Metainformation Engine v1.0

This document details the architecture of the Metainformation Engine v1.0

This entire document is proprietary technical data, which DMI requests not be released to anyone without written permission from the authors.

Anirban Bhaumik

Dr. Michael Bieber

Jan 13th 2003
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PRIMARY COMPONENTS</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRTUAL DOCUMENT</td>
<td>4</td>
</tr>
<tr>
<td><strong>A Frame</strong></td>
<td>5</td>
</tr>
<tr>
<td>ME BROKER (MEB)</td>
<td>7</td>
</tr>
<tr>
<td><strong>Traversal Path Manager (TPM)</strong></td>
<td>7</td>
</tr>
<tr>
<td>MAPPING RULES ENGINE (MEMRE)</td>
<td>8</td>
</tr>
<tr>
<td><strong>Mapping Rules</strong></td>
<td>8</td>
</tr>
<tr>
<td>ENGINE DESKTOP</td>
<td>10</td>
</tr>
<tr>
<td>METAINFORMATION APPLICATION WRAPPERS (MAWS)</td>
<td>11</td>
</tr>
<tr>
<td>MESSAGE FLOW IN THE ME</td>
<td>12</td>
</tr>
</tbody>
</table>
Metainformation Engine Architecture

The Metainformation Engine (ME) is a loosely coupled system, where various engine components communicate with each other via messages that conform to a well-defined standardized internal protocol. This approach allows new engine components to be developed and added without affecting existing engine components and functionality. The ME’s goal is to supplement the output of most computer applications with link anchors and lists of links for each anchor, all with minimal or no changes to them. The ME will serve any application and user interface (e.g., Web browsers) that has implemented an appropriate wrapper. To integrate a new application or type of interfaces, it is necessary to develop a wrapper (i.e., identify elements, mapping rules, metadata, etc.) for it. Thus, to supplement the output of an application, the developer only has to develop and register the wrapper. This may prove straightforward or complex as we describe later. But in any case it only must be done one time for a specific application to apply to any instance of that application.

**Primary Components**

![Diagram of ME components]

The ME v 1.0 consists of three primary components:

- The Engine Desktop translates the displayable portion of the ME’s internal messages, from the standard internal XML format to a format that can be displayed to a user via a web browser and vice versa.
o The ME Broker enables the communication between the Engine modules and works as the router for all the internal messages. All Engine messages pass through the Broker, which then redirects them to the appropriate engine component.

o The Mapping Rules Engine (MEMRE) maps the application data and relationships to hyperlinks at run-time. MEMRE maps the element instances in the application’s output to the global element types (classes), and finds the links for them. Once the links are produced they are sent through to the engine desktop to be displayed in an appropriate interface to the user.

All engine components that are capable of processing a ME message are known as enginelets. All application wrappers (also known as Metainformation Application Wrapper or MAW) are enginelets. Enginelets can also be used to log intermediate messages, process for accuracy, provide filtering services etc. In order to provide supplemental hypermedia functionality for an application (add links and other metainformation) the ME runtime instance must also contain one or more enginelets, one (or more) of which is a Metainformation Application Wrapper (MAW).

A MAW manages the communication between the ME and the application system, translates the user requests from the ME’s internal format to a format the application can process, receives the output from the application and converts it to the ME format. The MAW also identifies and marks the “elements of interest” within the application’s output to which hyperlinks and other metainformation are mapped.

**VIRTUAL DOCUMENT**

Any discussion of the architecture and information flow in the ME must be preceded by a discussion of the communication protocol between the various engine components. This communication protocol is the Virtual Document schema.

Figure 2: Virtual Document Schema
- **ID**: Uniquely identifies the Document. The value of this ID is the timestamp or milliseconds since epoch (January 1, 1970) that this document was first generated. The Virtual Document creation mechanism ensures that this ID is unique for each document created in a specific instance of the ME.

- **UserName**: The name of the user who is logged in to the ME and whose actions generated this Virtual Document instance.

- **Source**: The name of the enginelet that generated this document.

- **Processor**: The name of the enginelet that is the final destination for this document.

- **Path**: Defines the traversal paths that will or have processed this document using the following sub-elements:
  - **EngineLet**: This is the name of the enginelet that is in the traversal path.
  - **isProcessed**: Whether or not the specified enginelet has processed this document, the value is yes if that is true.
  - **RequestContext**: Models the entire HTTP request that is created by the user in following a link or submitting a form.
  - **Response**: Models the response generated by the destination (and optionally modified by intermediate enginelets in the traversal path). This element contains the following sub-element:
  - **FrameGroup**: A FrameGroup loosely models a “window” in a traditional user interface. In a web-browser this is a specific browser instance. A FrameGroup contains one or more Frames that implement a document to be displayed to the user.

**A Frame**

A Frame represents a section of a document that has the same context and typically has been generated by the same enginelet. From a UI perspective a Frame loosely corresponds to an HTML frameset. A Frame follows the following schema:
The Frame consists of the following sub-elements:

1. **Document**: This models the actual screen or document generated by the application that the user is interacting with. The ME’s user interface layer, the ME desktop, supports an XHTML document, an XML document following an arbitrary schema as specified by the enginelet that generated this document, or any arbitrary object that can be rendered by the Lens.

2. **Metainformation**: This element models the metadata and relationships for different objects in the ME.
   - **DocumentMI**: Models the metainformation at the document level. Currently unused by the ME.
   - **ElementMI**: Models metainformation about elements of interest.
   - **LexicalMI**: Models metainformation for pre-determined lexical values. The lexical value may be, or not, an instance of an element. Currently unused by the ME.
   - **SystemMI**: Models metainformation for the entire engine instance and all applications being wrapped by the engine. Currently unused by the ME.

All metainformation objects extend the type MIType (Metainformation type). The MIType class contains two sub-elements:

3. **Locator**: The XPATH expression that defines the location of the object or element of interest that this metainformation is about. This XPATH expression must evaluate to a node in the document object.
**ME Broker (MEB)**

The main objective of the MEB is to enable and manage communication between different enginelets. The MEB will receive a message, determine where to route it next and pass it on to the appropriate component. In order to maintain high availability, the MEB instantiates each enginelet named in a registry and pools them in an object pool for reuse. Enginelets provide services and often connect to stateful applications, it maybe expensive (computing resources wise) to re-instantiate and reconnect to these applications. In order to alleviate this issue the ME instantiates a fixed number of instances of an enginelet at startup, persists these an object pool and reuses them as and when needed.

The MEB performs the following functions:

- Maintains a registry of enginelets. This is maintained in an XML file named “enginelets.xml” that defines the class that implements the enginelet, the number of enginelets that must be pooled, the startup order and the unique name of the enginelet. The enginelets.xml file must be available to the ME at runtime.

- Instantiates the enginelets named in the enginelet registry in the order specified by their startup order.

- Pools the enginelets.

- Receives messages from different enginelets.

- Sends messages to other enginelets.

- Sends and receives messages simultaneously.

- In order to find any intermediate enginelets that must process messages between the source and destination enginelets, the MEB delegates to the Traversal Path Manager (TPM) the task of finding the next enginelet.

**Traversal Path Manager (TPM)**

The TPM is a component of the MEB that determines the intermediate enginelets that must process a message from a specific enginelet before it is processed by the destination...
Intermediate enginelets can alter the document or message, or update internal status parameters or logs. For every pair of source and final destination enginelets a list of intermediate enginelets exists. This list is known as the traversal path of an engine message. This Traversal Path for every pair of source and destination is stored in the traversal path registry (the file traversalpaths.xml), which must be accessible at runtime to the ME.

The source enginelet thus decides the final destination of the message and not the intermediate points. Based on the entries for the pair of source and destinations, the TPM obtains the traversal path and hands this to the MEB. The MEB then routes the message to each enginelet in the traversal path in the order listed. Should an entry for a pair of messages between a specific source and destination not exist, then the traversal path is empty and no intermediate enginelets process the message and the MEB simply routes the message directly to the destination.

**Mapping Rules Engine (MEMRE)**

The MEMRE provides the core functionality of the engine i.e., it identifies a class of elements and provides hyperlinks for each instance of that class. For every element with at least one resolved link, MEMRE places a link anchor to be displayed with that element. In order to generate these links the MEMRE maintains a mapping rule registry (the file mappingrules.xml) that must be available at runtime to the ME.

**Mapping Rules**

There are three kinds of basic hypertext objects in the ME: nodes, links and anchors. Nodes are documents containing elements of interest. Links represent relationships among two or more nodes. Thus nodes are the endpoints of links.

In the ME context, a node or ‘document’ is the dynamically generated output of an enginelet or application that should be displayed by a User Interface. Physically, some instances of the elements that can be considered as ‘elements of interest’ in a document are: a character or set of adjacent characters, a word or set of adjacent words, a phrase or set of adjacent phrases, a paragraph or set of adjacent paragraphs, a table, a figure, an image, etc., as well as any adjacent combination of these, including the document as a whole.

A mapping rule bridges two different domains, i.e.; it takes an object and “maps” it from one domain to another [Bieber & Kimbrough 1992]. Mapping rules can connect the same object, such as the same person or account. They also can connect related elements (such as employees working on the same project) or characteristics of an element (such as an employee and his or her address). The connected objects can be in the same or different applications. Thus a user may follow a link from an object in a specific application to the related element another in a completely different application. Often the destination application will need to execute a command (e.g., a query) to generate the destination. 
An element. This command will be associated with the link and executed in the process of navigating or “traversing” it.

After a MAW locates the “elements of interest” in a document it also creates a Relationship object for each instance of an “element of interest” and adds it to the Metainformation node in the Frame containing the document that is to be displayed to the user. A Relationship object is a set of Mapping Rules that apply to a given element, it is also referred to as list of links.

Mapping rules provide a mechanism to add commands to a link. When the user selects a link from the list of links generated, the mapping rule defines the command(s) for that link and the enginelet that will execute that command. A Virtual Document object is thus generated by the Desktop that sends this request to the enginelet that then generates the appropriate response.

A Relationship object follows the following schema:

- **SemanticType**: This is the type of the element of interest for which a Mapping Rule applies. The MAW specifies this when it identifies an “element of interest” while parsing an application document for display.

- **CommandSet**: The set of commands to be executed when a user follows a link. Contains the following sub-elements:
  - **Command**: The commands to be executed by the destination enginelet (or its application).
  - **Condition**: The fully qualified class name of the condition for a mapping rule to be valid and available as a link to a user. Conditions are executed
by the MEMRE at runtime to decide whether to include this link given the current system state.

- **URI**: The URI of this command set if any. Currently unused by the ME.
- **ParameterList**: Is a set of name of value pairs that will be added to the query string added to the displayable link. The value of these parameters is computed from the XPATH expression in the Locator object or on the basis of existing values “hard-coded” in the mapping rules registry.
- **CommandProcessor**: The name of the enginelet that will execute this command.
- **DisplayLabel**: The text to be on the link when displayed to the user in the UI.
- **LinkMetadata**: A set of name-value pairs that model the metainformation about the link itself. Currently unused.
- **Target**: This is a UI specific command that specifies the display window of the executed command i.e., the name of the window of the UI the results of the command will be displayed in. This may be used to traverse to links to external systems without leaving the engine or to other information that may need to be displayed in another window.

The MEMRE performs the following functions:

- Receive a message from another enginelet, through the MEB.
- Iterate through the Metainformation objects in each Frame in each Framegroup.
- For each Metainformation object, lookup the mapping rule registered for each semantic type.
- Compute the parameters for this rule.
- Evaluate the condition (if any) associated with this mapping rule. If this condition evaluates to true then this CommandSet is added to the existing Relationship object.

**ENGINE DESKTOP**

The desktop enables communication between the user interface (the so called “desktop” of the user) and the ME.

Flow of events in the desktop:

- The desktop receives an HTTP Request from the user’s browser as a result of the user following a link or submitting a form.
- The desktop creates a Virtual Document object that models this request, with the RequestContext populated with user’s request.
- This Virtual Document is handed over to the MEB for onward processing (generally by an application).
- When the MEB returns the Virtual Document, the desktop iterates through all the CommandSet objects in each FrameGroup in each Frame, and creates a list of
links to be displayed in a popup window when the user clicks on the “show me list of links” icon (link anchor) next to an “element of interest”.

- The document generated by the enginelets is then rendered using the lens specified for a specific Frame. This Frame is embedded in a FrameGroup and displayed in a window.

**Metainformation Application Wrappers (MAWs)**

MAWs must handle the communication between the application and the ME. Either the application or the MAW must provide the following support to integrate with the ME. This list closely resembles compliance sets identified by other OHS researchers [Davis et al. 94].

- **Communicate with the Application**: The ME must intercept the communications between the application and its user interface. Application messages are routed through the MAW to the ME. The engine then can map hypermedia link anchors to it. The more loosely coupled (the more independent the computational and interface portions) and modular an information system is, the simpler the hypermedia integration will be. Communicating with the application also includes the processing of the messages coming from the UI through the ME to comply with the application’s API.

- **Message markup**: The ME must identify application objects in messages in order to determine whether to map hypermedia anchors to each. The ME relies on the MAW to parse each document and screen passed from the application for display. It adds the document or screen to the current message as a Metainformation node. MAW then identifies and marks up each application element by adding a Relationship object to a metainformation node, and adding that metainformation node to the Frame that contains the document generated by the application. The MAW must specify the following:
  - Locator object in the metainformation node. The value of the Locator must be a valid XPATH expression to the “element of interest” in the document that contains it.
  - The semantic type of the “element of interest”.

- **Application developers must develop MAWs**: The person with domain knowledge of the application should provide its MAW’s parsing routines and mapping rules, but only needs to do this one time. Once in place, the mapping rules should map hypermedia to any instance of the applications screens and documents produced in the future, as long as the application structure does not change. If it does, the corresponding MAW parsing routines or mapping rules may need to be updated accordingly. (Application users need have no knowledge of mapping rules. To them the ME’s supplemental linking occurs automatically).

Coding a wrapper is potentially the hardest part of application integration. If the application has an application-programming interface (API), then it usually is quite easy
to parse the output displays and detect the elements within it. If the application provides adequate metadata in tags or other ways (which is becoming increasingly prevalent with XML), parsing can take advantage of these and be straightforward. If documents/screens follow a well-defined template or format, which is the case with many e-commerce systems, then parsing also should be relatively easy. Otherwise, if a document or screen’s content is unstructured and without embedded metadata, then the MAW may have to rely solely on lexical analysis to identify elements of interest within it. Adding lexical analysis is a task for Phase II of this research.

**MESSAGE FLOW IN THE ME**

![Message Flow Diagram](image-url)

Figure 6: Message Flow in the ME
The following events occur in generating an interface to display to the user:

- When a user follows a link to execute an engine command, the user’s browser issues an HTTP request.

- The Engine Desktop generates a Virtual Document modeling the user’s request.

- The MEB reads the source of the document (i.e., Engine Desktop) and the destination (the enginelet that will execute the command) and looks up the traversal path for that message through the TPM. The document is then routed through all intermediate enginelets in the order that they are listed in the traversal path.

- The intermediate enginelets create/edit new Frames (and/or FrameGroups) or process the Virtual Document in some manner.

- After all enginelets in the traversal path have processed the message the message is sent to the destination enginelet - the MAW. The MAW then executes the command by communicating with the application via its native API. It then marks up the elements of interest in the application’s output and adds the output document to a Frame and FrameGroup.

- This Virtual Document is then sent back to the MEB. The MEB again looks up the traversal path for this document, using the MAW as the source and the Desktop as the destination. The Virtual Document is again processed by all intermediate enginelets.

- After all enginelets in the traversal path have processed the message, the MEB passes the message on to the Mapping Rules Engine. The document contains the marked up “elements of interest” that the MAW located.

- The MEMRE looks up the mapping rules for “each element of interest” and generates the list of links. These list of links are added as Relationship objects in the Metainformation nodes of the Frames being manipulated.

- The Virtual Document is returned to the MEB, which then returns it back to its source desktop.

- The desktop uses the specified lens to translate each Frame and display embedded inside a FrameGroup for the user.