Design and Implementation of the APGen DSL parser

Pierre F Maldague

1 Introduction

In this document I want to address the APGen DSL and in particular how it is parsed by the APGen executable. My goal is to answer the following questions:

1. How is the DSL specified?
2. How does APGen read and interpret an AAF file?
3. Why is it difficult to maintain the DSL parser?
4. Would it be better to switch to Python or Java or C++ as the adaptation language?
5. What are the requirements on the DSL?
6. What are the requirements on the DSL parser?
7. How can the DSL parser be tested?
8. How could an easier-to-maintain parser be implemented?

In the process of answering these questions, a number of design issues will have to be discussed. As a result, the present document can be considered to provide an informal Design Document for APGen, or at least for the part of APGen that deals with parsing the DSL into executable objects.

2 The current APGen DSL parsing process

In this section I will address the first two questions listed in the introduction: "how is the DSL specified?" and "How does APGen read and interpret an AAF file?".

2.1 *yacc* and *lex*: the basics

The APGen DSL (Domain-Specific Language) evolved incrementally over 20 years. To describe the APGen parser at the end of this incremental evolution, I will describe how the DSL is parsed in the current version of APGen (I am using the reset-proto-B branch because it is the most stable currently). To make things specific, I will be looking at a simple example of an AAF which defines a function call Add:

```
apgen version "sample AAF containing a single function definition"
function Add(a, b)
parameters
   a: float default to 0.0;
   b: float default to 0.0;
{
        return a + b;
}
```

To parse this AAF, APGen uses a so-called "compiler compiler" called *yacc* (Yet Another Compiler Compiler). Although *yacc* is about 45 years old, it is extremely efficient and most compiler compilers found in other languages such as Java and Python use similar techniques. *yacc* is actually the name of an open-source utility which can be invoked as follows:

```
yacc -d grammar.y
```

This command will cause *yacc* to read file grammar.y, which is assumed to be in yacc format, and to produce two output files: one called grammar.c which will be in C source code format, and a C header file named grammar.h. We will return to these two files in a moment.

A whole chapter of computer science is devoted to what is known as "context-free languages" and how to parse them. I won't have time to get in the details here, but I'd like to give the reader an idea of what goes on inside a compiler compiler such as *yacc*. To accomplish my goal, let me explain how one goes about writing a language specification for *yacc* and in particular what a *yacc* source file looks like. The actual file that defines the DSL language in APGen is called grammar.y, located in subdirectory src/apcore/parser of the git repository for APGen. A real *yacc* source file such as grammar.y contains a couple of preambles which contain necessary technical information such as include files and token definitions. I won't get into include files but let's talk about tokens for a second.

The designers of *yacc* thought it was best not to read the input text file directly, but to first break down the text to be parsed into meaningful chunks called tokens. This is accomplished by *yacc*'s companion program, called *lex* (for lexical analyzer). *lex* is a much simpler program than *yacc* because it focuses on regular expressions, which are much simpler beasts than context-free languages. For example, here is how you might define the fixed string "apgen", an arbitrary variable, and an integer or floating-point number in a *lex* source file:

```
"apgen" {
        return TOK_APGEN;
}
[a-zA-Z][a-zA-Z0-9_]* {
        return TOK_VARIABLE;
}
```

```
[0-9]+"."?[0-9]* {
        return  TOK_NUMBER;
}
```

This fragment defines three regular expressions, the first one for the string *apgen*, the second one for variables, and the third one for numbers. According to the above, a variable must start with a letter and be followed by an arbitrary number (including 0) of characters that are either a letter, a digit or an underscore. The regular expression for a number says that the integer part of the number should contain at least one digit, and be followed by an optional decimal point and optional digits. Note in passing that the actual definition used by APGen is more complicated and allows for numbers that start with a decimal point as well as numbers that figure an engineering-style exponent starting with one of the letters E, e, D or d.

The program produced by *lex* when provided with this input above will recognize regular expressions that express the string *apgen* as well as variables and numbers as specified above. Whenever it recognizes one of these patterns, it will return a token value, which in the above example is specified as *TOK_VARIABLE* for a variable and *TOK_NUMBER* for a number. These symbolic variables are defined in the *yacc* source, and the C specification for token values will be found in the grammar.y produced by *yacc* when invoked as shown on the previous page.

Let us now investigate how to write a source file for *yacc* that will allow us to parse the AAF listed a few pages ago. We will assume that our *lex* program (that is, the lexical analyzer program produced by *lex* when fed a suitable source file similar to the above) recognizes the strings *apgen*, *version*, *function*, *parameters*, *float*, *default to*, and *return*, and that it returns the constants *TOK_APGEN, TOK_VERSION, TOK_FUNCTION, TOK_PARAMETERS, TOK_FLOAT, TOK_DEFAULT_TO*, and *TOK_RETURN* respectively. We also assume that the lexical analyzer returns *TOK_STRINGVAL* whenever if finds a string enclosed in double quotes. We will see later how the lexical analyzer can pass the actual value of the string to the *yacc*-generated parser through a *semantic value*-passing mechanism. Likewise, we assume that an arbitrary string other than the ones already defined causes the lexical analyzer to return *TOK_SYMBOL*, and that arbitrary numbers causes it to return *TOK_NUMBER*.

With these preparations, we can define our *yacc* source file like this (we omit the token definition statements):

```
input_file: TOK_APGEN TOK_VERSION TOK_STRINGVAL file_body ;
file_body: function_definition
        | file_body function_definition ;
function_definition: TOK_FUNCTION TOK_SYMBOL '(' parameters ')'
        TOK_PARAMETERS parameter_declarations
        '{' program '}' ;
parameters: TOK_SYMBOL
        | parameters ',' TOK_SYMBOL ;
parameter_declarations: declaration
        | parameter_declarations declaration ;
declaration: TOK_SYMBOL ':' TOK_FLOAT TOK_DEFAULT_TO TOK_NUMBER ;
program: statement
        | program statement ;
statement: assignment
        | return_statement ;
assignment: TOK_SYMBOL '=' expression ';' ;
return_statement: TOK_RETURN expression ';' ;
expression: TOK_SYMBOL
        | TOK_NUMBER
        | expression '+' expression ;
```

Each line in the *yacc* source file is a parsing rule, which defines a *non-terminal* such as *declaration* in terms of other non-terminals and tokens. Alternative possibilities are separated by the "or" symbol, /. Each rule terminates with a semicolon. Notice how *yacc* allows you to use recursion in rules; this makes it very simple to define a program as an arbitrary number of statements, for example.

Having defined our *yacc* and *lex* source files, we can now process them with the UNIX *yacc* and *lex* commands (or their "modern" equivalents *bison* and *flex*) to produce C or C++ output files, which we can link together with a little bit of gluing code to create a parser for our tiny language. We must admit that, at least at this point, our parser is not very useful; all it can do is read an input file. It produces no useful output. However, we can run yacc in debug mode (turning on the YYDEBUG variable), which will provide a rather verbose trace of the parsing process and, more usefully, show us errors if the input file does not conform to our syntax definition (for example, the parser would show us that we have misspelled 'function' as 'fumction'). Thus, we can use our (very simple) parser to validate input files and certify that they conform to the syntax encoded in our *yacc* source file.

2.2 Creating parsers that perform useful work

In order to accomplish something more useful, we must find a way for our parser to generate data structures that will do something useful. For example, many *yacc* tutorials show how to create a source file that generates a calculator. The key in making a parser useful is to associate *semantic values* and *semantic actions* with the rules in the *yacc* source file. The parser generated by *yacc* associates with each non-terminal and each token a semantic value, which in the case of a C++ parser is an instance of a class specified by *yacc*'s YYSTYPE built-in symbol. By default, YYSTYPE is an integer. These semantic values can be manipulated by semantic actions, which are snippets of C or C++ code inserted after each alternative in a *yacc* rule specification. For example, suppose that we have not redefined YYSTYPE, so that semantic values are integers. We can make the rule that defines addition perform some useful work as follows:

```
expression: expression '+' expression
        {
        $$ = $1 + $3;
        };
```

What this does is set the semantic value of the left-hand side as the sum of the semantic values of the two terms appearing in the right-hand side of the rule. Right-hand side terms are identified by their position in the sequence that defines the rule, starting with position 1. Thus, the three items in the right-hand side of the above rule have semantic values $1, $2 and $3 respectively.

Using semantic actions in this way, it is possible to make the parser work as a calculator. But from APGen's point of view, that is not very useful: such a parser only does useful work once, when it reads the input file. In the case of APGen, what we want the parser to do is to create data structures that can be "executed" as many times as necessary after parsing is completed. In other words, we need to define a class of objects capable to represent all the items present in an adaptation file, and we want these objects to have an *evaluate()* or *execute()* method that can be used to perform useful modeling work. We will refer to the class of such objects as the *executable* class.

To this day, the yacc source used by APGen to parse adaptation files still generates a C language parser. In that case, the semantic value must be a C structure such as the default *int* or, as is often used, a C *union* containing the various possible types for a semantic value. But what we really want is to create instances of the executable class, as discussed above. APGen accomplishes this by having the yacc source file call C functions that are implemented in a C++ source file, where it is possible to create the required instances of the executable class without difficulty. This, unfortunately, makes the parsing system complex and difficult to maintain, as will be shown in more detail in the next section.

In addition, the current implementation of APGen does not provide one executable class, but two. Both are defined in APGen's pEsys (**p**arsed **E**xpression **sys**tem) namespace. The first of these is pEsys::*ParsedExpressionContainer*, a class which was refactored from the old *pE* class by Chris Lawler; its main purpose is to encapsulate arbitrarily complex expressions, and the class contains a method called evaluate() which evaluates the expression in terms of its components. The second executable class is pEsys::*instruction*, which encapsulates the statements found inside "programs" such as the decomposition and modeling sections of an activity type, for example. This dual system adds complexity to the overall parsing system by requiring developers and maintainers to work with multiple header and source files, not to mention other nasty details such as compile-time and run-time error reporting.

All this complexity exacts a price from the development process; this will become quite clear in the next section, which is devoted to the APGen implementation of the DSL parser.

3 Implementing the DSL parser

3.1 Current Implementation

Let me start with a table showing the header and source files that comprise the APGen "parsing system".

file name	***file type***	***# of lines***	***content***
grammar.y	*yacc* source (turns into C source when processed by *yacc*)	1,859	states 455 parsing rules and "semantic actions" which create *ParsedExpressionContainer* and *program* objects containing *instructions*
grammar_intfc.C	C++ source	3,091	creates object instances as required by the semantic actions in grammar.y (recall that grammar.y results in a C file which cannot handle C++ constructors directly)
res_intfc.C	C++ source	1,006	*ditto*
apf_intfc.C	C++ source	583	*ditto*
ParsedExpressionSystem.H	C++ header	610	defines the *ParsedExpressionContainer* and related classes (most parsed objects are instances of a class derived from *ParsedExpressionContainer*)
ParsedExpressionSystem.C	C++ source	1,979	implements most of the methods in the *ParsedExpressionContainer* and related classes
RES_eval.C	C++ source	3,333	implements many of the methods in the *ParsedExpressionContainer* and related classes
RES_eval.H	C++ header	323	defines *program* and *instruction* classes
tokens.l	*lex* source (turns into C source when processed by *lex*)	287	states 25 rules for parsing text into tokens and symbols (tokens are used by the rules listed in grammar.y)
lexer_support.C	C++ source	627	turns the strings extracted from the adaptation file into tokens suitable for use by the *yacc*-generated parser
Total		**13,698**	

Thus, the parsing system consists in thirteen thousand lines of code spread among ten different files. These files can easily be found by browsing the APGen git repository (MPS/apgen), and a quick look at their contents will make any elaboration unnecessary beyond the obvious comment "this system is a mess".

In the first half of 2017, I refactored APGen to implement the "Activity Restart Timeline" (ART) requested by the Europa project. This capability affected the parsing system in a limited way, because the DSL did not change apart from the introduction of a few new statements. Most of the changes occurred inside the *ParsedExpressionContainer* class, and in particular in the way in which symbol tables were stored and searched when exercising the *evaluate()* and *execute()* methods. This led to an ART-enabled version of APGen which, although it satisfied the new functionality required by Europa, has the unfortunate side effect of making APGen slower by about a factor of two. As a result, it became necessary to refactor APGen one more time, to try and regain the speed lost in the ART implementation. This work is now in process, and a new design (partially implemented in the *faster-art* branch of APGen) has been demonstrated to result in performance two and a half times better than the pre-ART version of APGen.

Before turning to the new design and the challenges that it brings about, let's look at the way APGen organizes data, and in particular at the layout of symbol tables in the APGen engine.

3.2 APGen symbol table organization

To introduce the topic of symbol table organization, let us revisit the simple *Add* function introduced on page 2. The only variables necessary to implement it are the float variables *a* and *b*. These variables are declared as parameters, and they are accessible anywhere from within the body of the function. When the APGen creates the *Add* function object, it creates an instance of a class called *task*. A *task* contains the members listed below. In our description of task members, we make use of containers defined in the Standard Template Library (STL): *vector* (similar to a C language array of items) and *map* (a set of indexed items that can be retrieved in logarithmic time, i. e., in a time that grows like *logN* for a map of size *N*). Here are the essential members of the *task* class:

- the name of the task ("Add" in this case)
- an STL *vector* of values to hold the values of all symbols needed by the task (2 in this case, *a* and *b*); we refer to this vector as the "values vector"
- an STL *map* that maps variable names ("a" or "b" in this case) into their index in the values vector
- a *program* which tells APGen how to execute the task

Strictly speaking, this is the current design of an APGen task in the faster-art branch of the APGen git repository. Previously delivered versions of APGen use a design which is functionally equivalent to this one, but which was based on home-brewed classes instead of STL containers. For clarity, we are ignoring this difference, which is only of historical interest.

The above is sufficient to describe how data is organized in a single *task*. However, a typical APGen adaptation can contain many more objects in addition to global functions such as *Add*. Here is the list of contents of a typical adaptation:

- global variables
- global functions (such as *Add*)
- concrete resource definitions
- abstract resource definitions
- activity type definitions
- constraints

In this document, we are not going to discuss constraints. In APGen, constraints are passive and never executed; they can therefore be discussed separately from the "active" elements of the adaptation. We will cover constraints in a separate document, or in a chapter to be written later.

To further discuss the organization of data storage in APGen, we must introduce the notion of *scope*.

3.3 Storage scope in APGen

Figure 1 below illustrates the five basic types of objects found in an APGen adaptation: global variables (represented as little boxes), global functions (represented as gears), concrete resources (light green rectangles), abstract resources (light blue rectangles), and activities (orange rectangles).

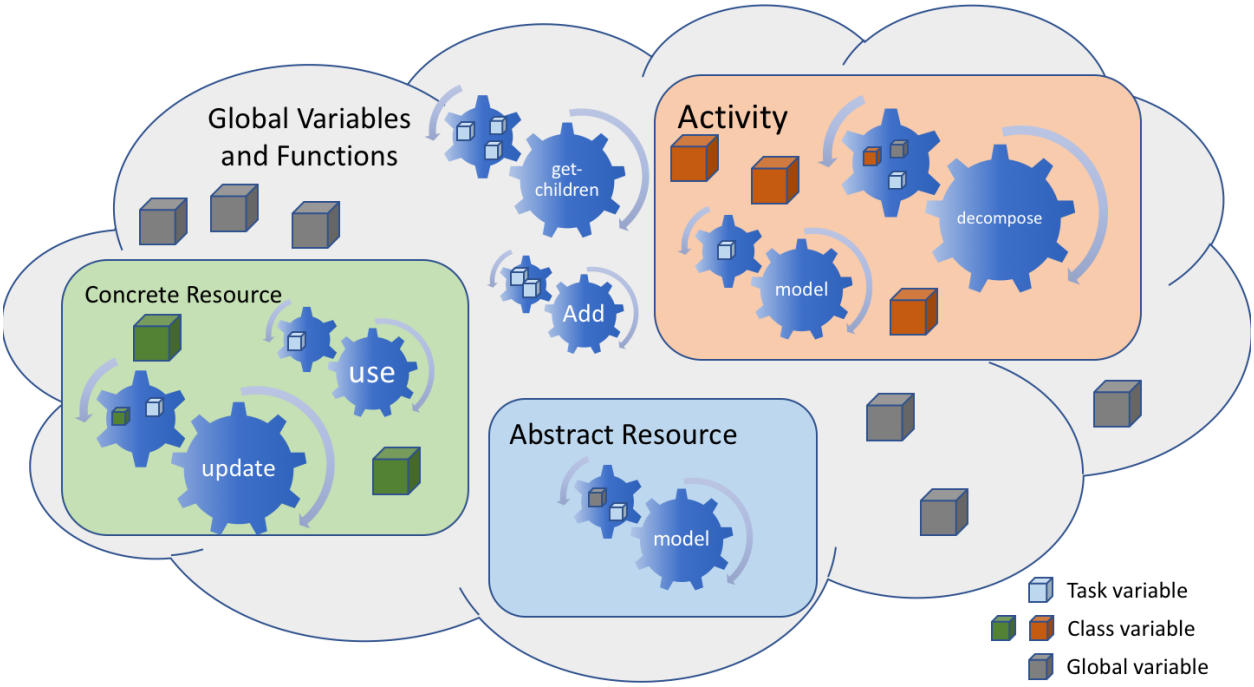

Figure 1: The various types of objects found in an APGen adaptation

Here is how these objects are implemented in the APGen engine. We have already mentioned the fact that global functions are implemented by the *task* class. To provide uniformity in object descriptions, APGen groups tasks into higher-level entities called *Behaviors*. More precisely, these entities are instances of the *Behavior* class. The term *Behavior* is inspired from JPL's IMCE (**I**ntegrated **M**odel-**C**entric **E**ngineering) ontology; it is not a perfect fit, but it reminds us that the whole purpose of APGen and its adaptations is to reproduce as faithfully as possible the behavior of complex objects such as spacecraft, the planetary environment, the ground system, and the links between them. The *Behavior* class contains the following members:

- the name of the *Behavior*
- a set of *tasks* (*task* pointers, actually) arranged as an STL *vector* (the "task vector")
- an STL *map* , the "task index", which maps the name of a *task* into the *task's* index in the task vector
- a set of lower-level behaviors ("SubTypes") which are used to store the actual members of a resource array

The first element of the task vector is always a special *task* called "constructor". If we think of a *Behavior* as a class, then the variables defined inside the constructor can be thought of as the members of that class. The remaining tasks of a Behavior can be thought of as the methods of the class, and the variables defined inside those tasks are the local variables used by the method. This is illustrated in figure 2 below.

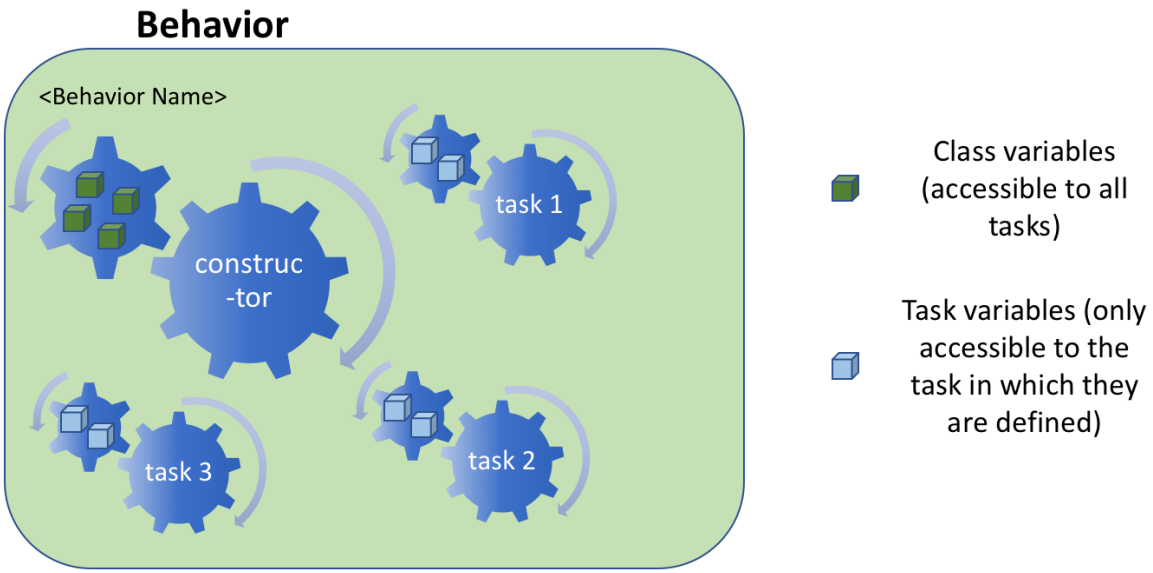

Figure 2: Organization of variables within a Behavior

To make the connection with the DSL, here is how the various elements of the design show up in an adaptation file:

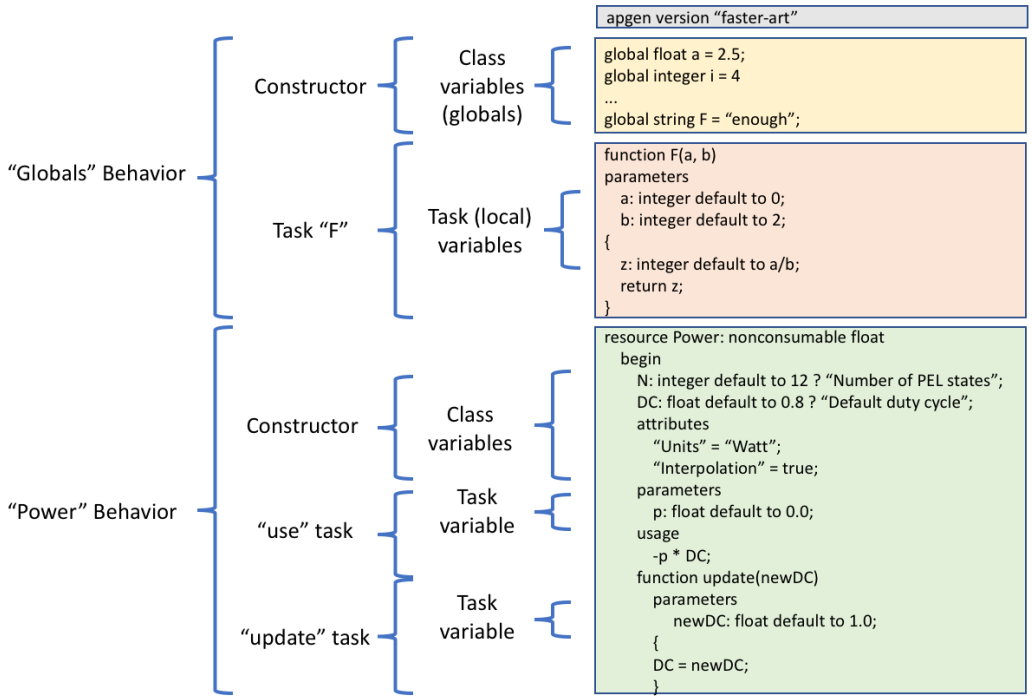

Figure 3: Mapping DSL code into Behaviors & Tasks

Note that we have introduced two new features into the DSL:

- class variables, which are declared the same way as task variables, but appear before the attributes section
- resource functions, such as update, whose definition is similar to a global function but appears inside a resource definition

There are five basic kinds of *Behavior* instances in APGen:

1. Globals (there is only one of these)
2. Concrete Resource (many instances created once and for all in the adaptation)
3. Abstract Resource (many instances created every time the resource is used)
4. Activity (many instances)
5. Constraints (many instances)

3.4 New ("faster") Implementation

In August 2017, I started refactoring APGen again, with the intent to restore the performance of APGen to its previous level or better. The first step of this refactoring effort was to analyze the cause for the slowdown. What came out of the analysis is that the reset-enabled version of APGen made extended use of the symbol tables in which the various objects defined in the adaptation store and maintain their local variables. Symbol tables are implemented via balanced tree structures (AVL trees, to be precise - named after Georgy Adelson-Velsky and Evgenii Landis) which guarantee access time of order log(N) where N is the size of the table. But for large N (say a few hundred) that makes access time non-negligible. By increasing reliance on symbol tables as opposed to more efficient but ad-hoc pointers in object classes, the reset-enabled version of APGen made things worse.

The solution appeared to be clear: the log(N) access time of AVL trees was not good enough, and the APGen symbol tables had to be made more efficient. A little analysis showed that a more efficient design could be obtained by defining symbol tables as STL vectors, which guarantee access in constant time (independent of the size of the data) - provided that you know the index of the variable you are looking for. But the index of any variable is known at "compile" time, i. e., when an adaptation file is read into APGen. The strategy, therefore, was as follows:

1. Replace AVL-style symbol symbol tables by STL vectors, which made access to symbols possible in constant time as opposed to time proportional to log(N).
2. Compute all variable indices at compile time, and store the index so that it could be easily retrieved during execution of the adaptation code.
3. Modify the execution code to take advantage of fast indexing.

To implement these changes, however, required dealing with the unwieldy structure discussed previously: C language files generated by *yacc* and *lex*, linked to interfacing C++ files referencing a variety of classes defined in multiple header files. Since a lot of the code has to be rewritten anyway, it was decided to make some changes to alleviate the pain of reworking the old, complex structure:

4. The Makefile was modified so *yacc* and *lex* produced C++ files instead of C files, making it unnecessary to confine the C++ code to interface files.
5. All semantic data were to be implemented as parsedExp, i. e., smart pointers to instances of the ParsedExpressionContainer (PEC) class which had already been reworked for greater clarity.
6. A new, uniform class called Behavior was introduced to capture the adaptation data.

To make a long story short, this strategy was partially implemented around October 2017. The transition was not complete, but enough had been accomplished that a test case from the Europa project could be read and executed in the new APGen. Doing so demonstrated a performance improvement which not only undid the effects of the switch to the reset-enabled APGen, but was actually two and a half times better than the previous, "fast" release of APGen to the Europa project.

Unfortunately, the refactoring effort which led to this partial implementation was rushed and took some shortcuts which made it painful and time consuming to try and complete that task. In hindsight, the new design suffered from a few basic flaws:

1. Construction of the symbol tables was carried out while parsing the adaptation file. This was consistent with the early design of APGen, but made the code quite complicated.
2. Although the role of the PEC class was enhanced (as it should), the old instruction class - which has significant overlap with the PEC class and should ultimately be eliminated was kept around, making the interface between parsing and execution murkier than it needed to be.

3.5 New Parser Design and Implementation

In December of 2017 I came up with a new design for the APGen parser. The design is illustrated in the figure below.

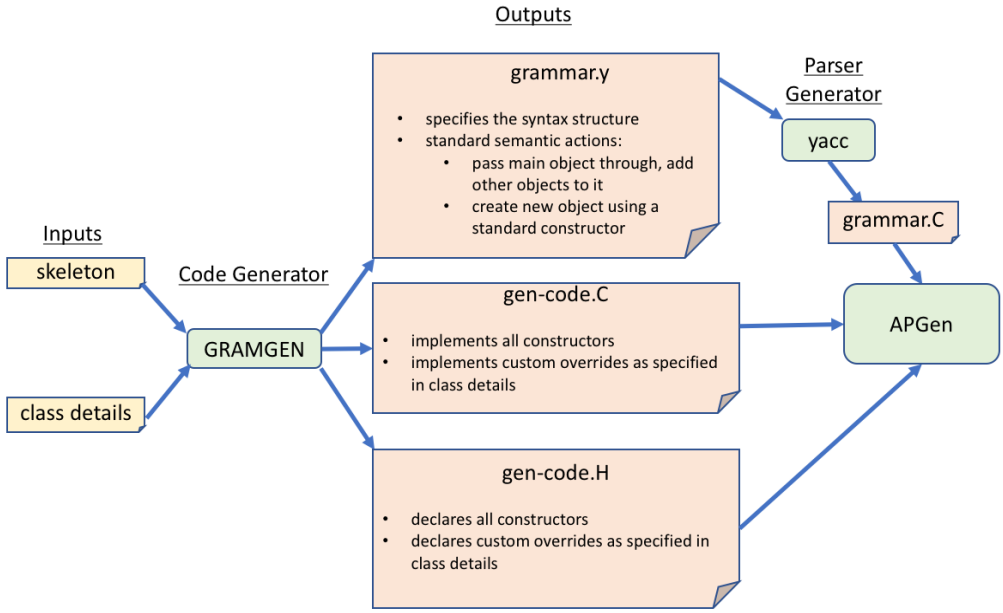

Figure 4: Parser generation process

In this new design, the grammar file that is input to *yacc* is not generated by hand; it is generated by a file generator called *gramgen*. The input to *gramgen* consists of

- a skeleton file, which contains a concise but complete specification of the APGen DSL together with the names of the classes to be associated with the most important objects in the grammar
- a "class details" file, which contains non-boilerplate information which *gramgen* will insert into the generated .C and .H files