import *
            tmp_obj = Vector_cself(VECTOR_python.Vector_unit_x0(self.cself, ))
            Vector._statics[tmp_obj.cself] = 1
            return Vector(tmp_obj)
         if name == 'unit_y':
            from Vector import *
            tmp_obj = Vector_cself(VECTOR_python.Vector_unit_y0(self.cself, ))
            Vector._statics[tmp_obj.cself] = 1
            return Vector(tmp_obj)
         if name == 'unit_z':
            from Vector import *
            tmp_obj = Vector_cself(VECTOR_python.Vector_unit_z0(self.cself, ))
            Vector._statics[tmp_obj.cself] = 1
            return Vector(tmp_obj)
      else:
         raise AttributeError, name
   def __setattr__(self, name, value):
      if name in ['zero', 'unit_x', 'unit_y', 'unit_z']:
         raise AttributeError, name + ' is read only'
      else:
         raise AttributeError, name
   def __init__(self, *arg):
      '''Vector.__init__ accepts the following arguments:
         [FloatType x, FloatType y, FloatType z] returning a(n) none
         [Vector v] returning a(n) none
      '''
      sig0 = ['FloatType', 'FloatType', 'FloatType']
      sig1 = ['Vector']
      sig2 = ['Vector_cself']
```

```
    try:
        if checksig(arg, sig0):
            self.__dict__['cself'] = VECTOR_python.Vector_Vector0(arg[0], arg[1], arg[2])
        elif checksig(arg, sig1):
            self.__dict__['cself'] = VECTOR_python.Vector_Vector1(arg[0].cself,)
        elif checksig(arg, sig2):
            self.__dict__['cself'] = arg[0].cself
        else:
            raise ArgumentError, Vector.__init__.__doc__
    except ArgumentError, val:
        argerror(val)
def __del__(self):
            '''Vector.__del__ takes no arguments returning a(n) none'''
            if self.cself not in Vector._statics.keys():
                VECTOR_python.Vector_destruct0(self.cself, )
            else:
                pass
def assignment(self, *arg):
    '''Vector.assignment accepts the following arguments:
        [FloatType d] returning a(n) void
        [Vector v] returning a(n) void
    '''
    sig0 = ['FloatType']
    sig1 = ['Vector']
    try:
        if checksig(arg, sig0):
            VECTOR_python.Vector_assignment0(self.cself, arg[0])
        elif checksig(arg, sig1):
            VECTOR_python.Vector_assignment1(self.cself, arg[0].cself)
        else:
            raise ArgumentError, Vector.assignment.__doc__
    except ArgumentError, val:
        argerror(val)
def compare(self, *arg):
    '''Vector.compare accepts the following arguments:
        [Vector v, FloatType tol] returning a(n) IntType
    '''
    sig0 = ['Vector', 'FloatType']
    try:
        if checksig(arg, sig0):
            return VECTOR_python.Vector_compare0(self.cself, arg[0].cself, arg[1])
        else:
            raise ArgumentError, Vector.compare.__doc__
    except ArgumentError, val:
        argerror(val)
def __getitem__(self, key):
    '''Vector.__getitem__ accepts the following arguments:
        [IntType index] returning a(n) FloatType'''
    from types import IntType
    try:
        if type(key) is IntType:
            if key > -1:
                return VECTOR_python.Vector_op_brack0(self.cself, key)
            else:
                raise IndexError, 0
        else:
            raise IndexError, 1, IntType
    except IndexError, val:
        indexerror(val)
```

```
def __setitem__(self, key, value):
   '''Vector.__setitem__ accepts the following arguments:
      [IntType index, FloatType value] returning a(n) void'''
   from types import *
   try:
      if type(value) is FloatType:
         if type(key) is IntType:
            if key > -1:
               return VECTOR_python.Vector_op_brack1(self.cself, key, value)
            else:
               raise IndexError, 0
         else:
            raise IndexError, 1, IntType
      else:
         raise IndexError, 1, FloatType
   except IndexError:
      indexerror(val)
def __sub__(self, *arg):
   '''Vector.__sub__ accepts the following arguments:
      [Vector w] returning a(n) Vector
   '''
   sig0 = ['Vector']
   try:
      if checksig(arg, sig0):
         return Vector(Vector_cself(VECTOR_python.Vector_op_sub0(self.cself, arg[0].cself)))
      else:
         raise ArgumentError, Vector.__sub__.__doc__
   except ArgumentError, val:
      argerror(val)
def __neg__(self):
   '''Vector.__neg__ takes no arguments returning a(n) Vector.'''
   return Vector(Vector_cself(VECTOR_python.Vector_unary_sub0(self.cself, )))
def __add__(self, *arg):
   '''Vector.__add__ accepts the following arguments:
      [Vector w] returning a(n) Vector
   '''
   sig0 = ['Vector']
   try:
      if checksig(arg, sig0):
         return Vector(Vector_cself(VECTOR_python.Vector_op_add0(self.cself, arg[0].cself)))
      else:
         raise ArgumentError, Vector.__add__.__doc__
   except ArgumentError, val:
      argerror(val)
def __mul__(self, *arg):
   '''Vector.__mul__ accepts the following arguments:
      [FloatType d] returning a(n) Vector
   '''
   sig0 = ['FloatType']
   try:
      if checksig(arg, sig0):
         return Vector(Vector_cself(VECTOR_python.Vector_op_mul0(self.cself, arg[0])))
      else:
         raise ArgumentError, Vector.__mul__.__doc__
   except ArgumentError, val:
      argerror(val)
__rmul__ = __mul__
def __div__(self, *arg):
   '''Vector.__div__ accepts the following arguments:
```

```
      [FloatType d] returning a(n) Vector
   '''
   sig0 = ['FloatType']
   try:
      if checksig(arg, sig0):
         return Vector(Vector_cself(VECTOR_python.Vector_op_div0(self.cself, arg[0])))
      else:
         raise ArgumentError, Vector.__div__.__doc__
   except ArgumentError, val:
      argerror(val)
def length(self):
         '''Vector.length takes no arguments returning a(n) IntType'''
         return VECTOR_python.Vector_length0(self.cself, )
def norm(self):
         '''Vector.norm takes no arguments returning a(n) FloatType'''
         return VECTOR_python.Vector_norm0(self.cself, )
def dot(self, *arg):
   '''Vector.dot accepts the following arguments:
      [Vector v] returning a(n) FloatType
   '''
   sig0 = ['Vector']
   try:
      if checksig(arg, sig0):
         return VECTOR_python.Vector_dot0(self.cself, arg[0].cself)
      else:
         raise ArgumentError, Vector.dot.__doc__
   except ArgumentError, val:
      argerror(val)
def angle(self, *arg):
   '''Vector.angle accepts the following arguments:
      [Vector v] returning a(n) FloatType
   '''
   sig0 = ['Vector']
   try:
      if checksig(arg, sig0):
         return VECTOR_python.Vector_angle0(self.cself, arg[0].cself)
      else:
         raise ArgumentError, Vector.angle.__doc__
   except ArgumentError, val:
      argerror(val)
def cross(self, *arg):
   '''Vector.cross accepts the following arguments:
      [Vector v] returning a(n) Vector
   '''
   sig0 = ['Vector']
   try:
      if checksig(arg, sig0):
         return Vector(Vector_cself(VECTOR_python.Vector_cross0(self.cself, arg[0].cself)))
      else:
         raise ArgumentError, Vector.cross.__doc__
   except ArgumentError, val:
      argerror(val)
def unit(self):
         '''Vector.unit takes no arguments returning a(n) Vector'''
         return Vector(Vector_cself(VECTOR_python.Vector_unit0(self.cself, )))
def __str__(self):
         '''Vector.__str__ takes no arguments returning a(n) StringType'''
         return VECTOR_python.Vector_print0(self.cself, )
__doc__ = '''
```

```
Class Vector defines the following attributes:
   zero (READ ONLY) with the type Vector
   unit_x (READ ONLY) with the type Vector
   unit_y (READ ONLY) with the type Vector
   unit_z (READ ONLY) with the type Vector
Class Vector defines the following methods:
   __init__ with the following arguments:
         [FloatType x, FloatType y, FloatType z] returning a(n) none
         [Vector v] returning a(n) none
   __del__ takes no arguments returning a(n) none
   assignment with the following arguments:
         [FloatType d] returning a(n) void
         [Vector v] returning a(n) void
   compare with the following arguments:
         [Vector v, FloatType tol] returning a(n) IntType
   __getitem__ with the following arguments:
      [IntType value] returning a(n) FloatType
   __setitem__ with the following arguments:
      [IntType item, FloatType value] returning a void
   __sub__ with the following arguments:
         [Vector w] returning a(n) Vector
   __neg__ takes no arguments returning a(n) Vector.
   __add__ with the following arguments:
         [Vector w] returning a(n) Vector
   __mul__ with the following arguments:
         [FloatType d] returning a(n) Vector
   __div__ with the following arguments:
         [FloatType d] returning a(n) Vector
   length takes no arguments returning a(n) IntType
   norm takes no arguments returning a(n) FloatType
   dot with the following arguments:
         [Vector v] returning a(n) FloatType
   angle with the following arguments:
         [Vector v] returning a(n) FloatType
   cross with the following arguments:
         [Vector v] returning a(n) Vector
   unit takes no arguments returning a(n) Vector
   __str__ takes no arguments returning a(n) StringType
‘’’
```

VECTOR_python.C

```
#include <Vector.h>

#include <Vector.h>
#include <Vector.h>
#include <Python.h>
#include <iostream.h>
#include <strstream.h>
typedef void _GLOBAL_;
void Python_parse_error()
{
   cout << “Python_tuple_parse_error” << endl;
   exit(1);
}
extern “C”
{
```

```
static PyObject *Vector_zero0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", &gen_name_Vector0->zero);
}
static PyObject *Vector_unit_x0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", &gen_name_Vector0->unit_x);
}
static PyObject *Vector_unit_y0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", &gen_name_Vector0->unit_y);
}
static PyObject *Vector_unit_z0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", &gen_name_Vector0->unit_z);
}
static PyObject *Vector_Vector0( PyObject *self, PyObject *args )
{
   (void)self;
   double x;
   double y;
   double z;
   if (!PyArg_ParseTuple( args, "ddd", &x, &y, &z))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector(x, y, z));
}
static PyObject *Vector_Vector1( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* v;
   if (!PyArg_ParseTuple( args, "l", &v))
      {
         Python_parse_error();
```

```
      }
   return Py_BuildValue( “l”, new Vector(*v));
}
static PyObject *Vector_destruct0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, “l”, &gen_name_Vector0))
      {
         Python_parse_error();
      }
   delete gen_name_Vector0;
   Py_INCREF(Py_None);
   return Py_None;
}
static PyObject *Vector_assignment0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   double d;
   if (!PyArg_ParseTuple( args, “ld”, &gen_name_Vector0, &d))
      {
         Python_parse_error();
      }
   *gen_name_Vector0 = d;
   Py_INCREF(Py_None);
   return Py_None;
}
static PyObject *Vector_assignment1( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* v;
   if (!PyArg_ParseTuple( args, “ll”, &gen_name_Vector0, &v))
      {
         Python_parse_error();
      }
   *gen_name_Vector0 = *v;
   Py_INCREF(Py_None);
   return Py_None;
}
static PyObject *Vector_compare0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* v;
   double tol;
   if (!PyArg_ParseTuple( args, “lld”, &gen_name_Vector0, &v, &tol))
      {
         Python_parse_error();
      }
   return Py_BuildValue( “i”, gen_name_Vector0->compare(*v, tol));
}
static PyObject *Vector_op_brack0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   int int0;
   if( !PyArg_ParseTuple( args, “li”, &gen_name_Vector0, &int0 ) )
```

```
      {
         Python_parse_error();
      }
   return Py_BuildValue( "d", (*gen_name_Vector0)[int0]);
}
static PyObject *Vector_op_brack1( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   int int0;
   double double0;
   if( !PyArg_ParseTuple( args, "lid", &gen_name_Vector0, &int0, &double0 ) )
      {
         Python_parse_error();
      }
   return Py_BuildValue( "d", (*gen_name_Vector0)[int0] = double0);
}
static PyObject *Vector_op_sub0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* w;
   if (!PyArg_ParseTuple( args, "ll", &gen_name_Vector0, &w))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector( *gen_name_Vector0 - *w));
}
static PyObject *Vector_unary_sub0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0 ))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector( - (*gen_name_Vector0) ));
}
static PyObject *Vector_op_add0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* w;
   if (!PyArg_ParseTuple( args, "ll", &gen_name_Vector0, &w))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector( *gen_name_Vector0 + *w));
}
static PyObject *Vector_op_mul0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   double d;
   if (!PyArg_ParseTuple( args, "ld", &gen_name_Vector0, &d))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector( *gen_name_Vector0 * d));
```

```
}
static PyObject *Vector_op_div0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   double d;
   if (!PyArg_ParseTuple( args, "ld", &gen_name_Vector0, &d))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector( *gen_name_Vector0 / d));
}
static PyObject *Vector_length0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "i", gen_name_Vector0->length());
}
static PyObject *Vector_norm0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "d", gen_name_Vector0->norm());
}
static PyObject *Vector_dot0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* v;
   if (!PyArg_ParseTuple( args, "ll", &gen_name_Vector0, &v))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "d", gen_name_Vector0->dot(*v));
}
static PyObject *Vector_angle0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* v;
   if (!PyArg_ParseTuple( args, "ll", &gen_name_Vector0, &v))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "d", gen_name_Vector0->angle(*v));
}
static PyObject *Vector_cross0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   Vector* v;
```

```
   if (!PyArg_ParseTuple( args, "ll", &gen_name_Vector0, &v))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector(gen_name_Vector0->cross(*v)));
}
static PyObject *Vector_unit0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   return Py_BuildValue( "l", new Vector(gen_name_Vector0->unit()));
}
static PyObject *Vector_print0( PyObject *self, PyObject *args )
{
   (void)self;
   Vector* gen_name_Vector0;
   if (!PyArg_ParseTuple( args, "l", &gen_name_Vector0))
      {
         Python_parse_error();
      }
   strstream ss;
   ss << *gen_name_Vector0 << flush;
   return Py_BuildValue( "s", ss.str());
}
PyMethodDef VECTORMethods[] = {
   {"Vector_Vector0", Vector_Vector0, 1},
   {"Vector_Vector1", Vector_Vector1, 1},
   {"Vector_destruct0", Vector_destruct0, 1},
   {"Vector_assignment0", Vector_assignment0, 1},
   {"Vector_assignment1", Vector_assignment1, 1},
   {"Vector_compare0", Vector_compare0, 1},
   {"Vector_op_brack0", Vector_op_brack0, 1},
   {"Vector_op_brack1", Vector_op_brack1, 1},
   {"Vector_op_sub0", Vector_op_sub0, 1},
   {"Vector_unary_sub0", Vector_unary_sub0, 1},
   {"Vector_op_add0", Vector_op_add0, 1},
   {"Vector_op_mul0", Vector_op_mul0, 1},
   {"Vector_op_div0", Vector_op_div0, 1},
   {"Vector_length0", Vector_length0, 1},
   {"Vector_norm0", Vector_norm0, 1},
   {"Vector_dot0", Vector_dot0, 1},
   {"Vector_angle0", Vector_angle0, 1},
   {"Vector_cross0", Vector_cross0, 1},
   {"Vector_unit0", Vector_unit0, 1},
   {"Vector_print0", Vector_print0, 1},
   {"Vector_zero0", Vector_zero0, 1},
   {"Vector_unit_x0", Vector_unit_x0, 1},
   {"Vector_unit_y0", Vector_unit_y0, 1},
   {"Vector_unit_z0", Vector_unit_z0, 1},
   {NULL, NULL}
};
void initVECTOR_python()
{
   (void) Py_InitModule("VECTOR_python", VECTORMethods);
}
```

GLOBAL.py

Vector.h and Vector.C do not have any global attributes or methods so this is a skeleton file.

```
import VECTOR_python
from Intrinsic import *
class _GLOBAL__cself:
   __inherit__ = ('_GLOBAL__cself',)
   def __init__(self, arg):
      self.cself = arg
   def __del__(self):
      del self
class _GLOBAL_:
   __inherit__ = ('_GLOBAL_',)
   __sof__ = 'VECTOR_python.so'
   _statics = {}
   def __setattr__(self, name, value):
      raise AttributeError, name
   def __init__(self, *arg):
      '''_GLOBAL_.__init__ accepts the following arguments:
         [] returning a(n) none
      '''
      sig0 = ['']
      sig1 = ['_GLOBAL__cself']
      try:
         if len(arg) == 0:
            self.__dict__['cself'] = 0
         elif checksig(arg, sig1):
            self.__dict__['cself'] = arg[0].cself
         else:
            raise ArgumentError, _GLOBAL_.__init__.__doc__
      except ArgumentError, val:
         argerror(val)
   __doc__ = '''
Class _GLOBAL_ defines the following methods:
   __init__ with the following arguments:
         [] returning a(n) none
'''
```

LogFile

```
Global name clash on operator << between Vector.h and Vector.h
Global name clash on operator >> between Vector.h and Vector.h
Generating /home/fly/ccm_rose/grad,1_fly/grad/tcl/tc12/p_VECTOR/Vector.py
operator+= not available for class <Vector> because it is not
    supported in the Python language.
operator-= not available for class <Vector> because it is not
    supported in the Python language.
operator*= not available for class <Vector> because it is not
    supported in the Python language.
operator/= not available for class <Vector> because it is not
    supported in the Python language.
/home/fly/ccm_rose/grad,1_fly/grad/tcl/tc12/p_VECTOR/Vector.py generation complete.
```

```
Generating /home/fly/ccm_rose/grad,1_fly/grad/tcl/tc12/p_VECTOR/_GLOBAL_.py
/home/fly/ccm_rose/grad,1_fly/grad/tcl/tc12/p_VECTOR/_GLOBAL_.py generation complete.
```