Specification of OMEI V1.0: Open Mathematical Engine Interface

Institute for Computational Mathematics
Department of 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. The objective of OMEI is to provide a group of generic and abstract function prototypes with well-specified semantics. The function prototypes are specified in a C-style syntax to maximize its acceptance. The OMEI as a standard proposal can play 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.*

This technical report gives a detailed description for OMEI function prototypes. For each function prototype we give a generic description, followde by the formal definition of the function prototype and the parameters.

To put OMEI into practical use, the OMEI specification has to be mapped onto a specific language. This technical report also introduces JMEI (Java Mathematical Engine Interface), an Java language mapping for OMEI, and demonstrates the implementation strategies for JMEI drivers.

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

Contents

Chapter 1

Introduction

1.1 What is OMEI?

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 CLI (Call-Level Interface) in database area, which is the foundation for Microsoft's *Open Database Connectivity* (ODBC) and Sun Microsystem's *Java Database Connectivity* (JDBC).

1.2 Motivations

The motivations for this work come 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 mathetmatical 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 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.

1.3 Programmer's View of OMEI

To the programmer, OMEI is a set of function prototypes that allow applications to exploit the compuational capabilities of mathematical software. OMEI provides the standardized pattern for an application to send encoded expressions or other commands to compute engines, and fetch results afterwards.

A typical pattern to access a compute engine consists of openning a connection to the engine, check the capabilities and supported encodings from the connected engine, set the intended encoding format for the result, send the computational commands to the engines, and fetch the result. Encapsulating this pattern into a set of functions will increase the portabily of applications.

1.4 Implementor's View of OMEI

For implementors who want to provide an Application Programming Interface (API) for either new compute engines or existing engines, OMEI provides a standard that they can refer to in their interface design. They don't have to re-invent the wheel and design their API from scratch. The only thing they need to do is to map OMEI function calls to their specific programming language, and implement this specific language mapping based on the platform which their compute engine is running over.

OMEI can not only facilitate the development of compute engine APIs, but also increase the accessibility of these APIs. The OMEI function prototypes have been specified in a way that they can be easily mapped to most mainstream programming languages. Moreover, the APIs defined based on OMEI will follow similar pattern so that the users "learning curve" is improved.

Chapter 2

Architecture and Framework

Before we go into details of the OMEI specification, let's overview the framework within which we expect to use OMEI. This section we first describe the architecture of distributed mathematical systems based on OMEI. We then will present the application framework of OMEI.

2.1 Distirbuted Mathematical Computation: A Layered Architecture

The layered architecture for distributed mathematical computation based on OMEI is shown in Figure 2.1. The distributed mathematical computation in general has been a research topic for a long time. Researchers worldwide have made contributions in both the encoding level and user interface level. Some well-known projects are OpenMath [1], MathML [4] and MP [3] in the encoding level, and CAS/PI [6], SUI [2] and GI/S [20] in the user interface level. Nevertheless, few efforts can be seen in the application programming interface level. The OMEI can be viewed as an attempt in this aspect.

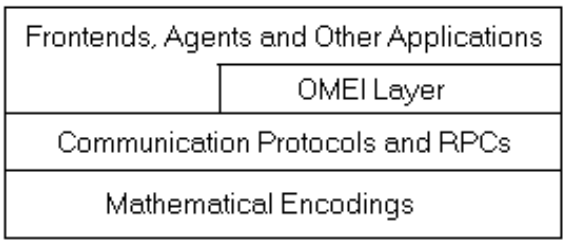

Figure 2.1: Distributed Mathematical Computation Architecture

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.

2.2 OMEI Application Framework

The application framework for OMEI is depicted in Figure 2.2.

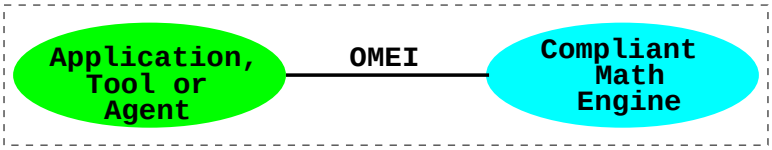

Figure 2.2: Interfacing Using OMEI

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 2.3 shows the application framework in this situation.

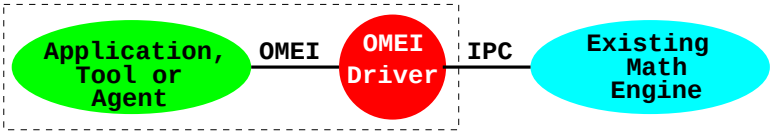

Figure 2.3: Interfacing with an OMEI Driver

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.

Chapter 3

OMEI Function Prototypes

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 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.

3.1.1 OMEIAllocConnect

Function Prototype

```
OMEIRETURN OMEIAllocConnect(
     OMEICONHANDLE *ech /*pointer to 32 bit output, a henv */
     );
```

Description
OMEIAllocConnect allocates data structure for a specific connection. This function is called before before a Connection is established.

Parameters
The function *OMEIAllocConnect* comes with only one parameter: **ech*. *ech* is the connection handler that points to data structure where the related information for a connection is stored.

- *ech:* Engine connection handle.

Examples

3.1.2 OMEIConnect

Function Prototype

```
OMEIRETURN OMEIConnect(
 OMEICONHANDLE ech,
 String engineURL,
 String userid,
 String passwd
 );
```

Description
The function *OMEIConnect* establishes a connection to the specified compute engine, using the pre-allocated data structure *ech* to save any related information.

Parameters
The function *OMEIConnect* has four parameters:

- *ech:* The preallocated connection handle.
- *engineURL:* The URL that uniquely identifies the engine.
- *userid:* The user name.
- *passwd:* The password for the specified user.

Examples

3.1.3 OMEIDisconnect

Function Prototype

```
OMEIRETURN OMEIDisconnect(
OMEICONHANDLE ech
);
```

Description
Close the connection as specified in the connection handle *ech*. This function doesn't release the information and resources allocated for this connection. A previously closed connection can be reopened, but it has no guarantee that the status will be restored, which depends on the implementation.

Parameter:

- *ech:* Engine connection handle.

Examples

3.1.4 OMEIFreeConnect

Function Prototype

```
OMEIRETURN OMEIFreeConnect(
OMEICONHANDLE ech
);
```

Description
If the connection has not been closed, close the connection. Then destroy the connection handle allocated when the *OMEIAllocConnect* was invoked. Release all the resources assigned for the connection.

Parameter:

- *ech:* Engine connection handle.

Examples

3.2 Query Engine Capabilities

A successfully established connection is required for OMEI applications to to interact with the compute engine. In order to submit the actual computation requests to the engine over the connection, the application may need to *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 for connected engine;
- *HelpOnCommand* information for each individual command;
- *Command template* that gives the syntax of an individual command
- *Mathematical encodings* supported;

Correspondingly, OMEI provides four function prototypes for query operations: `OMEIGETCanDo()`, `OMEIGetHelpOnCommand()`, `OMEIGetTemplateOnCommand()`, and `OMEIGetSupportedEncodings()`.

3.2.1 OMEIGetCanDo

Function Prototype

```
OMEIRETURN OMEIGetCanDo(
OMEICONHANDLE ech,
String canDoList
)
```

Description
OMEIGetCanDO returns a string that represents the command names the connected engine can provide. The command name string ends with a period(".") and individual commands are seperated by a semicolon ";". Forexample, if the connected compute engine can provide and can only provide computations "Integrate" and "Factor", the OMEIGetCanDo will get a CanDo string "Integrate;Factor.".

Parameter:

- *ech:* Engine connection handle.
- *canDoList:* The CanDo string.

Examples

3.2.2 OMEIGetHelp

Function Prototype

```
OMEIRETURN OMEIGetHelp(
OMEICONHANDLE ech,
String helpInfo
);
```

Description
OMEIGetHelp retrives help information that usually specifies what the engine can do, how to perform computations over the engine, and etc.

Parameter:

- *ech:* Engine connection handle.
- *helpInfo:* The help information in plain text format.

Examples

3.2.3 OMEIGetHelpOnCommand

Function Prototype

```
OMEIRETURN OMEIGetHelpOnCommand(
OMEICONHANDLE ech,
String command,
String helpInfo
);
```

Description
OMEIGetHelpOnCommand checks help information for the specified command. If no help can be retrived about the given command, or any other error occurs, a *-1* will be returned. The help information is saved in the parameter *helpInfo*.

Parameter:

- *ech:* Engine connection handle.
- *command:* The command we intend to get help information.
- *helpInfo:* The help information in plain text format.

Examples

3.2.4 OMEIGetTemplate

Function Prototype

```
OMEIRETURN OMEIGetTemplate(
OMEICONHANDLE ech,
String command,
String template
);
```

Description
OMEIGetTemplate fetches the template information for the given commnad. The template is XML-encoded, structured text information. The *Template* can be viewed as a special type of help information for the given command.

Parameter:

- *ech:* Engine connection handle.
- *command:* The command we intend to get *Template* information.
- *template:* The *template* data in XML format.

Examples

3.2.5 OMEIInputEncodings

Function Prototype

```
OMEIRETURN OMEISupportedEncodings(
OMEICONHANDLE ech,
OMEISMALLINT encodingValue,
OMEIBOOLEAN supportedOrNot
);
```

Description
This function checks whether the connected engine supports the standard encodings, including MathML Presentation, MathML Content, OpenMath and MP. The *encodingValue* has 4 possible values.

- *MATHMLPRESENTATION(0):* MathML Presentation encoding, as proposed by W3C.
- *MATHMLCONTENT(1):* MathML content encoding. This is also proposed by W3C.
- *OPENMATH(2):* OpenMath encoding.
- *MP(3):* MP encoding.

Parameter:

- *ech:* Engine connection handle.
- *encodingValue:* The mainstream standard mathematical encoding formats.
- *supportedOrNot:* *true* or *false* representing the corresponding encoding format is supported or not.

Examples

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.

3.3.1 OMEIAllocCommand

Function Prototype

```
OMEIRETURN OMEIAllocCommand(
OMEICMDHANDLE *cmdh
);
```

Description
OMEIAllocCommand allocates a handle for the execution of a command. This handle represents the execution environment for a specific command. If there is enough resources for command execution, the *OMEIAllocCommand* allocates and reserves the resources in the format of a command handle. Otherwise, or anything unusual happens, a *null* will be returned.
Parameter:

- *cmdh:* The command handle to be allocated. A *null* value will be returned if for some reason the command handle is not allocated sucessfully.

Examples

3.3.2 OMEICreateCommand

Function Prototype

```
OMEIRETURN OMEIStringCommand(
OMEICMDHANDLE cmdh,
int encodingFormat,
String commandString
);
```

Description
The *OMEIStringCommand* generates a command using the created command handle and the specified command string and its encoding format. Basically, the function call will convert the encoded command string into an internal representation that is suitable for the compute engine to execute. This internal representation of the command will be saved in the command handle.
Parameter:

- *cmdh:* The previously allocated command handle.
- *encodingFormat:* The encoding format for the command string.
- *commandString:* The command string.

Examples

3.3.3 OMEIFileCommand

Function Prototype

```
OMEIRETURN OMEIFileCommand(
OMEICMDHANDLE cmdh,
int encodingFormat,
File *file
);
```

Description
Similar to *OMEIStringCommand*, the *OMEIFileCommand* generates the command using the created command handle and specified encoding format. Unlike the *OMEIStringCommand*, the *OMEIFileCommand* generates the internal command by reading the encoded data from the file.
Parameter:

- *cmdh:* The previously allocated command handle.
- *encodingFormat:* The encoding format for the command string.
- *file:* The File stream.

Examples

3.3.4 OMEINativeCommand

Function Prototype

```
OMEIRETURN OMEINativeCommand(
OMEICMDHANDLE cmdh,
String nativeCommand
);
```

Description
The *OMEINativeCommand* creates a internal command using a format that is "native" to the connected compute engine. This function call is for some advanced users who are familar with the connected engine and its command syntax.
Parameter:

- *cmdh:* The previously allocated command handle.
- *nativeCommand:* The native command string.

Examples

3.3.5 OMEIPutCommandName

Function Prototype

```
OMEIRETURN OMEIPutCommandName(
OMEICMDHANDLE cmdh,
String commandName
);
```

Description
The *OMEIPutCommandName* call, combined with the subsequent *OMEIPutParameter* call, give yet another way to create a command. In this way, the command is created by assigning the command name first, then adding the parameters one by one.
Parameter:

- *cmdh:* The previously allocated command handle.
- *commandName:* The command name string.

Examples

3.3.6 OMEIPutParameter

Function Prototype

```
OMEIRETURN OMEIPutParameter(
OMEICMDHANDLE cmdh,
int paraType,
OMEIPointer *value
);
```

Description
The *OMEIPutParameter* call is usuall used after *OMEIPutCommandName* to add parameter values for creating an internal command. In the case that a command has more than one parameter, the *OMEIPutParameter* may need to invoked multiple times to add these parameter values.
Parameter:

- *cmdh:* The previously allocated command handle.
- *paraType:* The parameter type.
- **value:* The parameter value.

Examples

3.3.7 OMEIEndCommand

Function Prototype

```
OMEIRETURN OMEIEndCommand(
OMEICMDHANDLE cmdh
);
```

Description
The *OMEIEndCommand* function call explicitly terminates the creation of a command. When used with *OMEIPutParameter* calls, the *OMEIEndCommand* will wrap up the command creation using the command name and parameter values as already provided; when used with *OMEIStringCommand* or *OMEIFileCommand*, the *OMEIEndCommand* does nothing.

Note that there is no corresponding call to begin the creation of a command. The function *OMEIPutCommandName* implicitly begins the creation of a command.
Parameter:

- *cmdh:* The previously allocated command handle.

Examples

3.3.8 OMEIFreeCMDHandle

Function Prototype

```
OMEIRETURN OMEIFreeCMDHandle(
OMEICMDHANDLE cmdh
);
```

Description
The *OMEIFreeCMDHandle* releases the specified command handle and frees all the resources allocated to it.
Parameter:

- *cmdh:* The previously allocated command handle.

Examples

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`– is 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.

3.4.1 OMEIExecuteCommand

Function Prototype

```
OMEIRETURN OMEIExecuteCommand(
OMEICONHANDLE conh,
OMEICMDHANDLE cmdh,
int encoding
);
```

Description
As implied in its name, the *OMEIExecuteCommand* function call schedules the execution of the created command. This is an asynchronous call which means a call to *OMEIExecuteCommand* will return immediately without waiting for the completion of the command.
Parameter:

- *conh:* The previously allocated connection handle.
- *cmdh:* The previously allocated command handle.
- *encoding:* The intended encoding format for the execution result.

Examples

3.4.2 OMEIWaitFor

Function Prototype

```
OMEIRETURN OMEIWaitFor(
OMEICMDHANDLE cmdh;
boolean &dialogRequest
);
```

Description
The *OMEIWaitFor* waits for the completion of the execution of a command, or the halt of the command execution due to the pending dialog request. In other words, the *OMEIWaitFor* is a blocking call and it returns only in two conditions: 1). the completion of the command execution; 2). the execution halts to wait for dialog data.

If the return of *OMEIWaitFor* call is for dialog data, it can be invoked again after the dialog data is feeded.
Parameter:

- *cmdh:* The previously allocated command handle.
- *dialogRequest*: If the return of the current call is for the dialog data, *dialogRequest* will be true. Otherwise, it will be false.

3.4.3 OMEICancel

Function Prototype

```
OMEIRETURN OMEICancel(
OMEICMDHANDLE cmdh
);
```

Description
The *OMEICancel* function call cancels the execution of a command. It is usually used to cancel an abnormal execution, such as the execution that takes too much time already.
Parameter:

- *cmdh:* The previously allocated command handle.

Examples

3.4.4 OMEIQueryDialogType

Function Prototype

```
   OMEIRETURN OMEIQueryDialogType(
    OMEICMDHANDLE cmdh;
int &type;
    );
```

Description
The *OMEIQueryDialogType* call is invoked after the *OMEIWaitFor* returns with a dialog request. This function call checks the data type for the pending dialog request so that the application can put data in the correct type.
Parameter:

- *cmdh:* The previously allocated command handle.
- *type*: The data type in the Integer value. INTEGER(0): Integer; FLOAT(1): Float; CHAR(2): Char STRING(3): String;

Examples

3.4.5 OMEIPutDialogData

Function Prototype

```
    OMEIRETURN OMEIPutDialogData(
     OMEICMDHANDLE cmdh,
     OMEIPOINTER *value
     );
```

Description
The *OMEIPutDialogData* sends the data to the connected compute engine upon a dialog request. The data to be sent must be the required type. The *OMEIQueryDataType* function call can be used to query the requested data type.
Parameter:

- *cmdh:* The previously allocated command handle.
- *value*: The data value to be sent.

Examples

3.4.6 OMEIGetResult

Function Prototype

```
OMEIRETURN OMEIGetResult(
 OMEICMDHANDLE cmdh,
 int resultType,
 OMEIPOINTER *result
 );
```

Description
The *OMEIGetResult* function call retrieves the result for a delivered command execution. It is one of the most common used functions. If the result is retrieved successfully, the *OMEIGetResult* returns 0 and the result will be put in the parameter *result*. Otherwise, it returns -1.
Parameter:

- *cmdh:* The previously allocated command handle.
- *resultType*: The data type for the retrieved result.
- *result*: The retrieved result.

Examples

Chapter 4

A Sample Program

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. The following code shows how to perform the simple computation *integrate(sin(x), x)* in C programming language.

```
   #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 | engineURL, userName, passwd);

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

10)   OMEIExecuteCommand(mc, cmd, MATHML_PRESENTATION);

11)   OMEIGetResult(mc, cmd, mathMLResult);

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

The above code carry out these steps:

Connect to Compute Engine(Lines 6-7)

As shown in the code , a connection to the compute engine is established using the parameters *mc, engineName, userName, passwd.*

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

In the first call, OMEIAllocConnect, a command handle *mc is allocated to save the environment information for the connection to be established.In the second call,* OMEIConnect, the engine is uniquely identified by the parameter *engineName or engineURL . The last two paramenters in this call,* userName and passwd, are used to identify the authorized user for this specific engine.

Create a Command(Lines 8-9)

To create a command, we have to get a command handle to save information for the execution of the command. The command handle is allocated using function call *OMEIAllocCommand(). Once the command handle is allocated, the command can be created by calling* OMEICreateCommand and specifying the encoding format and data content.

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

Execute Command(Line 10)

When the create command is submitted to the connected engine for execution, the intended encoding for the computation has to be specified. The function prototype *OMEIExecuteCommand hence has three parameters, as shown in the following code.*

```
10)   OMEIExecuteCommand(mc, cmd, MATHML_PRESENTATION);
```

Get Result(Line 11)

Now it is time to get the execution result.

```
11)   OMEIGetResult(mc, cmd, mathMLResult);
```

Cleaning Up(Lines 12-14)

Finally we close the connection and free the allocated resources.

```
12)   OMEIDisconnect(mc);
13)   OMEIFreeCMDHandle(cmd);
14)   OMEIFreeCONHandle(mc);
```

Chapter 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.*

5.1 JMEI: OMEI's Java Language Mapping

Similar to JDBC, JMEI consists of JMEI Driver API and DriverManager specification. The JMEI Driver API defines the Java interfaces for building JMEI drivers, the dynamically loadable Java modules that encapsulate and expose the capabilities of compute engines. DriverManager is a tool to aid the management of multiple JMEI drivers.

JMEI API specification is designed after studying Java language features and the scenario on how the Isv accesses compute engines. First, an Isv may request computational services from multiple engines, which means it should be allowed for the Isv to load multiple JMEI drivers. The DriverManager class is thus defined to simplify the management of multiple drivers. Second, to be able to utilize an engine's capabilities exposed by its JMEI driver, a Connection must be established. Over the established connection, the Isv can create and submit Commands to the compute engine and fetch Results. In some cases, the Isv may request information services to check capabilities and command usages.

The core classes and interface can therefore be outlined from the above scenario: DriverManager, Driver, DriverInfo, Connection, Command and Result. The relationship among these classes and interfaces is illustrated in Figure 3.

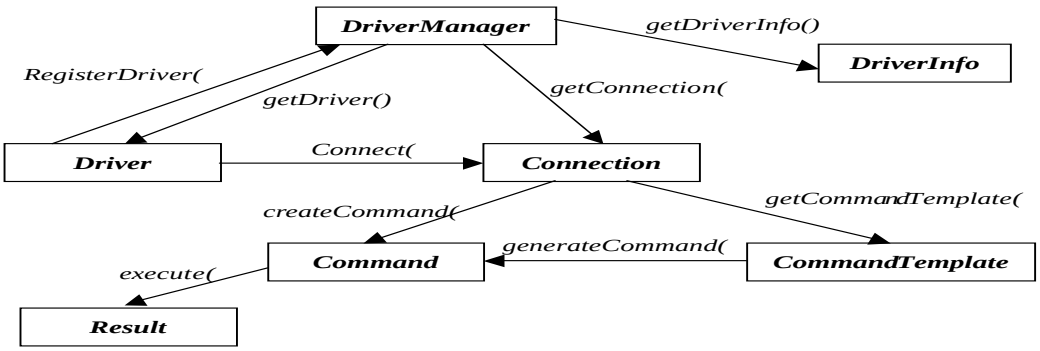

Figure 5.1: Relationship among JMEI Interfaces and Classes

5.1.1 JMEI Driver API

The main interfaces and classes in JMEI Driver API are described as follows.

DriverManager *Class*
The DriverManager class manages multiple JMEI drivers and dispenses Connection objects. As part of initialization, the class will locate and load the JMEI drivers the user specifies in the jeei.drivers property. A newly loaded driver class calls registerDriver() to make itself known to the DriverManager, and deregisterDriver() to remove itself from the driver list. The getDriver() method in DriverManager class locates a driver that understands the given JMEI URL. The getConnection() method establishes a connection to the given JMEI URL.

```
public class DriverManager {
  public static synchronized Connection getConnection(String engineName);
  public static Driver getDriverByName(String engName);
  public static Enumeration getDrivers();
  public static synchronized void registerDriver(Driver driver);
  public static void deregisterDriver(Driver driver);
  public static DriverInfo getDriverInfo(String engName);
}
```

Driver *Interface*
The Driver is an interface that every JCEC driver must implement. Once a driver is loaded, it creates an instance of itself and then registers it with the DriverManager.

```
public interface Driver {
  public abstract Connection connect(String engUrl, String uid, String
passwd);
  public abstract String [] getPropertyInfo(String engName);
  public abstract Boolean isJMEICompliant();
  public abstract int getVersion();
}
```

Connection *Interface*
A Connection object represents a session with a specific compute engine. It provides a context for executing computational commands and processing their results.

```
public interface Connection {
  public String getCanDo();
  public Help getHelpOnCommand(String command);
  public CommandTemplate getCommandTemplate(String command);
  public String getEncodingsSupported();
  public Command createCommand(int encodingFormat, String command);
  public Command createCMDFromMPStream(InputStream mpStream);
  public void close();
}
```

Command *Interface*
A Command object is used for executing a computational command and obtaining the results produced by it. Optionally, it can also be used to execute a batch of commands and fetch result set in one time.

```
public interface Command {
  public int execute();
  public int executeNative();
  public int putDialogData(int seqNo, Object value);
  public Result getResult();
  public int cancel();
  public Connection getConnection();
}
```

Result *Interface*
The Result object provides methods to access the result of a command execution.

```
public interface Result {
  public int getType();
  public String getString();
  public String getMathMLString();
  public String getOpenMathString();
  public Integer getInteger();
  public Float getFloat();
  public Double getDouble();
  public InputStream getAsciiStream();
  public InputStream getBinaryStream();
  public InputStream getDoubleStream();
  public InputStream getMPStream();
}
```

DriverInfo *Interface*
The DriverInfo object provides information services, including the supported commands, command usage, and etc.

```
public interface DriverInfo {
  public String getCanDo();
  public Help getHelpInfo(String command);
  public Template getTemplate(String command);
  public String getEncodingsSupported();
  boolean supportsMPInput();
  boolean supportsMathMLPresInput();
  boolean supportsMathMLContenetInput();

}
```

5.1.2 JMEI URL: Locate JMEI Driver Over Networking Environment

The IAMC server and compute engine are usually reside on the same host and communicate with each other through interprocess communication mechanism (IPC), or reside on cluster environment and interoperate through high performance network. The IAMC server doesn't talk to compute engine directly. Whereas, it interoperates with a JMEI DriverManager object, which further interoperates with JMEI Driver instance. One can envision that it is JMEI Driver that exposes capabilities of a compute engine to IAMC servers.

Under the above situation, one major concern is how to uniquely address a JMEI driver, or, in other word, the compute engine encapsulated by this driver. Like JDBC, JMEI employs Unified Resource Locator (URL) syntax to identify a compute engine in networking environment. The following naming scheme described a JMEI URL:

JMEI://domain name:port/engine id

For example, suppose we have a Maxima package installed on horse.mcs.kent.edu and the port for JMEI Maxima driver is 5999. Then this compute engine will have a URL:

JMEI://horse.mcs.kent.edu:5999/Maxima

5.2 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 5.2 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 5.3 shows this approach.

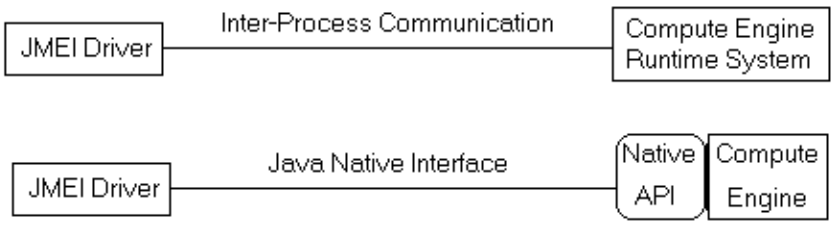

Figure 5.2: Implement JMEI Driver through IPC or JNI

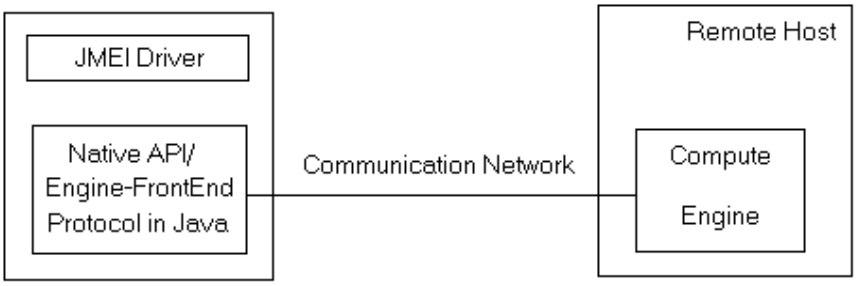

Figure 5.3: Implement a Network-Enabled JMEI Driver

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.

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 mush easier.

5.3 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 5.4. 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.

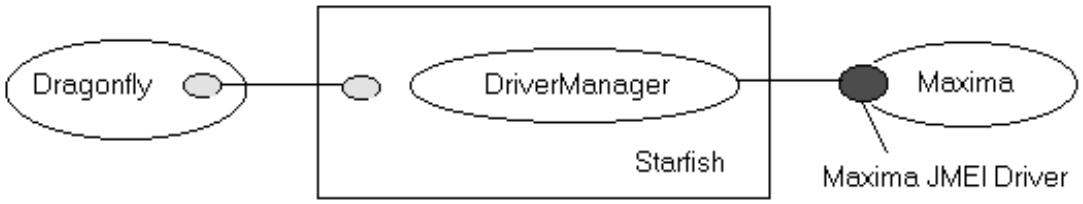

Figure 5.4: IAMC Prototyping Structure

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;
```

Chapter 6

Conclusions

This technical report specifies OMEI: Open Mathematical Engine Interface. OMEI is an effort that intends to establish a uniform API for mathematical compute engines. A unified API can no doubt facilitate the development of mathematical applications, espeically distributed/parallel mathematical application that make use of heterogeneous compute engines.

Although we specify OMEI function prototypes in C syntax, OMEI is intended to be language neutral. We try to specify these prototypes in a way that they can be mapped to other programming languages as easily as possilbe. As an example, this technical report also investigates our OMEI's Java language mapping, the Jave Mathematical Engine Interface (JMEI). Some implementations details are also mentioned.

OMEI has been employed as an important component in IAMC (Internet Accessible Mathematical Computation) Framework. IAMC Framework is an ongoing project occuring here in ICM/Kent. OMEI is used in IAMC Framework to connect the IAMC server prototype, Starfish, and compute engines including Maxima and Mathematica. 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.

Bibliography

[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.*