Fermilab FERMILAB-TM-2115 June 2000

MAD parsing and conversion code

Dmitri Mokhov[1], Oleg Krivosheev[2], Elliott McCrory[2],
Leo Michelotti[2] and Francois Ostiguy[2]
[1]*University of Illinois at Urbana-Champaign*
[2]*Fermi National Accelerator Laboratory*

June 14, 2000

Abstract

We describe design and implementation issues while developing an embeddable MAD language parser. Two working applications of the parser are also described, namely, *MAD*→C++ converter and C++ factory. The report contains some relevant details about the parser and examples of converted code. It also describes some of the problems that were encountered and the solutions found for them.

1 Introduction

The MAD[1] lattice description language became the *lingua franca* of computational accelerator physics. Any new developments in accelerator physics computational codes and libraries had the requirements to read and understand lattice descriptions written in MAD. The goal of our code is to produce embeddable parser which is able to read, parse and store in memory lattice descriptions, with the ability to generate output compatible with the MXYZPLTK[3] lattice description plus the ability to work directly as a C++ factory module.

2 Design Issues

2.1 General Constraints

MAD language as described in [2] is not well suited for parsing using standard tools like Lex and YACC. Possible alternatives are to write hand-coded parser or to use Lex and YACC but put some restrictions on possible MAD input files. Our preference is to use industry standard tools and second alternative was chosen. We wanted the parser to be usable not only in C++ libraries, but also as module linked to C or Fortran codes, so C language was chosen as least common denominator to write the parser. Therefore it is possible now to link parsing code to any libraries and tools that are required to read and understand the MAD format.

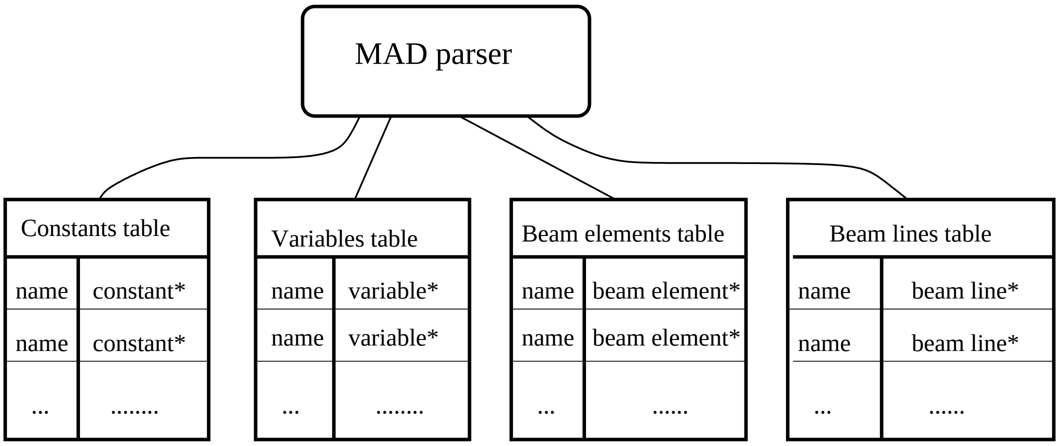
The other design decision came directly from MAD language definition and parser requirements. Because in MAD variables, and thus beam element definitions, could be altered at any point, the only sensible way to build parser is to make it a two-stage program. First stage is reading the MAD input file and do the parsing stage in memory. The second stage of the parser is designed to generate actual output in C++. Such design also has greater flexibility, because output module (second parser stage) could be modified, used to produce C++ object on the fly (C++ factory) and altered in any way required by the task on hands.

2.2 First stage - the parser itself

The first stage, which uses the lexer that is built with Lex (in its Flex[4] incarnation), recognizes MAD keywords, identifiers, numbers, strings, and comments from regular-expression-based rules. It returns corresponding tokens and semantic values to the parser. The parser, written in YACC (we are using the Bison[5] flavor of YACC), contains the grammar for MAD definitions and uses it to recognize those definitions and store them into internal data structures. Because lex and YACC communicate with each other via tokens, one of the parser problems is to take into account the possibility to use shortened directives, e.g HKICKER could be shortened (and often is) to HKICK. We did not offer generic way to handle shortened names and directives, but choose to handle several very common cases separately. Namely, shortened versions of HKICKER, VKICKER, KICKER and MONITORs are recognized now. If there is any demand to handle other shortened directives it can be handled by altering the lexer and parser in very simple and non-intrusive ways. Because of general constraints, the parser stores in memory all the constants, variables, beam element definitions and beam line definition. In order to keep as much information as possible for further processing, all constant and variable definitions that include expressions are parsed and kept as expression trees, not as calculated values. Expression tree allows us either to calculate the expression value or produce output expression in the same way it was written in MAD input file.

2.2.1 Parser internals

The MAD parser itself contains four major tables for storing constants, variables, beam element definitions and beam lines. Because there is a requirement to do fast O(1) search, hash tables were chosen as the container for tables. The comments are stored in an array of strings, because there is no requirements for fast search. Because C is the implementation language, we use data structures from Glib library[6]. The general parser scheme is shown below.



As was mentioned above, almost all of the data structures that are created by the parser to store MAD definitions come from the Glib library. Hash tables are kept for storing information about constants, variables, beam elements, and beam lines. In each hash table, the key is a pointer to the name of the object and the value is a pointer to the corresponding structure. The name is the hash key in the table for fast pointer retrieving during lookup. The tables below show all these structures.

Constant
name
string value
algebraic expression
global line number
local file number
file name

Variable
name
algebraic expression
global line number
local line number
file name

Beam Element
name
kind
length
array of parameters
global line number
local line number
file name

Beam Line
name
beam element list
counter
global line number
local line number
file name

N-ary trees from Glib are employed by the parser for storing algebraic expressions used by constants, variables, and beam elements. Doubly-linked lists (GList pointers in Glib) are used for storing information about the elements of a beam line. Finally, arrays of pointers (GPtrArray pointers in Glib) are used for comments.

2.2.2 What in MAD is Handled

- MAD constant definitions are parsed and stored in the relevant table. Constants can be assigned algebraic expressions as well as string values. Built-in constants from MAD (π, etc.) are predefined.
- All variables with arbitrary algebraic expressions as allowed by MAD syntax are also parsed and stored in the table.
- All beam element definitions are parsed, including exotic ones like matrix and lump elements.
- Beam line definitions are parsed and stored as well, including beam line expressions: inversion, inclusion, and replication.
- Finally, MAD comments are handled in simplistic way. Namely, because it is impossible to analyze the comment, a very simple heuristic is used. We associate the comment that is on the same line as a statement with that statement and associate full-line comments with the statement right after it.

2.2.3 What In MAD Is NOT Handled

- Most importantly, the parser was designed for handling data definitions only. Hence, MAD commands are not interpreted. The lexer understands them, but the parser essentially throws them away by simply outputting a message to the log file. The development direction include adding another table for storing commands. The only exception is multiple include file handling. MAD parser is able to handle arbitrary depth include files and collect information about file names and local line numbers.
- As mentioned above, only limited number of shortened directives are handled. Fortunately, parser will produce error if there are any such cases presented.

2.2.4 Internal Massage Details

After MAD definitions are stored into internal data structures and before any output takes place, there are several actions that need to be taken: checking for variable loops, sorting, and dependence resolutions. Before the output takes place, constants, variables, beam elements, and beam lines are sorted by the line number on which they were defined in the MAD file. Then dependences are also checked and resolved by re-arranging the order of the corresponding objects. Standard Depth-First Search algorithm is used to walk the expression trees and check for graph being acyclic. *At the time of the writing, a circular definition (that is, a loop) can be detected by running the dependency check, which will not properly terminate in case of such a definition.*

2.3 Parser Output Stage

After the parsing step is finished, the parser has all four tables available for further processing. Using Glib supplemental function the user is able to pass through the tables and construct C++ output or create the objects on the fly.

2.3.1 C++ output

One of the parser goals was the creation of C++ output file suitable for inclusion into an Accelerator constructor in the MXYZPLTK[3] library. The conversion MAD -> C++ is performed in four steps. First, all the constants are output, then the variables, then the beam elements, and finally the beam lines. This is done both for easier usage of the resultant C++ file and to better reflect the language structure of MAD, even though the definitions might not be in such separate blocks in the original MAD file. In addition, comments (changed from MAD to C++ style) are also printed in all sections. It is worth mentioning that we produce the same expression structure on output as it was in input file.

2.3.2 Conversion Problems

There are three main problems with converting from MAD to CFG(MXYZPLTK input format, essentially C++), which are results of the absence of direct correspondence between their elements. First, some elements (for example, MULTIPOLE and YROT) are correctly stored in the memory but are printed as comments to the CFG file. Second, other elements (for example, SOLENOID) are correctly stored in the memory but are printed as instances of fictitious classes. Third, elements like ELSEPARATOR and the collimators are replaced by drifts. Output for the most elements includes comments that tell about the above problems and changes. The comments also list the values of parameters that do not have equivalents.

2.3.3 How To Run The Converter

After compilation there is only one executable to run, which is called `mad2cfg`. If executed with no arguments, it prints out a message telling how to use it:

```
Usage: mad2cfg mad_file [cfg_file]
```

There are no options right now. The input file always has to be specified, while the output file argument is optional. If it is not specified, the CFG output will go to STDOUT. Errors always go to STDERR.

2.3.4 C++ factory

When using code in C++ factory mode there is no need to keep expressions stored as trees. Therefore all expressions are calculated and their respective values are used. After first step is done, we walk through the beam elements table and create one object per each entry in this table. The third and last step is to create the beamline and insert each beam element into relevant beam line via clone interface. Then the initial beam elements table and madparser itself may be destroyed.

3 Tools Requirements

Several tools were used for developing and testing the MAD parser. The C-language part of the parser code was written in and tested with GCC v.2.95 compiler. However, the code is written in fairly portable fashion (it passes gcc with -Wall options without warnings) and we expect it to be compiled by any ANSI-C-compatible compiler. The lexer was created using Flex v.2.5.4 scanner generator. It should be fairly compatible with AT&T Lex but may require minor code changes. Bison v.1.27 parser generator was used to create the parser. Again, it should be quite compatible with AT&T YACC, but some small changes in the code may also be required. The program consists of many files which can be compiled and linked using the provided makefile, which is written for GNU Make v.3.77. In addition, C data structures for storing the information about MAD objects were created using the library Glib v.1.2.4, found on the GIMP Toolkit site `http://www.gtk.org`. All the tools mentioned above were made by GNU project `http://www.gnu.org`.

4 Code Availability

The code for the parser is available via read-only anonymous CVS from the server reboot.fnal.gov. In order to be able to connect, set your CVSROOT environment variable to

:pserver:anoncvs@reboot.fnal.gov:/usr/local/cvsroot

Then execute "cvs login" with password "guest". After the login is successfully performed, you can get your private copy of the source files by executing "cvs checkout madparser". Then you should check GNUmakefile to see how include parameters are set, since you might need to change them. Finally, run "make" to compile and build the MAD parser. If you have question or want to have read-write access to repository, please send e-mail to kriol@fnal.gov, who is currently maintaining the code.

5 Conclusion

The MAD parser is fully functional, but there are certainly some features that can be added to it. Most notable omission is the ability to parse MAD commands. Parser was tested with several "real" lattices including Tevatron, Recycler and NLC ones, several bugs were found and fixed.

References

[1] F.Christoph Iselin, "The MAD program(Methodical Accelerator Design) Version 8.13/8", Physical Methods Manual, CERN/SL/92, 1992.

[2] Hans Grotte, F.Christoph Iselin, "The MAD program(Methodical Accelerator Design) Version 8.13/8", User's Reference Manual, CERN/SL/90-13(AP), 1990.

[3] Leo Michelotti et al, MXYZPLTK/Beamline class library, `http://www-ap.fnal.gov/~michelot/`, 1999.

[4] Vern Paxson, Flex, version 2.5.4. A fast scanner generator, Free Software Foundation, 1999.

[5] Charles Donnelly and Richard Stallman, Bison, version 1.28. The YACC-compatible parser generator, Free Software Foundation, 1999.

[6] Now part of Free Software Foundation GNOME project, `http://www.gtk.org`, 1999.