A LEX-BASED MAD PARSER AND ITS APPLICATIONS[*]

O. Krivosheev[†], E. McCrory, L. Michelotti, D. Mokhov[‡], N. Mokhov, J.-F. Ostiguy
FNAL, Batavia, IL 60510, USA
[‡] University of Illinois at Urbana-Champaign, USA

Abstract

An embeddable and portable Lex-based MAD language parser has been developed. The parser consists of a front-end which reads a MAD file and keeps beam elements, beam line data and algebraic expressions in tree-like structures, and a back-end, which processes the front-end data to generate an input file or data structures compatible with user applications. Three working programs are described, namely, a MAD to C++ converter, a dynamic C++ object factory and a MAD-MARS beam line builder. Design and implementation issues are discussed.

1 INTRODUCTION

The MAD[1] lattice description language has become the *lingua franca* of computational accelerator physics. In order to achieve acceptance, new codes and libraries need to recognize lattice descriptions expressed in MAD format. Our objective was [2] to produce an embeddable parser able to read, parse and store lattice descriptions in memory. The parser had to be flexible enough to support various formats; in particular, we needed to translate MAD input format files into C++ files compatible with the BEAMLINE class library [3] as well as a dynamic C++ factory module compatible with that library.

2 DESIGN ISSUES

2.1 General Constraints

Because its grammar does not conform to the LALR(1) conventions[1], the MAD language as described in [4] is not well-suited for parsing using standard tools like Lex and YACC. While it is technically possible to hand-code a specialized parser, the flexibility arising from using Lex and YACC is compelling. We therefore chose to eliminate ambiguities by putting minor restrictions on admissible MAD input files.

The first design decision we faced was to select an implementation language. We wanted the parser to be usable not only with C++ class libraries, but also as a module linkable with C or Fortran. C, the least common denominator, was chosen.

The other design decision came directly from the MAD language definition and parser requirements. Because MAD variables, and thus beam element definitions, can be altered at any point, the only sensible way to build a parser is to make it a two-stage program. The first stage, or front-end, reads the MAD input file and parses it in memory. The second stage or back-end, generates output in a suitable format, e.g. C++. This design is very flexible since the back-end can be modified to support other formats or to dynamically instantiate data structures (C++ factory).

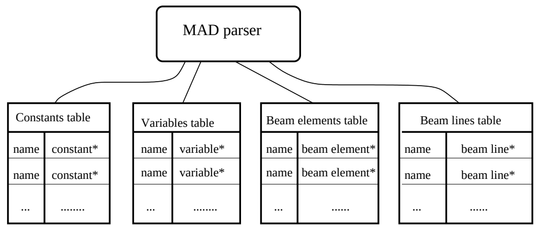
2.2 Front-End

The front-end uses a lexical analyzer built with Lex (in its Flex[5] incarnation). It recognizes MAD keywords, identifiers, numbers, strings, and comments from regular-expression-based rules and returns corresponding tokens and semantic values. The parser, written in YACC (we are using the Bison[6] flavor of YACC), contains the grammar for MAD definitions. It recognizes those definitions and stores them into internal data structures. Because Lex and YACC communicate with each other via tokens, support for abbreviated keywords is an issue. For example, HKICKER is often shortened to HKICK. We decided not to support shortened names and directives generically. Rather, we handle several common cases separately. Any shortened form can be handled by altering the lexical analyzer and parser in a simple, non-intrusive way.

Because of its two stage design, the parser needs to store constants, variables, beam element definitions and beam line definitions in memory. In order to preserve as much information as possible for further processing, expressions are parsed and kept as expression trees, not as calculated values. They can either be evaluated explicitly or appear in the translated output in a form equivalent to the one used in the original MAD input file.

Parser internals The MAD parser uses four tables for storing constants, variables, beam element definitions and beam lines. Since accelerator descriptions often require thousands of symbols, there is a requirement to perform fast searches. Hash tables provide $O(1)$ performance and are therefore used as the container for tables. C does not provide standard containers and algorithms, so we used those provided by the Glib library[7]. The general parser schema is shown below.

[*] Work supported by Universities Research Association, under contract DE-AC02-76CH03000 with the U. S. Department of Energy.
[†] kriol@fnal.gov
[1] In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead.



In each hash table, the key is the name of the object and the value is a pointer to an expression tree. The tables below show all these structures. 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 beamline elements. Finally, arrays of pointers (GPtrArray pointers in Glib) are used to store comments.

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

What 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 parsed.
- All beam element definitions are parsed, including exotic ones like matrix and lump elements.
- Beamline definitions are parsed and stored, including beamline expressions: inversion, inclusion, and replication.
- All MAD comments are preserved. Because it is impossible to analyze a comment, we associate a comment appearing on the same line as a statement with that statement. A full-line comment is associated with the statement that immediately follows it.

What Is NOT Handled

- The parser was designed for handling data definitions only. Hence, MAD commands (e.g. TWISS) are not interpreted with the exception of the INCLUDE command which imports definitions from another file. Although the lexical analyzer recognizes all commands, the output is limited to a message to the log file. The parser can handle nested INCLUDE commands; information about file names and local line numbers is preserved.
- As mentioned before, only a limited number of specific shortened directives are handled. The parser produces an error if unsupported shortened forms are encountered; it is usually a simple matter to substitute the long form using a text editor.

Implementation Details Once MAD definitions are stored into internal data structures and before any output is generated, several actions need to be taken: checking for variable loops, sorting, and dependence resolutions. Constants, variables, beam elements, and beam lines are sorted according to the line number on which they were defined in the MAD input file. Forward dependencies are checked and resolved by re-arranging the order of appearance of the definitions. To prevent and detect circular definitions, a standard Depth-First Search algorithm is used to walk the expression trees and verify that the corresponding graphs are acyclic.

2.3 Back-End

Once the parsing step is completed, all tables are available for further processing. Using Glib support functions, one can walk through the tables to either generate output in a suitable format (e.g. C++) or alternatively, dynamically instantiate objects as needed.

C++ output A major goal for us was to produce a tool that would support translation of MAD input files into a format suitable for the BEAMLINE[3] C++ class library. This has been accomplished. In the BEAMLINE description, constants, variables, beam elements and beamlines are defined in that order. This makes the resultant C++ file easier to use and better reflects the structure of the MAD input language. All expressions and comments are preserved and included in the translation.

Conversion Difficulties Difficulties arise in the translation from from MAD to BEAMLINE input format(essentially C++) because of the absence of an exact one-to-one correspondence between MAD and BEAMLINE elements. Elements like ELSEPARATOR and collimators have no direct BEAMLINE equivalent for the moment and are replaced by drifts. Other elements are correctly represented in memory but either generate comments in the BEAMLINE input file or are treated as instances of placeholder classes (for example, SOLENOID). For most elements, the generated output includes informative comments whenever the correspondence is not exact. Values of parameters that do not have any equivalent are listed in these comments.

C++ factory The MAD parser module can also be used in C++ factory mode. In that case, there is no need to preserve expressions; and they are therefore evaluated. Once execution of the first stage is completed, the code walks through the beamline element table and instantiates objects for every entry. Beamlines are then instantiated and populated with cloned (i.e. deep copies) of the previously created elements. Beamlines objects, as defined in the BEAMLINE class library, contain pointers to beamline elements rather than the elements themselves. Cloning the elements ensures that all beamline elements are distinct. Once all beamlines have been instantiated, the tables and the parser itself may be destroyed.

3 MAD-MARS BEAM LINE BUILDER

The parser has been adapted to serve as a basis for the *MAD-MARS Beam Line Builder* (MMBLB). This module reads a MAD lattice file and constructs the corresponding MARS [8] geometry to allow realistic Monte Carlo simulations of beam-induced energy deposition effects in arbitrary accelerator and beamline configurations. The MMBLB is already successfully used in applications to the Fermilab Booster, Proton Driver, extraction of the Main Injector beam to the NuMI beam line, and to the joint KEK-JAERI project to study beam loss distributions, induced radiation effects and design beam collimation systems. An example of the injection-collimation region description created for the Proton Driver is shown in Fig. 1.

4 TOOLS

Several tools were used for developing and testing the MAD parser. The parser code was tested with the GCC v.2.95 compiler. It should be fairly portable (it passes gcc with -Wall options without warnings) and we expect that compilation with any ANSI-C-compatible compiler should not pose any problem. The lexical analyzer was created using the GNU Flex v.2.5.4 scanner generator. It should be fairly compatible with AT&T Lex but minor changes to the input file may be required. Similarly, the GNU Bison v.1.27 parser generator was used to create the parser and the input file should be highly compatible with AT&T YACC. The program can be compiled and linked using the provided makefile, which is written for GNU Make v.3.77. As mentioned, the C data structures for storing the information about MAD objects were created using the Glib v.1.2.4. library. The source code for this library can be freely downloaded from `http://www.gtk.org`. The GNU tools can be obtained from the GNU project website `http://www.gnu.org`.

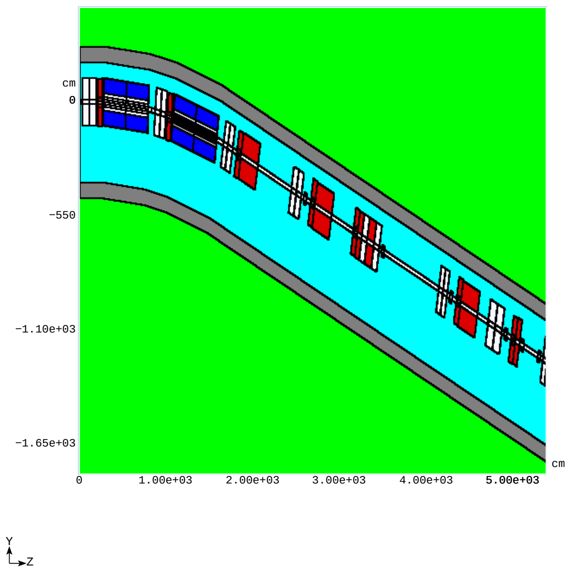


Figure 1: Longitudinal view of the collimation region with shielding as built with MMBLB and implemented into the MARS14.

5 CONCLUSION

The MAD parser is now functional and development continues. It was successfully tested with several very large lattice descriptions including the Tevatron, the Recycler Ring and the NLC. For information about the code and its availability contact kriol@fnal.gov.

6 REFERENCES

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

[2] D.N. Mokhov et al.: “MAD Parsing and Conversion Code”, Fermilab-TM-2115, 2000.

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

[4] 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.

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

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

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

[8] N. V. Mokhov, “The MARS Code System User’s Guide”, Fermilab-FN-628 (1995); N. V. Mokhov and O. E. Krivosheev, “MARS Code Status”, Fermilab-Conf-00/181 (2000). http://www-ap.fnal.gov/MARS/.