

OMEI: An Open Mathematical Engine Interface

Weidong Liao Dongdai Lin* Paul S. Wang[†]

Email: {wliao,pwang}@mcs.kent.edu, ddlin@mmrc.iss.ac.cn

Institute for Computational Mathematics

Department of Mathematics & Computer Science

Kent State University

Kent, Ohio 44242, U.S.A.

Abstract Open Mathematical Engine Interface (*OMEI*) aims to establish a uniform application programming interface specification (*API*) for heterogeneous mathematical computation systems. *OMEI* can play an essential role in making mathematical engines easily accessible by front-ends, tools, and servers. The interface enables the development of individual applications that can serve different engines. The motivation, application framework, specification, usage scenarios, and Java implementation for *OMEI* are presented. An application of *OMEI* to connect Starfish with MAXIMA is described.

Keywords: API, OMEI, Mathematical Compute Engine, Internet, IAMC

1 Introduction

Open Mathematical Engine Interface (*OMEI*) is an application programming interface (*API*) specification for providing computational services through a mathematical compute engine. *OMEI* enables tools and applications to use the same uniform interface to access any compliant mathematical engine. The concept of *OMEI* is similar to that of Microsoft's *Open Database Connectivity* (*ODBC*) which has become the standard for PCs and LANs.

The motivation for this work comes from several areas. The Internet has been a platform for a variety of services for more than a decade, but the deployment of mathematical services over the Internet has relatively lagged behind. There is an increasing need to make mathematical computing accessible over the Internet.

Cooperating with other institutions worldwide, the Institute of Computational Mathematics (*ICM*) at Kent State University initiated an *IAMC framework* project [8, 9, 16, 17, 18] to provide an infrastructure for bringing mathematical computational and educational services over the Internet. The *IAMC framework* aims to establish a common protocol to connect interoperable and heterogeneous mathematical clients and servers, to support both interactive and transparent access to mathematical computation on the Internet/Web, and to provide customizable prototypes and libraries to facilitate setting up Internet-based mathematical services. The *IAMC framework* includes an *IAMC client*, an *IAMC server*, and a layered protocol model for connecting *IAMC clients* and servers effectively and efficiently over the Internet.

The computation powers of an existing mathematical compute engine can be made available on the Web/Internet by an *IAMC server* or other similar programs. In order to do so, it is important

*Mathematics Mechanization Research Center, Chinese Academy of Sciences, Beijing 100080, China

[†]Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9721343 and the Ohio Board of Regents Computer Science Enhancement Funds

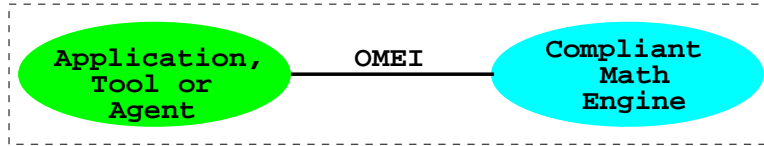


Figure 1: Interfacing Using OMEI

to have an easy and systematic way for network servers to interface with compute engines.

The second motivation comes from *distributed mathematical computation* (DMC), an important research area in symbolic and numeric computation. The goal of DMC is to make mathematical computation accessible and interoperable remotely. To access a remote mathematical compute engine, the architecture generally consists a user interface, a programming interface, and a mathematical encoding on top of the communication network/protocol layers. Researchers worldwide have made contributions in the user interface (e.g. CAS/PI [6], SUI [2] and GI/S [20]), and the encoding (e.g. OpenMath [1], MathML [4] and MP [3]), and the protocol (e.g. MCP [17, 18], OpenXM [15] and KQML [7]) levels. However, the programming interface area requires more investigation. With a well-defined application programming interface, distributed mathematical components, such as front ends, servers, and GUIs, can interoperate with many different remote mathematical engines. The Open Mathematical Engine Interface (OMEI) is an effort in this direction.

Another motivation for OMEI is from the development of new mathematical systems. Generally, a mathematical system contains two main parts: a computation kernel and a user interface. The same kernel can be served by a number of user interfaces designed for different end users—in industry, education, or scientific research. Depending on its purpose, a user interface can be simple and straight-forward or sophisticated and complex. A standard such as OMEI can separate the development of mathematical engines from user interfaces. Therefore, the two can be developed independently and both will be usable with any OMEI compliant components.

APIs for specific mathematical engines, such as Math/Link [13] by Wolfram Research Inc. and Matlab External Interface [14] from MathWorks, exist. They are vendor/engine specific and programming language dependent. Nevertheless, these interfaces provide valuable input and excellent reference for the OMEI effort.

Several other efforts for distributed mathematical computation in Java environment are also noted. JavaMath [5] is one of them. JavaMath is proposed as a standard Java API for client-server mathematical computation over Java and Java RMI. Different from JavaMath and other Java-based approaches, OMEI is an abstract programming interface with well-defined semantics. It specifies an interface that is language, platform, communication protocol, and mathematical engine independent. Nevertheless, an OMEI implementation must be done in a specific programming language for a target mathematical engine.

The rest of the paper begins with an overview of the framework in which OMEI is used. Then, it proceeds to cover the OMEI specification and JMEI, a Java implementation of OMEI. An actual application of JMEI involving interfacing the Starfish, an IAMC server, to the mathematical engine MAXIMA is also presented.

2 Applying OMEI

Before we go into details of the OMEI specification, let's overview the framework within which we expect to use OMEI. The application framework for OMEI is depicted in Figure 1.

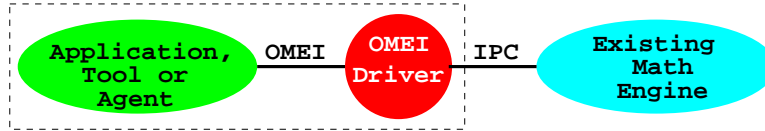


Figure 2: Interfacing with an OMEI Driver

- 1) A front-end – This is a mathematical application, a toolkit, or a networking agent such as a server for the Internet/Web.
- 2) An OMEI compliant mathematical engine – This is a mathematical engine that supports an OMEI compliant API. Here the front-end and the compute engine run as a single process.

A compute engine may come with an OMEI compliant API by itself. ELIMINO [12] may very well become the first compute engine of this type. If OMEI serves its purpose well, then developers will want to create new mathematical engines with a built-in OMEI API.

Interfacing to existing mathematical engines is another story. An existing engine can certainly be reprogrammed to support OMEI. Another approach is to develop a separate *OMEI driver*, a program that implements the OMEI interface for a particular engine in a specific programming language. Figure 2 shows the application framework in this situation.

In this case, the front-end and the OMEI driver run in one process. Depending on the driver implementation, the engine may be internal, external, or remote.

3 OMEI Specification

OMEI aims to be an interface general enough to work for most mathematical compute engines. It is specified after studying existing mathematical engines and their user/programming interfaces. The following properties were noticed:

- A compute engine may execute some initialization commands before it begins to perform user-requested computations. Usually, the initialization sets up the input/output modes, processes customization/configuration parameters, etc.
- A compute engine may execute in several different modes such as normal mode and debug mode.
- A compute engine may send a prompt when it is ready for inputs. Different compute engines use different prompts. The prompt may also indicate the current engine status/mode.
- A compute engine normally takes one command at a time. Each command has a terminating character.
- Commands in the form of mathematical expressions are usually encoded in either ad-hoc or standard formats. As an example of standard encoding format, MathML [4] seems getting more and more support from mathematical engine developers.
- A compute engine returns a result or an error message for a command. Results may contain text, mathematical expressions, and graphics. The beginning and end of the result are well-defined.

- A compute engine may ask for extra information from the user in order to complete a particular computation.
- A compute engine may support several types of user-generated interrupts.
- A compute engine may keep track of commands and results in generated variables.
- Help information and documentation can come from the engine or some other source.
- To conduct computations with a compute engine through its API, an external program usually needs to create and maintain a persistent connection with the engine.
- To manage computation requests to a compute engine through its API, an external program may need to keep track of certain environmental information such as the status of previously submitted requests.
- Before quitting, an external program usually must close the connection and free the allocated resources.

OMEI establishes several categories of function prototypes to encapsulate such properties and thus gives a unified programming-level interface for heterogeneous mathematical engines. The prototypes support several categories of operations for server-engine interaction:

1. Connecting to/disconnecting from a compute engine
2. Querying engine capabilities
3. Creating commands
4. Executing commands

Two handles are used in all these function prototypes: `OMEICONHANDLE` and `OMEICMDHANDLE`. They correspond to the execution environment for each connection and command, respectively. Before making connection to a compute engine, the program must allocate a connection handle. Similarly, a command handle must be allocated before a command is created.

Every OMEI call returns an `OMEIRETURN` code, which is an integer representing the execution status for each call: `-1` means an error occurs; `0` means the call is successful. The meaning of other return values depends on the individual call. For example, for `OMEIExecuteCommand()`, the return value `9` means more information is required to complete the command execution and thus one or more *Dialog* calls are expected (See the following subsection "Execute Commands").

The OMEI prototypes are presented in the following sub-sections.

3.1 Connect and Disconnect

Before the computational power of a compute engine can be exploited, a *communication channel* must be established. This channel is vital for server-engine interaction because it is where the commands are issued and the results obtained. The communication channel could be a simple inter-process communication (IPC) pipe or socket, or a network link based on a communication protocol. Usually, a channel must also contain the context for the specific server-engine pair. Such context information include the current command, the status of its completion, the result after the command execution is completed, and the error information if the command cannot be executed due to some reason.

In OMEI, the term *Connection* refers to the communication channel and the term *Connection Handle* refers to the connection context. Before a connection is established, a connection handle must be allocated. This connection handle can be freed after the connection is closed. OMEI specifies four function prototypes to manage this interaction. Details about these function prototypes can be found in [11].

3.2 Query Engine Capabilities

Once a connection is established, the *application* is ready to interact with the compute engine. Before submitting the actual computation requests to the engine, the application can *inspect* the computational capabilities supported by the compute engine.

Four types of engine capability information can be identified:

- A *CanDo list* that tells what computations the compute engine can provide for this specific connection
- *Help* information for each individual command
- *Command template* that gives the syntax of an individual command
- *Mathematical encodings* supported

Correspondingly, OMEI provides four query operations: `OMEIGETCanDo()`, `OMEIGetHelpOnCommand()`, `OMEIGetTemplateOnCommand()`, and `OMEIGetSupportedEncodings()`. Again, detailed specification for each operation can be found in [11].

3.3 Creating Commands

A command can be created in three different ways: by giving a string for the command, by giving a file containing a mathematical expression, or by making several OMEI calls to enter the command name and arguments. In the first and second approaches, the command can be either a literal string representing a native command for the engine, or math-encoded in a format such as MathML, OpenMath, or MP. In the third approach, several OMEI calls are used to specify the command name and arguments.

A *command handle* must be allocated before a command is created. The command handle is used to save the execution context for an individual command. From the command handle, we can check the status of a command, fetch the execution result, and interrupt the execution if necessary. Once the command execution is finished and the result is fetched, the command handle can be released to free the related resources.

3.4 Executing Commands

Once a command is created, it can be forwarded to the engine for execution. OMEI specifies a group of function prototypes to perform and coordinate the command execution. There are five function prototypes in this group: `OMEIExecuteCommand()`, `OMEICancel()`, `OMEIResultEncoding()`, `OMEIGetResult()`, `OMEIQueryDataType()`, and `OMEIPutDialogData()`.

`OMEIExecuteCommand()` is a *non-blocking call*— it returns without waiting for the result of the command execution. An ongoing execution may be cancelled through an `OMEICancel()` call. Once the execution is finished, the result can be fetched by `OMEIGetResult()`.

Two *Dialog* operations, `OMEIQueryDataType()` and `OMEIPutDialogData()`, are specified for the special situations when the compute engine needs to ask for additional information in order to

complete processing a command. For example, the MAXIMA command

integrate(1/x, x, a, b)

will initiate a dialog as follows:

Is b, negative, positive or zero

In this situation, the invocation of `OMEIGetResult()` following `OMEIExecuteCommand()` will return a special code indicating that a dialog is required to complete the execution. Now the application calls `OMEIQueryDialogType()` to check the dialog type, and then calls `OMEIPutDialogData()` to send the appropriate information to the compute engine.

4 Application-Engine Communication Scenarios

To illustrate the meaning and purpose of the OMEI specified interface, let us look at a typical scenario for an application to perform computations following the OMEI specification.

For example, to perform the simple computation *integrate(sin(x), x)*, an application would carry out these steps:

```
#include <omei.h>
main()
1){ OMEICONHandle *mc;      /* allocate connection handle */
2)  OMEICMDHandle *cmd;    /* allocate command handle    */
3)  OMEIString userName = "test", passwd ="test";
4)  OMEIString engineName = "maxima";
5)  OMEIString mathMLResult;

6)  OMEIAllocConnect(mc);
7)  OMEIConnect(mc, engineName, userName, passwd);

8)  OMEIAllocCommand(cmd);
9)  OMEICreateCommand(cmd, MATHML,
    "<math><apply><fn>INTEGRATION</fn> \
    <apply><sin/><ci>X</ci></apply> \
    <ci>X</ci></apply></math>");

10) OMEIResultEncoding(mc, cmd, MATHML);
11) OMEIExecuteCommand(mc, cmd);

12) OMEIGetResult(mc, cmd, mathMLResult);

13) OMEIDisconnect(mc);
14) OMEIFreeCMDHandle(cmd);
15) OMEIFreeCONHandle(mc);
}
```

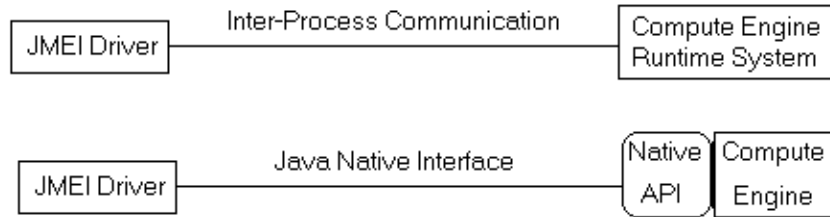


Figure 3: Implement JMEI Driver through IPC or JNI

First, a connection to the compute engine is established (lines 6-7). Then, an OMEI command is created from an MathML encoded string (lines 8-9). After specifying the intended encoding for the computation result, the created command is submitted to the connection for execution (line 10-11). The application then retrieves the result (line 12). Finally we close the connection and free the allocated resources (lines 13-15).

5 Implementation in Java

In spite of the C-style syntax used for its specification, OMEI remains abstract and language independent. A concrete implementation of OMEI must be in a specific language.

Let us consider implementing OMEI in Java. To make the task easier, we first translate the C-style specification into an object-oriented specification in the Java package `org.icm.omei`. The `org.icm.omei` package consists of Java *interface definitions* for the OMEI function prototypes. For more detailed information about `icm.omei` please refer to the ICM technical report [10].

Implementing the `icm.omei` interfaces, we have developed an OMEI driver for the mathematical system MAXIMA. The description of this Java OMEI driver can be helpful for implementing OMEI for another system or in another language.

In the following (and the corresponding technical report about `org.icm.omei` [10]) we use the term *JMEI* refer to our Java implementation of OMEI.

5.1 Implementation Strategies

There are basically three approaches to the implementation of JMEI for a specific compute engine. If JMEI uses inter-process communication (IPC) to access the mathematical engine, it may use the `java.lang.Process` and `java.lang.Runtime` facilities. If JMEI uses procedure calls to access to the engine, the Java Native Interface (JNI) scheme can be used. This approaches is feasible if only the compute engine comes with an API for a host language that JNI supports. Figure 4 illustrates these two approaches.

If the compute engine supports a protocol for network-based access, Mathematica for example, a 2-tier approach can be used to build a network-enabled JMEI driver. In this situation, the JMEI driver can use the client side API to communicate with a remote compute engine. Figure 5 shows this approach.

The IPC approach is the most generic yet the most complicated implementation paradigm. If the computational package comes only with a runtime system for interactive use, that is, without API support for any mainstream programming language, the IPC would be the only implementation choice for its JMEI driver. Here JMEI employs a two-way pipe between itself and the external compute engine.



Figure 4: Implement a Network-Enabled JMEI Driver

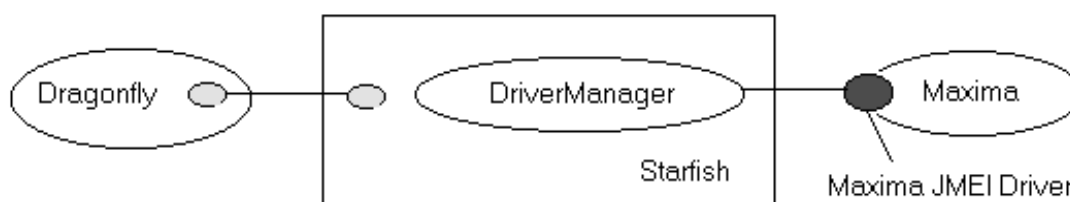


Figure 5: IAMC Prototyping Structure

The complication for the IPC approach lies in the synchronization between the process's input channel and output channel. To read the execution results, the beginning and end of the result for each command must be identified correctly. This may become quite involved if we take into consideration that the result may be an error message. The solution is often engine-specific. If the compute engine output results and errors in MathML, then it becomes much easier.

5.2 JMEI MAXIMA Driver

The IAMC framework follows a 3-tier architecture that consists of an IAMC client, an IAMC server, and a back-end mathematical compute engine. OMEI is used to connect the IAMC server to external engines.

Our Java-based IAMC prototyping system is shown in Figure 5. Dragonfly is a generic IAMC client, and Starfish is the IAMC network server. The JMEI driver is a dynamically loadable module that encapsulates the computational capability of back-end engines. Starfish is configured to dynamically locate and load the OMEI driver. With the loaded driver, Starfish will be able to forward the computational requests to the engine and results to Dragonfly or any other IAMC clients.

For testing JMEI framework and as part of efforts of IAMC prototyping, we have implemented a JMEI driver for MAXIMA, a common-Lisp based symbolic package. Maxima comes with no application programming interface. The IPC approach is therefore chosen to implement this Maxima driver.

To establish inter-process communication channel, *java.lang.Process* and *java.lang.Runtime* are used, as shown in the following code:

```
Process proc = Runtime.getRuntime().exec("maxima");
```

```
InputStream inputChannel = proc.getInputStream();
OutputStream outputChannel = proc.getOutputStream();
InputStream errChannel = proc.getErrorStream();
```

The JMEI Maxima driver consists of the following main Java classes:

```
public class MaximaDriver implements org.icm.omei.Driver;
public class MaximaConnection implements org.icm.omei.Connection;
public class MaximaCommand.java implements org.icm.omei.Command;
public class MaximaResult.java implements org.icm.omei.Result;
public class MaximaDriverInfo implements org.icm.omei.DriverInfo;
```

6 Conclusions and Future Work

The OMEI effort investigates ways to establish a uniform API for mathematical compute engines and how such a uniform API can make distributed mathematical computation easier. We have applied the OMEI specification in one application, namely connecting Starfish with MAXIMA. The success of this application encourages us to work further in this direction.

We will apply OMEI in connecting additional engines and we plan to make ELIMINO, a new symbolic computation system being developed at the Institute for System Sciences of the Chinese Academy of Science, OMEI compliant. The OMEI specification will be further refined and its performance evaluated. Eventually, we hope to make OMEI an independent specification, to build efficient OMEI drivers with good performance, and to promote the development of interoperable front-ends for mathematical compute engines.

References

- [1] ABBOTT, J., DIAZ., A., AND SUTOR, R. S. *Report on OpenMath*. ACM SIGSAM Bulletin (Mar. 1996), 21-24.
- [2] DOLEH, Y., AND WANG, P. S. *SUI: A System Independent User Interface for an Integrated Scientific Computing Environment*. In Proc. ISSAC 90 (Aug. 1990), Addison-Wesley (ISBN 0-201-54892-5), pp. 88-95.
- [3] GRAY, S., KAJLER, N., AND WANG, P. *Design and Implementation of MP, A Protocol for Efficient Exchange of Mathematical Expressions*. Journal of Symbolic Computation 25 (Feb. 1998), 213-238.
- [4] ION, P., MINER, R., BUSWELL, S., S. DEVITT, A. D., POPPELIER, N., SMITH, B., SOIFFER, N., SUTOR, R., AND WATT, S. *Mathematical Markup Language (MathML) 1.0 Specification*. (www.w3.org/TR/1998/REC-MathML-19980407), Apr. 1998.
- [5] JavaMath. <http://javamath.sourceforge.net/>.
- [6] KAJLER, N. *CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems*. Proceedings of ISSAC'92, ACM Press 1992
- [7] KQML Home Page. <http://www.csee.umbc.edu/kqml>.

- [8] LIAO, W. and WANG, P. S. *Building IAMC: A Layered Approach*. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00). pp. 1509-1516.
- [9] LIAO, W. and WANG, P. S. *Dragonfly: A Java-based IAMC Client Prototype*. Lecture Notes on Computing Vol.8. (Proceedings of ASCM 2000.) World Scientific Press, pp. 281-290.
- [10] LIAO, W. and WANG, P. S. etc. *Specification of JMEI: Java Mathematical Engine Interface*. ICM Technical Report. 2001. <http://icm.mcs.kent.edu/reports/index.html>.
- [11] LIAO, W. etc. and WANG, P. S. etc. *Specification of OMEI: Open Mathematical Engine Interface*. ICM Technical Report. 2001. <http://icm.mcs.kent.edu/reports/index.html>.
- [12] LIN D., LIU J. and LIU Z. *Mathematical Research Software: ELIMINO*. Proceedings of ASCM'98. pp. 107 - 116, Lanzhou Univ., China, 1998.
- [13] MathLink Home Page. <http://www.wolfram.com/solutions/mathlink>.
- [14] MATLAB External Interface. <http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml>.
- [15] OpenXM Home Page. <http://www.openxm.org>.
- [16] WANG, P. S. *Internet Accessible Mathematical Computation*. In the 3rd Asian Symp. on Computer Mathematics (ASCM'98), pp 1-13, Lanzhou Univ., China, 1998.
- [17] WANG, P. S., GRAY, S., KAJLER N., LIN D., LIAO W. etc. *IAMC Architecture and Prototyping: A Progress Report*. Proceedings of ACM ISSAC'01, University of Western Ontario, London, Ontario, Canada, July 22-25, 2001.
- [18] WANG, P. S. *Design and Protocol for Internet Accessible Mathematical Computation*. In Proc. ISSAC'99 (1999), ACM Press, pp. 291-298.
- [19] *Workshop on the Future of Mathematical Communication*. Dec., 1999. <http://www.msri.org/activities/events/9900/fmc99/>.
- [20] YOUNG, D. and WANG, P. S. *GI/S: A Graphical User Interface for Symbolic Computation Systems*. In Journal of Symbolic Computation. Vol 4. No. 3. 1987. pp. 365-380.