



Inter-Domain Data Mapping Using Implicit Bridging

Version 0.8
December 30, 2002

PathMATE Technical Notes

Pathfinder Solutions LLC
33 Commercial Drive, Suite 2
Foxboro, MA 02035 USA
www.PathfinderMDA.com
888-662-7284

Table Of Contents

- 1. Introduction..... 1**
- 2. External Message Interface (EMIF)..... 1**
 - Description1
 - Implementation Strategy1
 - Analysis Properties2
- 3. Example:..... 3**
 - Domain Service.....3
 - Class Instance Creation.....3
 - Sample Generated Dispatcher.....4
- 4. Summary 5**

1. Introduction

This technical note explains the basic strategy for mapping data elements between a realized external message interface domain (EMIF) and analyzed client domains. An implicit bridging strategy is employed, using extended translation properties to specify the connection between message data fields and analysis data atoms.

The technical note summarizes general capabilities independent of any specific application or message set, and is expected to form a starting point for project-specific bridge mapping.

2. External Message Interface (EMIF)

Description

Many embedded systems employ an external message interface domain (EMIF) as an interface between the analyzed application and other external software elements. These messages may come from external software elements in the form of requests for the application to perform some action, or as collections of data being supplied to the application. These requests and data are forwarded by EMIF to the appropriate application domains. This technical note only covers the inbound aspects of this type of mapping – the customer project may extend portions of these bridge mappings symmetrically for outbound messages.

Implementation Strategy

It is assumed that the messages and structures are defined in C/C++ header files, and corresponding message support functions are provided in some kind of library. It is expected that some form of existing or other realized code services required inter-task, inter-process, and/or inter-processor communication.

A common and effective Structured Design for EMIF processing is to separate the message processing into two threads of control. One handles the reception and transmission of the incoming and outgoing messages using realized routines and the other executes within the thread that services its analyzed client domains, providing external-to-internal data structure conversion and the dispatching of calls to client domain services. The actual message buffer is a shared structure, using project-specific (realized) inter-task protection mechanisms.

The routing of data is achieved using a combination of two strategies. One strategy uses client domain services to pass received message data to the client domains, and the other creates instances of client objects and then notifies the client of the creation.

In order to generate the necessary code for the conversion/formatting/dispatching of specific messages, "coloring" properties are used in conjunction with EMIF-specific templates. Client domains wishing to receive notification of, and/or data from an external message can specify a domain service to handle the information contained in the desired external message by an EMIF "coloring" property that specifies the message identifier

that service handles. The parameters of that domain service are colored with properties linking them to the message data items of interest.

In the case of a large set of structured data, it is presumed that EMIF's client domain has modeled a UML class to store such information. This class is "colored" with an EMIF message type property identifying the corresponding message identifier, and the class attributes are tagged with the data items of interest.

To maintain the integrity of data being passed into the application software, the client project can hand-write any data conversion functions needed to change from message units to application units. These functions can be abstracted as EMIF services. The use of these services can be specified directly in the "coloring" used to map the message data field to the analysis data atom (parameter or attribute).

Analysis Properties

There are 3 analysis properties that EMIF clients will use:

EmifRmsgType: (applies to Class and Domain Service)

Correlates a message type with the class to be created, or the service to be invoked.

- Specifies the message type.
- On a class this property indicates that upon receipt of this message type a set of instances of this class will be created (requires **EmifRmsgElementCount** to be set, indicating the number of classes to be created).
- On a domain service this indicates this domain service is to be called when this message type is received.

EmifRmsgElementCount: (applies to Class and Domain Service)

For messages with arrays of data, this identifies the message data field that indicates how many elements are in the message.

- On a class this property indicates the message field containing the number of instances to be created. If count is "" one instance is created.
- On a domain service this property indicates the message field containing the number of times this service is to be called (iterating over a set of parameter values). If count is "" the service is invoked once.

EmifRmsgElementField: (applies to Attribute and Domain Service Parameter)

Correlates a message structure data field with an analysis data atom.

- On a class attribute this indicates the message field that supplies this attribute's value - the actual parameter to the instance create call.

For any class that specifies a **EmifRmsgType**, every attribute requires a value for **EmifRmsgElementField**, or a default value.

- On a domain service parameter this indicates the message field that supplies this parameter's value (the actual parameter to the domain service call).

For any service that specifies a **EmifRmsgType**, every parameter requires a value for **EmifRmsgElementField**.

- This field may or may not contain an inline call to one of the EMIF conversion routines to convert the incoming data type.

3. Example:

The following examples assume an unspecified APP client domain, and various message structures to be received by the EMIF domain.

Domain Service

The APP domain wants to map any updates to the tree farming mode from the MSGTYPE_TFRM_MODE message to a call to the domain service APP:UpdateTreeFarmingMode:

APP:UpdateTreeFarmingMode: *This service selects updates the farming mode.*

in: farming_mode (sys_tree_farming_mode_e): Indicates which type of tree farming mode is in use.

Message Structure for MSGTYPE_TFRM_MODE:

```
// MSGTYPE_TFRM_MODE:
struct emif_message_tree_farming_mode_t
{
    sys_tree_farming_mode_e treefrm_mode
} emif_message_tree_farming_mode_t;
```

The APP:UpdateTreeFarmingMode domain service has the property:

EmifRmsgType = "MSGTYPE_TFRM_MODE"

The farming_mode service parameter has the property:

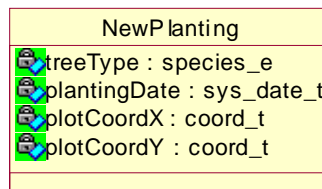
EmifRmsgElementField = "msg_buf->treefrm_mode"

(Please note: use of the base pointer "msg_buf" above is based on an assumption about the implementation of the EMIF message dispatcher.)

Class Instance Creation

In this example, class instances are created and then a domain service is called. An array of message elements is being iterated over, and a conversion service is to map message data fields from one type to another.

Assume the following UML class in the APP domain:



The APP domain would like instances of NewPlanting to be created when the MSGTYPE_NEW_PLANTINGS message is received. However, coordinates are externally conveyed with data fields of type double, which require conversion to the coord_t datatype by the domain service coord_t EMIF:DoubleToCoord(double). Once these instances are created, then the domain service APP:NewPlantingsInPlace() should be called.

Message Structures for MSGTYPE_NEW_PLANTINGS:

```
// Single planting:
struct emif_message_new_planting_t
{
    species_e tree_type;
    sys_date_t planting_date;
    double x, y;
} emif_message_new_planting_t;

// MSGTYPE_NEW_PLANTINGS:
struct emif_message_new_planting_set_t
{
    int planting_count;           // This is the number of parameters actually planted
    struct emif_message_new_planting_t plantings[MAX_PLANTING_GROUP_SIZE];
} emif_message_new_planting_set_t;
```

The APP:NewPlanting class has the properties:

EmifRmsgType = "MSGTYPE_NEW_PLANTINGS "

EmifRmsgElementCount = "msg_buf->planting_count"

Its attributes has the properties:

treeType: EmifRmsgElementField = "msg_buf->plantings[index].tree_type"

plantingDate: EmifRmsgElementField = "msg_buf->plantings[index].planting_date"

plotCoordX: EmifRmsgElementField = "EMIF:DoubleToCoord (msg_buf->plantings[index].x)"

plotCoordY: EmifRmsgElementField = "EMIF:DoubleToCoord (msg_buf->plantings[index].y)"

(Please note: use of the set iteration variable "index" above is based on an assumption about the implementation of the EMIF message dispatcher.)

The APP:NewPlantingsInPlace domain service will have the property:

EmifRmsgType = "MSGTYPE_NEW_PLANTINGS "

Sample Generated Dispatcher

The examples above assume the following realized message definition:

```
struct emif_message_type_t
{
    message_type_e message_type;
    union message_body
    {
        struct emif_message_tree_farming_mode_t mode_message;
        struct emif_message_new_planting_set_t planting_message;
    };
} emif_message_type_t;
```

and result in the generation of the following dispatcher function (C example):

```
bool_t EMIF_DispatchMessage(emif_message_type_t *base_buffer)
{
    bool_t retval = TRUE;

    // Switch the recognized types
    switch (base_buffer->message_type)
    {
        case MSGTYPE_TFRM_MODE:
        {
            // Pull out this message
            struct emif_message_tree_farming_mode_t *msg_buf =
                &(base_buffer->mode_message);

            // Invoke the requested domain service
            APP_UpdateTreeFarmingMode(msg_buf->treefrm_mode);
        }
    }
}
```

```
    }
    break;

    case MSGTYPE_NEW_PLANTINGS:
    {
        int index;
        // Pull out this message
        struct emif_message_new_planting_set_t *msg_buf =
            &(base_buffer->planting_message);

        // Created the requested instances
        for (index = 0; index < msg_buf->planting_count; index++)
        {
            APP_NewPlanting_create(
                msg_buf->plantings[index].tree_type,
                msg_buf->plantings[index].planting_date,
                EMIF:DoubleToCoord (msg_buf->plantings[index].x),
                EMIF:DoubleToCoord (msg_buf->plantings[index].y));
        }

        // Invoke the requested domain service
        APP_NewPlantingsInPlace();
    }
    break;

    default:
        retval = FALSE;
};
return (retval);
}
```

The above dispatcher would be called from realized code whenever an external message is received.

4. Summary

The capabilities outlined in this document are general in nature, and the details of both the property names/values, and the resulting generated dispatcher code is subject to change in response to project need.