# JMAD - INTEGRATION OF MADX INTO THE JAVA WORLD

K. Fuchsberger, V. Baggiolini, R. Gorbonosov, W. Herr, V. Kain, G. Müller,
S. Redaelli, F. Schmidt, J. Wenninger, CERN, Geneva, Switzerland

*Abstract*

*MADX* (Methodical Accelerator Design) is the de-facto standard software for modeling accelerator lattices at CERN. This feature-rich software package is implemented and maintained in the programming languages C and FORTRAN. Nevertheless the controls environment of modern accelerators at CERN, e.g. of the LHC, is dominated by JAVA applications. A lot of these applications, for example for lattice measurement and fitting, require a close interaction with the numerical models, which are all defined by the use of the proprietary *MADX* scripting language. To close this gap an API to *MADX* for the JAVA programming language (*JMad*) was developed. The current implementation provides access to a large subset of the *MADX* capabilities (e.g. twiss - calculations, matching or querying and setting arbitrary model parameters) without any necessity to define the models in yet another environment. This paper describes the design of this project as well as the current status and some usage examples.

## MOTIVATION

Since *MADX* [1] is the de-facto standard software for the design of particle accelerator lattices at CERN it is used by a very large community and there exist lattice models for many accelerators at CERN like the SPS, LHC and the transfer lines in between which are regularly maintained and updated. *MADX* is per design not a library but a standalone software with its own proprietary scripting language which is used to define the models and perform simulation tasks. Although this *MADX*-language contains many elements of a scripting language (like loops or if/else statements) it is by no means (and never was intended to be) a full programming language with custom libraries. Therefore the necessity arises to create *MADX* input files and postprocess output data with other tools, especially when doing complex simulation tasks. The classical way of using *MADX* from a higher level programming language other then *MADX* scripting is:

1. Create a input file for *MADX* (ASCII file) containing model definition calls, input parameters and commands to export the results.
2. Call *MADX* with the created input file.
3. Wait until *MADX* terminates.
4. Parse the *MADX* output files.
5. Postprocess the data (e.g. plot).

Although this can be easily done because of the highly configurable *MADX* text file output features it has many disadvantages, like e.g.:

- Creating *MADX* files by simply composing strings as demanded by the application is very error-prone and makes the application code very dependent on the *MADX* scripting language as well as on the model in use.
- Running *MADX* with different input files requires to start and stop *MADX* every time. Since this also requires to load the sequence (model definition) every time this becomes a very time consuming procedure, especially when many of such iterations are needed (e.g. for fitting purposes).
- Every application developer ends up in implementing its own *MADX* parser.

All these disadvantages can be avoided if steps 1 to 4 are encapsulated in a dedicated software package with a well defined interface for the higher level programming language. All the communication can then be done in the language-typical way which is normally (at least in the case of JAVA) compiler checked and type safe. Even the starting an stopping of *MADX* can be avoided by keeping a running instance with the actual model status in memory.

The JAVA programming language was chosen for the first implementation of this API simply because most of the existing accelerator controls software is written in Java. Meanwhile, also a Python implementation of such an API based on the same principles (*PyMad*) is under development.

## DESIGN

Figure 1 shows the key components of the *JMad* design. They are described in some more detail in the following.

### *JMad Service*

The main facade component for an application which is using the API is the interface `JMadService`. The key responsibility of this interface is to find available model definitions and create model instances from these definitions. The following description will focus on these two key responsibilities in the following, although a lot of additional functionality is provided by this service.

### *Models*

A model is the key component of *JMad*. It is represented by the `JMadModel` interface. Each `JMadModel` instance is associated to one dedicated *MADX* process. The `JMadModel` interface provides JAVA methods to act on the model (e.g. run a twiss calculation, get/set strength values and many more) which are passed on to the *MADX* process.

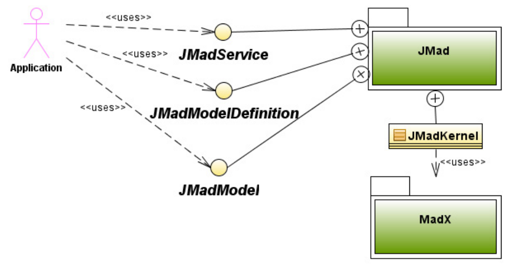

Figure 1: Overview of *JMad* key components

### *Model Definitions*

Although a model can be created simply from scratch by creating a new instance and calling certain methods one by one, the proposed (and most convenient) way is to use predefined model definitions. In the API, a model definition is represented by the JMadModelDefinition interface. Instances thereof contain all the information which is necessary to initialize the *MADX* process (e.g. all the required sequence- and strength-files) as well as available options which are possibly selectable by the user/application (like available sequences, ranges or optics).

Predefined model definitions stored in xml files which contain only the minimum necessary information. The design goal here was to profit as much as possible from the *MADX* scripting language and as a consequence being able to reuse as much as possible from existing *MADX*-files. As a consequence these xml model definitions only act as a link between *MADX*-scripts and are easily understandable and maintainable by conventional *MADX* users.

*JMad* contains an auto-detection mechanism for such model definitions. This mechanism searches for model definitions contained in the JAVA class path in a distinct package. This makes it very easy to extend the available model definitions: A new model definition file must simply be placed in the correct package on the classpath, either as a simple file or inside a jar. Model definition xml files can also be packaged inside a zip file, together with all required *MADX* files. This is very useful e.g. to archive model definitions or to pass them on to other *MADX* users.

## IMPLEMENTATION DETAILS

### *Communication with MADX*

Figure 1 shows that the object which is responsible for the direct communication with *MADX* is an instance of `JMadKernel`. A kernel takes care of its own *MADX* process and the related input-, output- and logging-files. Currently the communication is simply based on streams and files: All the commands and input data are directly written to the input stream of the *MADX* process by the kernel. All *MADX* output data is redirected to files (mostly twiss files) which are parsed by the kernel after the command has finished. Although one could imagine more sophisticated communication methods (e.g. compiling *MADX* as a shared library and communicating via JNA) this method was chosen for the first implementation because it works with the existing *MADX* binaries and only depends on the *MADX* scripting language (which is not supposed to change very frequently) but not on *MADX* internals.

Each `JMadModel` is using its dedicated instance of a `JMadKernel`. The communication between the model and the kernel is done by special command objects whose responsibility it is to compose the correct command strings for *MADX*.

### *Operating System Independence*

Since *MADX* binaries are platform dependent it is evident that a library based on executing these binaries can never be fully platform independent. The problem is circumvented the following way: Executables for different operating systems are included in every *JMad* release. On startup of the *JMad* service, the correct binary for the operating system is extracted to a temporary directory and run from there whenever needed. Currently the operating systems Windows, Linux and Mac OS X are supported.

## USAGE EXAMPLES

To illustrate the usage of the API and introduce some of the features, some simple JAVA code examples are shown in the following.

For example, a typical way to set up a `JMadModel` is shown in listing 1: First a new service is created (Typically an application would use only one such service). This service is then used to find a model definition named "ti2" and create a model. After starting the model (`model.init()`), it is ready to be used. When finishing the work with the model, `model.cleanup()` should be called to close all log files and end the corresponding *MADX* process.

**Listing 1: model setup example**

```
/* create a new JMad service */
JMadService jmadService = JMadServiceFactory.
    createJMadService();

/* find a model definition */
JMadModelDefinition modelDefinition =
    jmadService.getModelDefinitionManager().
    getModelDefinition("ti2");

/* create the model */
JMadModel model = jmadService.createModel(
    modelDefinition);
model.init();

/* do something with the model */

/* finally do a cleanup */
model.cleanup();
```

Although *JMad* also offers a lower abstraction layer which provides more direct (*MADX*-like) access to the model, the recommend abstraction layer to use is a full JAVA representation of the machine which is simulated.

Every machine-element (as defined in a *MADX* sequence) is modeled as a separate JAVA class with appropriate properties and accessor methods, e.g. `Bend`, `Quadrupole` or `Monitor`. The actual optics values (for the whole machine or for single elements) can be retrieved by separate methods. Listing 2 demonstrates some of these capabilities.

**Listing 2: retrieving optics values**

```
Range activeRange = model.getActiveRange();

/* retrieve the actual optics */
Optic optic = model.getOptics();

/* retrieve some optics values */
List<Double> betaxValues = optic.getValues(
     MadxTwissVariable.BETX,
     activeRange.getElements());

/* get an element by name */
Quadrupole aQuad = (Quadrupole) activeRange.
     getElement("MQIF.20400");

/* change a quad strength by 10 percent */
aQuad.setK1(aQuad.getK1() * 1.1);

/* IMPORTANT:
 * refetch optics after changing strengths,
 * it has changed! */
optic = model.getOptics();
```

## APPLICATIONS

*JMad* is used by different applications at the moment, of which some examples are given in the following:

To analyze measurement data for the LHC and perform fits to kick-response-, dispersion-, multiturn- and simple trajectory data a analysis tool called *Aloha* [3] was created. This tool uses *JMad* heavily to vary model parameters and find best fit solutions. During the development of *Aloha* also a graphical user interface to display model data was created. An example screenshot of this GUI is shown in Fig. 2, showing the beta- and dispersion functions of the TI 8 transfer line and LHC sector 78.

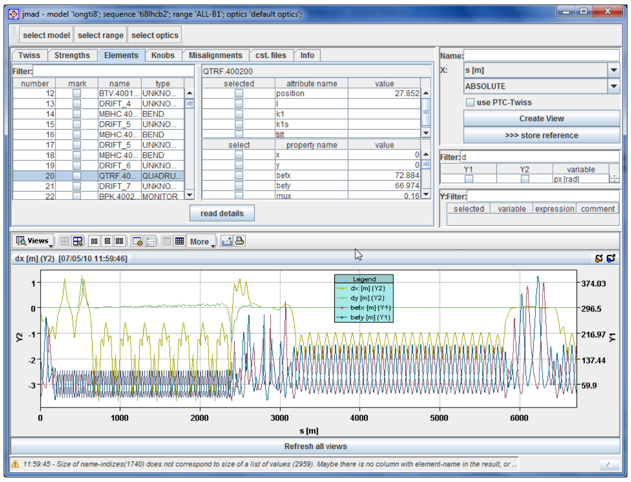

Figure 2: Simple GUI for *JMad*

The LHC Online Model [2] uses *JMad* functionality, especially the feature to define a machine and its optics in a `JMadModelDefinition`, as a backbone to calculate optics functions, upload them to LSA (LHC software architecture) and to create knobs. Additionally, the online model can extract power converter settings from LSA and directly pass them on the JAVA-level to *JMad* to predict aperture margins during aperture measurements (Fig. 3) or scan over generated settings of squeeze beam processes for the LHC, to verify the settings and check / predict the variation of machine parameters, like tune, chroma, beta beat or $\beta$* between the matched optics points.

## SUMMARY AND OUTLOOK

In its current version *JMad* offers the key features for using existing *MADX* models from JAVA, like changing model parameters and calculate optics values. *JMad* is currently used for the LHC online model and various optics-analysis tools. It is the key component to link any JAVA application (like the LHC controls software) in a natural way to *MADX* accelerator models, which are available for almost all accelerators at CERN. Also a simple GUI is available which provides editing- and plotting capabilities and can easily be integrated into other applications.

The next goals for *JMad* development are stabilizing the API and the model-definition format. Further plans include extending the interface for more *MADX* functionality, like e.g. tracking.

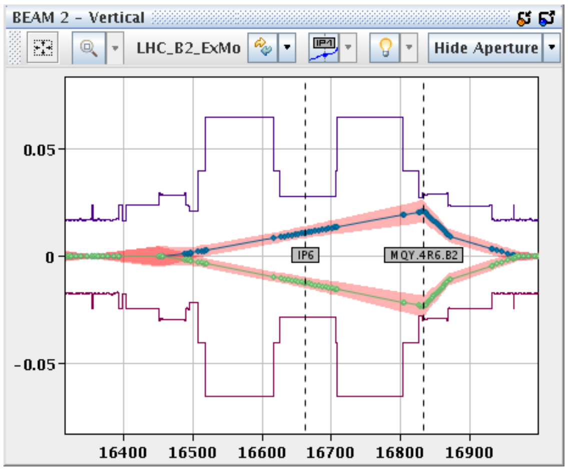

Figure 3: *JMad* calculated orbit excursions from setting extraction, beam envelope calculated from measurements (emittance, beta beat) inside the aperture model.

## REFERENCES

[1] W. Herr, F. Schmidt "A MAD-X Primer", CERN AB Note, CERN-AB-2004-027-ABP.

[2] G. Müller et al., "The Online Model for the Large Hadron Collider", these proceedings.

[3] K. Fuchsberger, "Aloha - Optics studies by combined kick-response and dispersion fits", CERN BE-Note-2009-020 OP.