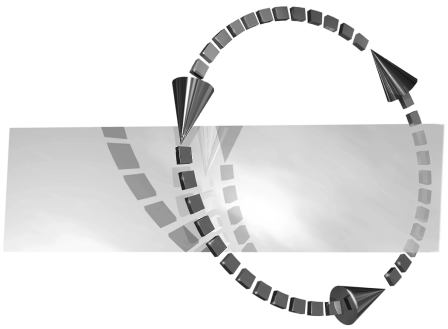

MÄK VR-Link 3.6 Release Notes

This release note provides the following release-specific information for MÄK VR-Link Release 3.6 and 3.6-ngc:

Because DMSO RTI 1.3 NG is not link compatible with the previous generation of DMSO RTI releases, we have (for PCs only) released two versions of the VR-Link – one that works with RTI NG, and one that works with RTI 1.3v*x*. The default directory for the version that supports RTI NG is *vrlink3.6-ngc*. The default directory for the other version is *vrlink3.6*.

Document ID: VRL-3.6-3-001023

Systems Supported

Table 1 lists the platforms currently supported by MÄK VR-Link 3.6. Please contact MÄK if you are interested in purchasing VR-Link for other platforms. Application code must be built with the indicated compilers in order to link to VR-Link libraries.

Table 1: Platforms supported

Platform	Compiler
SGI IRIX6.5 (VR-Link 3.6-ngc only)	MipsPro7.2.1 C++ compiler (available from SGI)
SGI IRIX6.2 or later (VR-Link 3.6 only)	MipsPro7.1 C++ compiler (available from SGI)
Sun Sparcstation with Solaris2.6 or later	SPARCompiler C++ version 4.2 (available from Sun)
PC with Red Hat Linux 6.0 (Linux 2.2.5)	GNU C++ (g++) distributed with Red Hat distribution. The command: `g++ -v` returns: `gcc version egcs-2.91.66 19990314/Linux (ecgs-1.1.2 release)`
PC with Windows NT, SGI NT, Windows 95/98/2000	Microsoft Visual C++ 6.0.

Disk Space Requirements

On UNIX, you need approximately 95 MB for VR-Link files and 45 MB for all documentation files.

On PCs, you need 42 MB for VR-Link files and 45 MB for all documentation files.

Online Help Browser Requirements

The online help system for the FOM Mapper Builder is an HTML-based system. To view online help, you must have a web browser installed that supports Javascript and Java. Java and Javascript must be enabled.

Backward Compatibility

There is a high degree of backwards compatibility with VR-Link 3.5. For example, the talk and listen example programs from VR-Link 3.5 compile and link unmodified against VR-Link 3.6.

Network Compatibility

HLA only

VR-Link 3.6 is compliant with:

- RPR-FOM 0.5, 0.7, 0.8, and 1.0
- MÄK Real-time RTI 1.3.3 and 1.3.3-ngc
- DMSO RTI 1.3v4 or later, RTI NG v1, v2, v3.0, v3.2.

DIS only

This release continues to support DIS 2.0.4, IEEE 1278.1, 2.1.4, and IEEE 1278.1a, and can therefore interoperate with DIS applications of any of these versions.

RPR FOM Versions Supported

VR-Link 3.6 has built-in support for versions 0.5, 0.7, 0.8, and 1.0 of the RPR FOM. It also supports VR-Link's ability to support alternative FOMs through the FOM Mapper. By default, VR-Link 3.6 uses RPR FOM 1.0.

If you want to use a version other than 1.0, pass the version number (0.5, 0.7, or 0.8) to the *DtRprFomMapper* constructor and pass the resulting object to the *DtExerciseConn* constructor. Also, make sure you are using a federation execution name that corresponds to the right FED file. For example:

```
DtExerciseConn conn("VR-Link08", "MyApp", new DtRprFomMapper(0.8));
```

VR-Link examples like *f18* and *hlaNetdump* have a command line option, `-f`, that you can use to choose an RPR FOM version by passing either `vlRPR05`, `vlRPR07`, or `vlRPR08` as the argument to `-f`.

RTI Support

The DMSO RTI NG-compatible version of VR-Link 3.6 comes with a version of the MÄK Real-time RTI 1.3.3 that is link-compatible with DMSO RTI NG (makRti1.3.3-ngc). VR-Link 3.6-ngc has been tested with DMSO RTI NG v1, v2, v3.0, and v3.2.

The non-NG-compatible version of VR-Link 3.6 comes with a version of the MÄK Real-time RTI 1.3.3 that is link-compatible with DMSO RTI 1.3 v4 or later (makRti1.3.3). This release has been tested with those DMSO RTI 1.3 releases.

VR-Link 3.6 may work with older versions of the MÄK Real-time RTI, but we recommend using it with release 1.3.3.

When using the MÄK Real-time RTI, remember to make sure that VR-Link can find your FED file, and optional *rid.mtl,* file by either putting them in the directory from which you are running, or by setting the environment variable RTI_CONFIG to the directory that contains them.

Library Names

VR-Link 3.6-ngc expects library names that match those of DMSO RTI NG v3.2. If you run VR-Link 3.6-ngc with DMSO RTI NG v3.2, both the UNIX and PC versions work without any modification. If you want to run VR-Link 3.6-ngc with DMSO RTI NG v1 or v2, you must make the following changes:

- For UNIX, you must use the libraries *libRTI-NGv2.so* or *libRTI-NGv1.so*, which are supplied by MÄK in the *./lib* directory. These libraries correspond to DMSO RTI NG v1 and v2. To use VR-Link 3.6-ngc with RTI NG v1 or v2, copy the appropriate library to the DMSO RTI NG *lib* directory and rename it *libRTI-NG.so*. (You might want to rename the v3.x *libRTI-NG.so* library first.)
- For PCs, rename the libraries that come with RTI NG v1 or v2 to the names used by the RTI NG v3.2 libraries (*libRTI-NG.lib* and *libfedtime.lib*).

New Features and Changes

This section describes the following features and changes to VR-Link:

- FOM Mapper Builder utility
- New clock mechanism
- Ability to automatically respond to HLA synchronization points
- Ability to delay discovery of reflected objects
- Changes to Makefiles
- Miscellaneous changes.

FOM Mapper Builder

The FOM Mapper Builder is a graphical user interface-based tool that can help you take advantage of the FOM agility features of VR-Link. VR-Link, and the MÄK products built against it (such as MÄK Stealth and MÄK VR-Forces), have built-in support for the HLA RPR FOM. They can also work with other FOMs if you create a FOM mapper that maps VR-Link's internal object model to the alternate FOM. The FOM Mapper Builder does much of the work of creating a FOM mapper.

Complete documentation for using the FOM Mapper is included as the last section of these Release Notes.

RTI Support for the FOM Mapper Builder

The FOM Mapper Builder requires an RTI to resolve FOM names. The *bin* directory contains the MÄK Real-time RTI libraries for this purpose. You need a MÄK Real-time RTI license and the FLEXLm license server in order to run the FOM Mapper Builder using the MÄK Real-time RTI. A full installation of the MÄK Real-time RTI is included on the distribution CD for VR-Link. Refer to the RTI documentation for information about getting a license.

If you want to use a DMSO RTI, remove the MÄK libraries from the *bin* directory and configure your system for a DMSO RTI.

The DMSO RTI 1.3*vx* returns attribute names in all lowercase while the MÄK Real-time RTI and DMSO RTI-NG preserves the case of attribute names. The result is that a FOM Mapper project generated with the DMSO RTI 1.3 cannot be correctly opened when using the MÄK Real-time RTI or DMSO RTI NG and vice versa.

New Clock Mechanism

In VR-Link 3.6, we have implemented a new more object-oriented interface for dealing with time. While the old, global functions (such as DtTimeInit(), DtSetSimTime(), DtGetSimTime(), and DtAbsRealTime()) still exist for backward compatibility, we recommend that users switch to the new method. However, it is important not to mix and match. If you switch to the new object-based method, make sure you replace all calls to the old global clock functions.

There is now a class called *DtClock*, which implements a VR-Link simulation clock. It is defined in *vlTime.h*. *DtClock* has member functions like setSimTime(), getTime-Time(), and absRealTime() that supersede the old global functions. An init function similar to DtTimeInit() is no longer necessary because initialization is performed by the class's constructor. Typically, users do not have to explicitly create an instance of *DtClock*, because one is created for you by the *DtExerciseConn* constructor. A pointer to *DtExerciseConn*'s clock is available through the new clock() member function.

As an example, here's how the clock object would be used in a standard simulation loop:

```
DtTime dt = 0.05;
DtTime simTime = 0;
DtClock* clock = exConn.clock();
while (simTime <= 10.0)
{
   // Tell VR-Link the current value of simulation time
   clock->setSimTime(simTime);

   // Wait till real time equals simulation time of next step
   DtSleep(simTime - clock->elapsedRealTime());
}
```

DtObjectPublishers and *DtReflectedObjects* will use their *DtExerciseConn*'s clock when performing dead-reckoning, thresholding, and so on.

In rare cases where you are creating an instance of an entity or aggregate state repository yourself outside of the context of a publisher or reflected object, you will need to tell the repository what clock to use if you want dead-reckoning to occur properly. This is done using the repository's setClock() function.

Ability to Automatically Respond to HLA Synchronization Points

VR-Link applications will now, by default, automatically respond to announced synchronization points by immediately indicating that the point has been achieved. This means that VR-Link-based applications will not hold up a federation execution when the application developer has not done anything special to respond to synchronization points.

To turn off this behavior, (if you want to respond to synchronization points yourself), call the following:

```
exConn.setReplyingToSynchronizationPoints(DtFalse);
```

Delayed Discovery of Reflected Objects

You can delay discovery of reflected objects until some user-defined condition is met. VR-Link will not add an object to a Reflected Object List or inform the application about it until the specified predicate evaluates to true.

While this functionality can be used in DIS as a means of filtering, it is especially useful in HLA applications where an object is discovered before attribute information arrives. For example, you can now indicate that VR-Link should not add the object to the reflected object list until values for certain important attributes (for example, entity type) arrive. This relieves applications from continuously checking whether various attribute values are valid yet.

In order to specify the discovery condition for a Reflected Object List, you must write a predicate function (one that returns a boolean value) and then register it with the list, using setDiscoveryCondition(). For example, if you want a Reflected Entity List to wait until the entity type is something other than the initial value of (0,0,0,0,0,0,0), you could write a function that looked like the following:

```
DtBoolean criteria(DtReflectedObject* obj, void* usr)
{
   DtReflectedEntity* ent = (DtReflectedEntity*) obj;
   if (ent->esr()->entityType() == DtEntityType(0,0,0,0,0,0,0))
   {
      return DtFalse;
   }
   return DtTrue;
}
```

Then tell the ReflectedEntityList about it like so:

```
DtReflectedEntityList rel(&exConn);
rel.setDiscoveryCondition(criteria, NULL);
```

The case where we want to wait for the entity type happens to be very common, so this behavior is already available in *DtReflectedEntityList,* but it is off by default. To indicate that you would like use it, when you create your list, call:

```
rel.discoverOnlyWhenEntityTypeKnown(DtTrue);
```

Calling it again with DtFalse turns this behavior off again.

Changes to Makefiles

The structure of our example Makefiles is different from VR-Link 3.5 and previous releases. If you want to use our Makefiles as-is, you need GNU make, rather than the native make that comes with some flavors of UNIX. You can download GNU make from http://www.gnu.org. In conjunction with this change, we have moved a lot of common Makefile variables and targets into various .mk files that are included by the individual example Makefiles. These .mk files are in *./mkinc*. Our Makefiles now contain "if" statements, so that you can build differently for different platforms with the same set of Makefiles.

From a usage point of view, little has changed. You still build HLA executables by invoking make (gmake actually) with no arguments, or with an argument of VERS=RPR. You build DIS executables using gmake VERS=5. The change only reflects the fact that we at MÄK wanted to take advantage of certain gmake features in our example Makefiles. You can continue to use native make to build your own VR-Link-based applications.

Miscellaneous Changes

This section describes a variety of changes and improvements to the VR-Link toolkit.

- DtObjectIdHashList::add now has an optional **oldData** argument, so that you can find out if you are replacing a previous entry associated with a particular object ID. It also has a new empty() function for removing all elements from the list.
- *DtIntrusiveList* now has removeAndDelete(), and removeAllAndDelete() member functions, which delete the specified data in addition to removing the pointer from the list.
- value() methods have been added to *DtStringHashKey*, *DtIntHashKey*, *DtSystem-HashKey*, and *DtObjectIdHashKey*.
- You can now look up a *DtHlaObject* in the *DtHlaObjectMgr* by object name.
- *DtTcpSocket* no longer drops packets when they can not be sent immediately, and now correctly pieces together packets that do not arrive in a single call to read. Also a DtBecomeBlocking() was added.
- The symbols DtGenericStateRepository, DtNotifyLevel, DtNewViewControlPdu, DtDumpFile, and DtTopoGeocTransCoord are now exported in Windows DLLs.
- *DtHlaObjectWithStateRep* now has stateRepForReflecting() and asSeenByRemote() member functions which provide access to the state repositories used for reflecting, and for keeping track of how remote simulations view locally simulated objects respectively. If there is more than one state repository for reflecting (for example, you have multiple RELs in your application), the first one is returned.
- Callback table for DIS PDUs now allows callbacks on PDU kind 255
- Accessors for have been added to *DtExerciseConn* for fedHandle(), fedName(), and execName().
- Netdump has a `-A` option to better support multiple ethernet cards.
- UNIX-style (no Ctrl-M's) FED files on Windows, or vice-versa, no longer cause problems for VR-Link.
- hlaNetdump, HLA version of f18, and the *myFomMap.dll* example now link against VR-Link DLLs, not statics.
- Warning messages only get printed once when you repeatedly try to send an attribute that has no associated encoder function.
- On Windows, the unnecessary *vrlFedTime* library is no longer distributed. Link with the RTI's *fedtime* or *libfedtime* library instead.

Documentation Updates

The URL listed on page 2-7, step 3 of the license key procedure, is incorrect. The correct URL is www.mak.com/licenses.htm.

Bug Fixes

VR-Link 3.6 fixes the following bugs:

- All PDU fields and attributes are now correctly decoded by *DtReflectedEmitterSystem*.
- *LgrRecFilePdu*'s and *LgrPlayFilePdu*'s filename() member functions work correctly now.
- Decoding of lists of HLA object names works correctly now. Previous, the decoded list would contain an extra empty name at the end.
- Aggregates and Emitter Systems were not using real time to do heartbeats. They are now.
- The DIS version if *DtEmitterSystemPublisher* no longer sends too many PDUs.
- Adding attributes to previously attributeless RPR FOM leaf classes used to cause crashes in VR-Link.
- *DtDump* functions (and by extension, functions like PDU and StateRepository printData() functions), now work correctly in Windows applications.
- GridDataRecType1 is now send properly in HLA
- The SGI version would occasionally crash with DMSO RTI NG, because getValuePointer() does not necessarily return a pointer to an eight-byte boundary. This has now been fixed.
- RadioTransmitter's CryptographicMode decoder works correctly now.
- *DtAggregateStateRepository* now allows marking sizes of up to 31 bytes for both HLA and DIS.
- With life-form entities, the bit shift values for setting concealed and camouflage appearance attributes were setting the wrong bits according to DIS standards.
- The DT_DLL_FOMMAP definition was removed from *MachineTypes.h*, so that users can build FOM mapper DLL'S properly without editing *MachineTypes.h*.
- hlaNetdump is no longer a CPU hog.

Known Problems

- In a FOM Mapper Builder project with saved mappings, if you unmap a class, and then choose Undo All Changes to Class Since Last Save, the mapping is restored, but the text labels that identify the class that a class is mapped to, are not displayed. If you try to unmap an attribute while attribute mapping is not active (the background of the attributes row is gray), the FOM Mapper Builder crashes.
- On some systems, when you activate the on-line help for the FOM Mapper Builder, if you already have a web browser open, the online help does not open up. If this occurs, close your open browser, then activate online help again. It should open up the default browser and display the help file.

Using the FOM Mapper Builder

This chapter explains how to start the FOM Mapper Builder and use it.

11.1. Introduction

The FOM Mapper Builder is a tool that can help you take advantage of the FOM agility features of MÄK VR-Link. VR-Link, and the MÄK products built against it (such as MÄK Stealth and MÄK VR-Forces), have built-in support for the HLA RPR FOM. In addition they can work with other FOMs using FOM mapping and, if necessary, by extending the VR-Link FOM-independent API. FOM agility and FOM mapping are discussed in detail in Chapter 6, "The HLA-Specific Interface" and Chapter 7, "FOM Agility Examples" of *MÄK VR-Link Developer's Guide*. You need to understand the concepts in these chapters to use the FOM Mapper Builder.

In a nutshell, the FOM Mapper Builder supports the creation of a FOM Mapper shared library (or DLL). The FOM Mapper library is implemented using C++ code that consists of a single FOM Mapper class and a set of two classes, an encoder and a decoder, associated with each FOM class mapping. Through these C++ classes, the FOM Mapper performs two primary operations:

- The FOM Mapper class registers the class mappings that associate the VR-Link internal object model (IOM) components with the FOM object and interaction classes. For example it registers associations between FOM object classes and VR-Link publishers so that the publishers know which FOM class to choose to represent a locally simulated object. It also registers associations between VR-Link reflected object lists and FOM object classes so that discovered objects are managed by the correct list.
- The encoder and decoder classes perform the encoding and decoding of data exchanged between the VR-Link API and the RTI (via RTI attribute and parameter value handle pair sets). Given an attribute representation of a vehicle type in the FOM (maybe one based on a flat enumeration, for example, GroundVehilceM1 and AircraftF16) a decoder must be able to interpret the FOM representation and convert it to a form that is usable by a VR-Link state repository to set its entity type (that is, a structure of hierarchically defined enumerations).

The FOM Mapper Builder allows you to do drag-and-drop mapping of the VR-Link IOM components to the components of the FOM. The drag-and-drop operations result in the automatic generation of the C++ code that implements the FOM Mapper class and the associated encoder and decoder classes. The resultant source and header files form the basis for generating a FOM Mapper library. Although you must do additional coding to complete the FOM Mapper, the FOM Mapper Builder does much of the required mechanical coding, saving you time and helping you to avoid coding errors.

In practice, the amount of code that must be modified depends on the similarity of the target FOM to the RPR FOM. The code generated for the FOM Mapper class is almost complete; excluded is the implementation of chooser functions. Chooser functions are needed for object class mappings where a VR-Link object publisher is associated with multiple FOM object classes. The code generated for encoders and decoders provides partially implemented stubs where only the data conversion must be completed. In some simple cases, where the target FOM data types closely match the RPR FOM (that is, types expected by VR-Link), the stubs require no modification at all. The FOM Mapper Builder includes placeholder IOM components (for example, MyExtendRepository and MyExtendInteraction) that support mappings for extensions to the FOM-independent API. However, the placeholder mappings require additional manual coding to replace the placeholder names with the actual extension names.

Figure 11-1 illustrates the FOM Mapper Builder window.

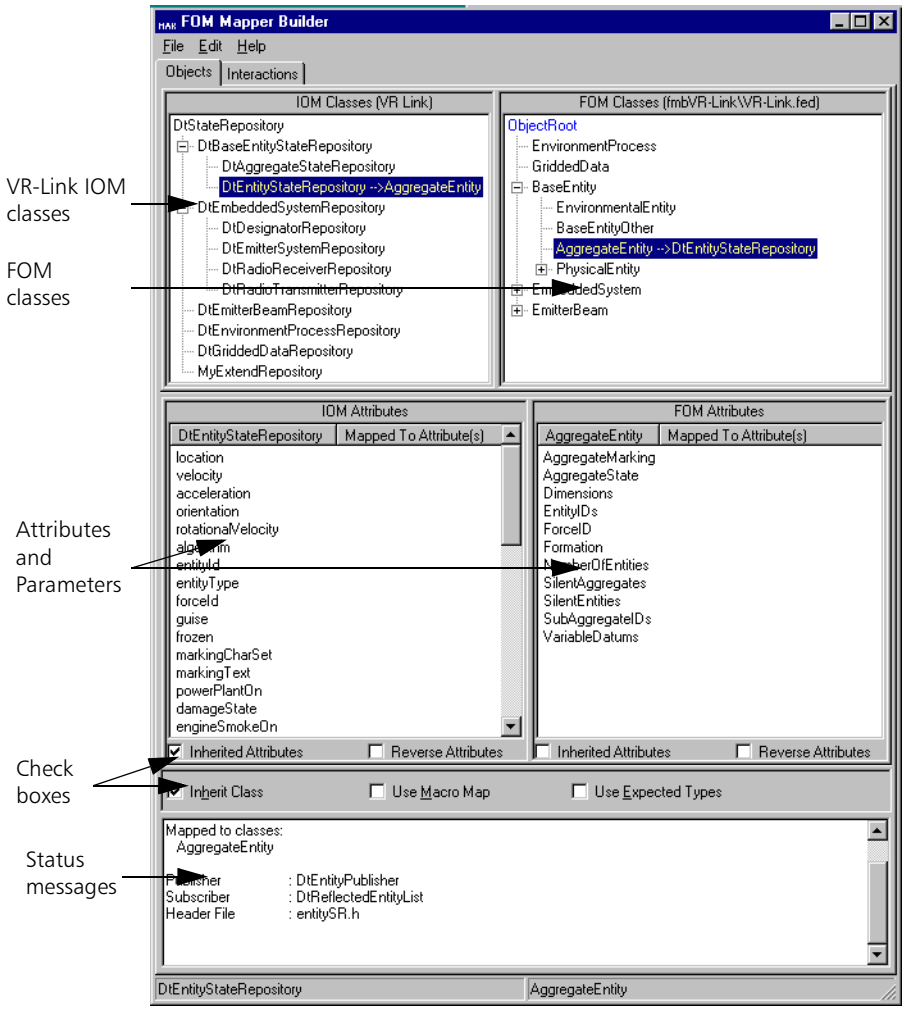

Figure 11-1. FOM Mapper Builder window

The VR-Link IOM classes and the FOM classes are separated into distinct object and interaction classes. The VR-Link interface also separates the mapping of these class types. One tab presents the object classes and another tab presents the interaction classes. Mapping between object and interaction classes is not allowed. The behavior of the object class and interaction class mapping is identical.

This chapter uses the terms “class” or “object” when referring to actions performed on object or interaction classes. The term “attribute” refers to both object attributes and interaction parameters, except where noted.

11.2. Installing the FOM Mapper Builder

The FOM Mapper Builder does not require a special installation process. When you install VR-Link, the FOM Mapper Builder executable is installed in the *./fmb* directory. Table 11-1 describes the subdirectories of the *./fmb* directory.

Table 11-1: FOM Mapper Builder directory structure

Directory	Description
./fmb/bin	FOM Mapper Builder executable.
./fmb/data	Parent directory for project directories.
./fmb/doc	FOM Mapper Builder documentation.

11.3. Starting the FOM Mapper Builder

You can start the FOM Mapper Builder from the command line, with a script or batch file, or, in MS-Windows, with the usual array of options.

To start the FOM Mapper Builder from the command line:

1. If you are using the FOM Mapper Builder with the DMSO RTI, start the rtiexec.

   If you are using the FOM Mapper Builder with the MÄK RTI, be sure the license manager is running.

2. Execute the following command:

   `fmb.exe [-f project_path]`

   where the optional parameter `-f project_path` specifies an existing FOM Mapper project directory. The path can be relative or fully qualified.

   The FOM Mapper Builder window opens (Figure 11-1).

If you start the FOM Mapper Builder without specifying a project, you can start a new project or open an existing project from the FOM Mapper Builder window. If you specify a project, the FOM Mapper Builder looks for a FED file in the project directory. If it finds more than one FED file, it asks you which one you want to use.

Other ways to start the FOM Mapper Builder on a PC include:

- On the Start menu, choose Programs → MÄK Technologies → FOM Mapper Builder.
- Create a shortcut to *./fmb/bin/fmb.exe* on your desktop
- Double-click *fmb.exe* in the Windows Explorer.

11.3.1. The FOM Mapper Builder Window

The FOM Mapper Builder window has two tabs, one for object classes, and one for interaction classes. Each tab has six panes arranged in four rows. The panes in the top row display the VR-Link API internal object model (IOM) classes in the left pane and the FOM classes in the right pane. The classes are displayed using a tree structure. Each tree displays the root component and the first level of classes that can be mapped. You use the top panes to map IOM classes to FOM classes.

The windows in the second row display the attributes of the classes selected in the top row. The attributes are displayed as lists. Check boxes for each list control the display of attributes.

The check boxes in the third row control how mappings are generated.

The fourth row displays status messages for mappings.

FOM Mapper Builder Window Status Reporting

The window has the following status features:

- Mapped classes and attributes are displayed in red
- When the background of the attributes row is gray, you cannot map attributes. When attribute mapping is enabled, the background changes to white.
- The name of the selected class component is displayed in the status bar at the bottom of the window frame.
- The left side of the status bar indicates the IOM component that has been selected. The right side indicates the FOM component that has been selected.
- By default, the IOM attribute list displays inherited attributes and the FOM attributes list does not. The Inherited Attributes check boxes below each attribute list control whether to display the inherited attributes or not. Attributes are listed alphabetically by class. When an attribute is selected, its name appears in the status bar next to the base class in which it is declared.

Changing the Layout of the FOM Mapper Builder Window

You can manipulate the window as follows:

- To expand or collapse the object trees, click the expand/collapse icon to the left of the component or double-click the component name.
- You can resize the height and width of the class and attribute panes and the height of the text window by dragging the border of the window pane.
- To display properties of a class, right-click the class name.
- The Reverse Attributes check boxes reverse the order in which attributes are listed. The default is base class attributes (if inherited attributes are displayed) followed by derived class attributes in alphabetical order by class. When reversed, the attributes are displayed with the derived class's attributes first and in reverse alphabetical order.

The root classes have no attributes that can be mapped; therefore, you cannot select them.

11.4. The FOM Mapper Project and Project Files

The FOM Mapper Builder stores the files it creates in a project directory. Project directories are named by prepending "fmb" to the name of the FED file that you select when you start a new project. The default location for project directories is *./fmb/data/fomMappers*, but you can save them wherever you want. All mapping files are generated in this directory. Initially, a newly created project contains only the target FED file, a source and header file containing a class definition that supports the FOM Mapper class (*classList.h* and *classList.cxx*), a template MS Visual C++ project (*fmbFomMapper.dsw* and *fmbFomMaper.dsp*), and a template Unix Makefile.

The first time you map a class, the FOM Mapper Builder creates both the files for the FOM Mapper class and the encoders and decoders for the class being mapped. Subsequent class mapping operations modify the FOM Mapper class files and create or delete the encoder and decoder class files. Initially, the encoder and decoder files contain just the basic constructor and destructor methods. Attribute mapping operations modify the encoder and decoder files to add or remove encoding and decoding methods.

The new files on disk are empty until you save them. If you exit without saving, the files are deleted.

When you open a project, the FOM Mapper Builder reads the mappings from the files in the project directory. Additional mapping operations performed during a session are added to the existing files, or new ones are created as necessary.

11.4.1. Creating A New FOM Mapper Project

To create a new FOM Mapper project:

1. Choose File → New. The Choose a FED file window opens. This is a standard open file window for your windowing system.
2. Select the FED file for the FOM you want to map to VR-Link.
3. Click Open. The Browse for Folder dialog box opens. Choose the directory in which you want to save the project. The default directory is the *./data/fomMappers* directory.
4. Click OK. The FOM Mapper Builder creates a project directory for the FOM Mapper and copies the FED file to it.

11.4.2. Opening A FOM Mapper Project

To open a FOM mapping project:

1. Choose File → Open. The Browse for Folder window opens. By default, it shows the *./data/fomMappers* directory and its project directories. If necessary, browse the directory tree to find your project.
2. Select the directory for the project you want to open.
3. Click OK.

11.5. Options for Mapping Classes

The FOM Mapper Builder has three options that you enable and disable with the following checkboxes:

- Inherit Classes
- Use Macro Map
- Use Expected Types.

These features offer trade-offs of convenience and safety. The default behavior is to inherit classes and to not use macros and expected types. With these settings, the FOM Mapper Builder creates fully expanded code. You will have to replace the type placeholders (`$ExpectedNetType$`) manually. These files will not compile unless you edit them. However, you are assured that you will address each encoding and decoding function, because the compiler will give you an error message if you miss editing a type.

If you use expected types, the FOM Mapper Builder makes a best guess at the type to include in the code. The resulting files will compile, but they may not work correctly if the FOM Mapper Builder guessed wrong. You will need to locate the incorrect types and correct them.

Use Macro Map provides the least access to the FOM Mapper Builder code.

You can select and clear the check boxes on a per-mapping basis, so that you can take advantage of the FOM Mapper Builder's coding abilities for classes that you know use expected types and reserve the questionable mappings for manual editing.

For more information see, "Mapping with Inheritance", on page 19, "Creating Mappings Using Expected Types", on page 20, and "Creating Mappings Using Macros", on page 22.

11.6. Mapping Object and Interaction Classes

The procedure for mapping object classes and interaction classes is identical. The only difference is the tab that you select before mapping the classes.

To map an IOM class to a FOM class:

1. Select the tab for the type of class that you want to map (objects or interactions.)
2. Select or clear the check boxes for class inheritance, use of macros, and use of expected net types.

Drag the IOM class onto the FOM class that you want to map it to. A message is displayed in the text window indicating which classes have been mapped. The IOM and FOM class labels are appended with each other's class name and the labels change to red type.

You can map only one IOM class to a FOM class. If you try to map more than one IOM class to a FOM class, the mapping is rejected and a rejection message is displayed in the text window.

Mapping with Inheritance

Both the FOM and the VR-Link IOM are defined as class hierarchies. In both class hierarchies, derived classes have access to inherited attributes. One important aspect of mapping attributes within the FOM class hierarchy is providing access to the inherited attributes; both from the IOM classes and from the base classes of the FOM class. For the IOM class hierarchy, it is only important that the IOM class being mapped to the FOM class contain all of the necessary IOM attributes (either inherited or not) needed to map to the FOM attributes. The easiest procedure is to select the IOM class at the deepest class level so that all of the attributes are available for mapping.

The interface with the RTI requires that the class mappings defined for derived FOM classes must have access to the mappings defined for the attributes of the FOM base classes. Access to mappings of these inherited attributes in a FOM class can be provided in one of two ways:

- The FOM inherited attributes can be associated with the derived FOM class that is mapped. When the FOM inherited attributes are displayed in the interface, the attribute mapping can be performed this way.
- The encoder and decoder C++ classes can derive from each other following the FOM class hierarchy. In this way, the encoder and decoder C++ classes inherit and have access to the attribute mapping methods of their base C++ classes.

Associating inherited attributes with the derived FOM class is appropriate if there is no requirement to subscribe to the FOM's base classes. However, if it is necessary to subscribe to the FOM's base classes, then these classes must be mapped. In this case, the FOM's inherited attributes should be mapped with the class that defines them and the encoder and decoder classes should use C++ class inheritance. Otherwise, the mapping of derived attributes will duplicate the attribute mapping code defined for the base classes.

11.7. Mapping Attributes and Parameters

Before you can map attributes and parameters, you must map an IOM class to the FOM class that has the attributes.

To map attributes and parameters:

1. Select the FOM class whose attributes you want to map. The attributes get displayed in the middle row. The background of the attribute row changes to white. (If the background is not white, the class has not been mapped to an IOM class.)
2. Drag an IOM attribute onto the FOM attribute you want to map it to. A message is displayed in the text window indicating which attributes have been mapped.

Notes:

- Mapping inherited attributes within a derived class does not affect the mapping status of that attribute in any other class.
- You can map an IOM attribute to more than one FOM attribute and you can map more than one IOM attribute to a FOM attribute. This many-to-many mapping relationship allows you to compose and decompose data between the IOM and FOM representations.

! Do not use macro mapping when mapping multiple IOM attributes to a single FOM attribute.

11.7.1. Creating Mappings Using Expected Types

The FOM Mapper Builder maps down to the attribute level. However, it does not have information about the attribute data types. For attribute data types, the FOM Mapper Builder can use an optimistic approach when it generates the encoding and decoding functions. It assumes that the FOM attributes will be defined using a VR-Link network data type that can be converted into the target VR-Link attribute. For example, the VR-Link *DtBaseEntityStateRepository* has a location attribute that uses the DtConstVector. This internal representation is represented on the network as a *DtNet64Vector*. The FOM Mapper Builder uses these "expected net types" to define the network values coming from the RTI.

If you want the FOM Mapper Builder to create files using expected types, check the Use Expected Types check box. Choosing this option means that you will have to do less coding, however if the expected type is not the correct type, your FOM mapper will not work.

You can select or clear the check box for each mapping, so that you can enable this option when you know that the expected type is the correct type and can disable it for those mappings that you know you have to code manually.

The following code shows code created with Use Expected Types disabled:

```
{
   RTI::ULong length = 0;
// Replace $ExpectedNetType$ with the VR-Link type to use
   $ExpectedNetType$* netVal = ($ExpectedNetType$*)
      attrs.getValuePointer(pairSetIndex, length);

   if (netVal)
   {
      if (length < sizeof($ExpectedNetType$))
      {
         FmbBaseEntityDecoder_SIZE_WARNING("AccelerationVector",
   $ExpectedNetType$, length);
         DtWarn("Attribute was discarded!\n");
      }
      else
      {
         if (length > sizeof($ExpectedNetType$))
            FmbBaseEntityDecoder_SIZE_WARNING("AccelerationVector",
   $ExpectedNetType$, length);

         stateRep->setAcceleration(*netVal);
      }
   }
```

The following code shows Use Expected Types enabled:

```
{
// The FOM Mapper Builder has chosen DtNetU32 as the expected type.
   DtNetU32 netVal = rep.primaryWeaponState();

   attrs->add(attrHandle, (char*)&netVal, sizeof(DtNetU32));
}
```

11.7.2. Creating Mappings Using Macros

The FOM Mapper Builder can create code that uses macros, instead of fully expanded code.

Decoding Macros

The use of DEFINE_SIMPLE macros for decoding is sufficient if the VR-Link attribute is known to have a conversion from the FOM net type. The following code shows a macro with Use Expected Types disabled:

```
DEFINE_SIMPLE_FmbPhysicalEntityDecoder_ATTR_DECODER(AlternateEntityType,
   $ExpectedNetType$, setGuise)
DEFINE_SIMPLE_FmbPhysicalEntityDecoder_ATTR_DECODER(ForceIdentifier,
   $ExpectedNetType$, setForceId)
```

The following code shows a macro with Use Expected Types enabled:

```
DEFINE_SIMPLE_FmbPlatformDecoder_ATTR_DECODER(HatchState, DtNetU32,
   setHatchState)
DEFINE_SIMPLE_FmbPlatformDecoder_ATTR_DECODER(AfterburnerOn,
   DtNetBoolean, setAfterburnerOn)
```

Encoding Macros

The use of DEFINE_SIMPLE macros for encoding is sufficient if the VR-Link attribute is known to have a conversion to the FOM net type.

11.8. Removing Class and Attribute Mappings

You can remove specific class or attribute mappings at any time, regardless of whether or not they have been saved. You can also undo all changes you have made to a class since the last time you saved the project.

To remove a class mapping:

1. Select the FOM class whose mapping you want to remove.
2. Choose Edit → Unmap Class. The class name is removed from the label. The FOM class changes from red type to black type to indicate that it is no longer mapped. The IOM class changes to black if no other FOM class is mapped to it.

To remove an attribute mapping:

1. Select the FOM class whose attribute mapping you want to remove.
2. In the FOM Attribute list, select the attribute you want to unmap.
3. Choose Edit → Unmap Attribute.

- If the FOM attribute is mapped to a single IOM attribute, the attribute name changes from red type to black type to indicate that it is no longer mapped. Unmapping the attribute is finished.
- If the FOM attribute is mapped to multiple IOM attributes, the Unmap from *attribute_name* dialog box (Figure 11-2) is displayed.

  Select the IOM attribute to unmap or <All> to unmap all of the IOM attributes. The selected attribute is removed from the Mapped To list. If you select <All>, then all of the attributes are removed from the Mapped To list and the attribute changes to black type.

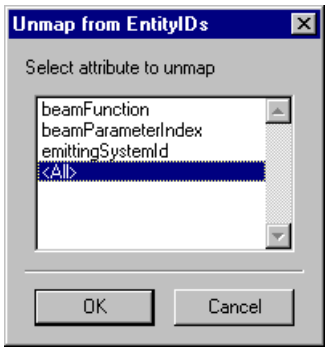

Figure 11-2. Unmap from *attribute_name* dialog box

To undo all changes made to a FOM class since the last time you saved the project (or first opened for new, unsaved projects):

1. Select the FOM class whose changes you want to undo.
2. Choose Edit → Undo All Changes to Class Since Last Save.

11.9. FOM Mapper Builder Files

The FOM Mapper Builder generates C++ source code that defines a FOM Mapper library for VR-Link. A header file declaration and a source code definition are generated for each C++ class. A single FOM Mapper class is created that serves as a central repository for FOM mapping information. For each mapping between IOM and FOM classes, an encoder and decoder class is generated. The attribute mappings are inserted as methods into the encoder and decoder classes.

The FOM Mapper files are created in a directory matching the federation name given at startup. The project directory is created beneath the directory in which the FOM Mapper Builder is located. If a windows shortcut link is used, the files are placed off of the working directory specified in the shortcut. FOM Mapper files are prefixed with "Fmb".

The FOM Mapper Builder looks in the project directory at startup for FOM Mapper files from a previous mapping session. These files are read into memory and the mapped classes are displayed in red type. Additional mappings performed during a session are added to the existing files. The files contain special marker tokens with associated data. The marker tokens are designated with the post fixed text "_M_". These markers and any other text on the same line are used to maintain the state of the mapper files. In addition, some of the marker tokens serve to delimitate the beginning and ending of code text associated with a class or attribute mapping.

Do not delete or change the markers, any text on a marker line, or the relative position of the marker lines within the file. Deleting or changing the markers within a file may corrupt their use for managing the state of future FOM Mapper Builder sessions.

Table 11-2: Marker tokens

Token	Description
FMB_ATE_M_	Mapping table entry; found in header files
FMB_ADEC_M_	Insertion point for method declaration(s)
FMB_AREG_M_	Insertion point for method registration(s)
FMB_ADFN_M_	Insertion point for method definition(s)
FMB_ADFNB_M_	Begin FOM attribute's method definition(s)
FMB_ADFNE_M_	End FOM attribute's method definition(s)
FMB_ASPEC_M_	Insertion point for IOM attribute specific code within a FOM attribute method
FMB_ASPECB_M_	Begin specific code
FMB_ASPECE_M_	End specific code
FMB_ANEED_M_	Insertion point for IOM attribute specific code within a FOM attribute "need test" method
FMB_ANEEDB_M_	Begin specific need code
FMB_ANEEDE_M_	End specific need code

11.9.1. The FOM Mapper Class Files

The FOM Mapper Builder maintains all changes in memory until the end of a mapping session or when you save the project, at which point, it saves the project in FOM Mapper class files. The file names match the C++ classes that they define with the appropriate ".h" or ".cxx" extensions. The FOM Mapper class name is the FED file name prepended with "Fmb" and post fixed with "Mapper" (for example, *FmbRPRMapper*). Certain characters are not allowed in C++ symbol names ("`~!@#$%^&*()-=+{[}]\\|;:'\",<.>/?"); these characters are replaced by underscores.

11.9.2. Class Mapper Encoder and Decoder Files

The class mapper files are created in memory after a mapping action. Four files are created for each class mapping: source and header files for both the encoder and decoder mapper classes. The encoder and decoder class names are the FOM class name prepended with "Fmb" and post fixed with a differentiation between object versus interaction and encoder versus decoder (for example, *FmbBaseEntityObjDecoder* or *FmbWeaponFireIntEncoder*).

11.9.3. Code Components

This section describes various components of the code generated by the FOM Mapper Builder.

Decode Attribute Size Checking

The decoders contain code to check the size of attributes coming from the RTI. An attribute that is smaller than the expected size is almost certainly corrupted and the processing of the attribute is discontinued. An attribute that is larger than the expected size might be valid, so only a warning is issued.

For most cases the expected size matches the size of the expected net type. However, this may not always be the case. For example, the expected type of a string might be "char*"; the size of the data will probably be variable (maybe with some upper limit). The size checking code should be examined when the size of incoming data might not match the expected net type.

Incomplete Translations

In some cases, a default translation of the VR-Link attributes cannot be automatically generated. Theses cases generate encoding and decoding functions that will not compile. A short comment delimited by "$" designates code segments that require modification.

One example is the *DtEntityStateRepository* articulated parts attribute. This attribute is too complex for any simple encoding and decoding method. Modification of the code is necessary to complete the translation.

Another case where the default encoding and decoding functions are not complete is when additional information is needed to perform the translation. An example is the *DtEntityStateRepository* light state attribute for specific light types. This attribute uses a simple Boolean along with a light type discriminator. The default mapping is to assume a boolean network data type, but the discriminator is needed to complete the translation. The discriminator may be implied by the FOM attribute or it may be part of the FOM attribute. In either case, completion of the translation requires modification of the code to provide the light state discriminator.

Class Mapping Registration

The FOM Mapper generated by the FOM Mapper Builder uses macros and a helper class to register the information about class mappings. The register macros create instances of the encoder and decoder classes and register them with the encoder and decoder factories. It uses an instance of *DtFmbClassList* (named classList) to record the class name mappings. The register macro for interactions also registers the interaction creator function with the interaction factory. After all of the class mappings have been registered using the register macros, the classList object has a record of each VR-Link object and interaction class that has been mapped and which FOM classes have been mapped to them. The FOM Mapper then invokes methods on the classList object to register the associations between VR-Link publishers and subscribers and the FOM object and interaction classes.

The use of the register macros and the classList helper is sufficient for registering all subscriber associations and for publisher associations where the VR-Link object or interaction class has only been mapped to a single FOM class. However, cases in which a VR-Link class has been mapped to multiple FOM classes require the creation of a class chooser function. For example, consider the case where *DtEntityStateRepository* has been mapped to the FOM classes of Vehicle, Aircraft, and Tank. In this case, the *DtEntityPublisher* must be associated with the three FOM object classes in such a way that it can choose the appropriate class for any particular instance of the *DtEntityStateRepository*. The solution is to create a chooser function that makes the choice of which class to use based on user-supplied data. The chooser function can then be registered with the VR-Link publisher by supplying it as an argument to any one or all of the register macros associated with the class mappings. If this chooser function is not supplied, then the VR-Link publisher will be associated with the last registered FOM class.

Specific VR-Link Attribute Encoding/Decoding

An encoder (or decoder) method is created when a FOM attribute is mapped with its first VR-Link attribute. The declaration of the method is added to the encoder header file and the registration and definition are added to the encoder source file. The declaration and registration are complete after the first mapping. This first time the definition code is generated in two steps. The first step creates the encoder definition with most of the basic code, but it does not contain the specific code to access (or mutate) the VR-Link attribute. The first step does, however, use the VR-Link attribute expected net type if that option is selected. The second step inserts the VR-Link attribute specific code using marker comments located within the encoder definition. Subsequent mappings of VR-Link attributes to that same FOM attribute result in only this second step being performed.

Removing a VR-Link attribute mapping only removes the VR-Link attribute's specific code. If removing a VR-Link mapping results in the FOM attribute having no mapping then the encoder method definition, declaration, and registration code is also removed.

11.10. Compiling A FOM Mapper Builder Project

This section explains how to compile a FOM Mapper Builder project for different supported platforms.

11.10.1. Compiling for A PC

You can use a MS Visual C++ template project file to compile the generated FOM Mapper Builder files into a DLL for use with Microsoft Windows versions of MÄK products. The project file uses some environment variables that need to be defined (Table 11-3).

Table 11-3: MSVC Project Environment Variables

Name	Description
MAK_VRLDIR	Path name of the MAK VR-Link installation
MAK_RTIDIR	Path name of the MAK RTI directory

In order to use the project template, you must add the generated FOM Mapper Builder files to it.

11.10.2. Compiling for Unix

You can use the Makefile to compile the generated FOM Mapper Builder files into a shared library for use with the UNIX versions of MÄK products. The Makefile defines two variables at the top of the file that need to be defined (Table 11-4). To use the Makefile, you must replace the files in the OBJ and SRC lists with the generated FOM Mapper Builder files. The Makefile uses the VR Link make includes.

Table 11-4: Unix Make Variables

Name	Description
MY_HOME	Path name of the FOM Mapper source directory
VL_HOME	Path name of the MAK VR-Link installation

The initial make command should be `make install`. The install target generates the make dependencies and begins the build of the FOM Mapper library. Subsequent make invocations need only use `make` with no target to build the FOM Mapper library.

Some platforms do not support the FOM Mapper dependencies for file names longer than 13 characters (less the extension). Invoking `make` with no target will always result in a recompilation of every FOM Mapper file.

11.11. Sample Code

This section contains two files example files generated by the FOM Mapper Builder. The first shows an example of a class mapping. The second shows a decoder class.

Class Mapping

```
/*********************************************************************
** VR-Link Fom Mapper
*********************************************************************/

#if DtHLA

#include "exerciseConn.h"
#include "fom.h"
#include "classList.h"
#include "FmbVR_LinkMapper.h"
#include "encFactory.h"
#include "decFactory.h"
#include "intDecFact.h"
#include "intEncFact.h"
#include "hInterFactory.h"
#include "intClassDesc.h"

#include "FmbBaseEntityObjDecoder.h"
#include "FmbBaseEntityObjEncoder.h"

// FMB_INC_M_

// Function to create an instance of the FOM Mapper
extern "C"
{

FMB_DLL_FOMMAP DtFomMapper* DtCreateFomMapper(void* usr)
{
   return new FmbVR_LinkMapper();
}

FMB_DLL_FOMMAP void DtDeleteFomMapper(DtFomMapper* mapper)
{
   delete mapper;
}

}


// Macro defintion for object registration
#define REGISTER_OBJECT_MAPPER(classNameStr, encoder, decoder,    \
                               publisher, subscriber, chooser)    \
   objClass = exConn->fom()->objClassByName(classNameStr);        \
   if (objClass)                                                  \
   {                                                              \
      stateEncoderFactory()->addEncoder(                          \
         objClass->handle(),                                      \
         new encoder(exConn, objClass));                          \
      stateDecoderFactory()->addDecoder(                          \
         objClass->handle(),                                      \
         new decoder(exConn, objClass));                          \
```

```
   }                                                                    \
   classList.addObjClassName(publisher, subscriber, classNameStr,
   chooser);


// Macro defintion for interaction registration
#define REGISTER_INTERACTION_MAPPER(classNameStr, encoder, decoder,     \
                                    publisher, subscriber,              \
                                    createFunc, chooser)                \
   interClass = exConn->fom()->interClassByName(classNameStr);          \
   if (interClass)                                                      \
   {                                                                    \
      interactionEncoderFactory()->addEncoder(                          \
         interClass->handle(),                                          \
         new encoder(exConn, interClass));                              \
      interactionDecoderFactory()->addDecoder(                          \
         interClass->handle(),                                          \
         new decoder(exConn, interClass));                              \
   }                                                                    \
   classList.addInterClassName(publisher, subscriber, classNameStr,
   chooser); \
   interactionFactory()->addCreator(classNameStr, createFunc);


FmbVR_LinkMapper::FmbVR_LinkMapper() :
   DtEmptyFomMapper()
{
    // Real initialization takes place in virtual init() function,
    // when a DtExerciseConn is available.
}

void FmbVR_LinkMapper::init(DtExerciseConn* exConn)
{
   DtFmbClassList classList;
   DtObjClassDesc* objClass = NULL;
   DtInterClassDesc* interClass = NULL;

    // Call base class init.  This sets myExConn, and initializes all
    // factories to default or empty factories (no encoders or decoders
    // registered, no mappings between FOM interaction classes and VR-Link
    //interaction classes.
   DtEmptyFomMapper::init(exConn);

    // Register mapper for object and interaction classes.
    // Macro performs the following steps:
    //
    // 1. Add encoders and decoders for the object and interaction
    //classes. When DtEntityPublisher, DtReflectedEntity, and VR Link
    // interactions ask the FOM mapper for encoders and decoders to use,
    // the FOM mapper will return clones of the encoders and decoders
    // registered here.
    // 2. Set up mappings for publishing.  Indicate the FOM class to use
    // for kind of DtObjectPublisher, and each kind of DtInteraction.  A
    // class chooser function must be supplied if multiple FOM classes
    // are associated with a single VR Link publisher.
    // 3. Set up mappings for subscribing.  Indicate which FOM class each
    //    kind of DtReflectedObjectList should subscribe to, and which FOM
    //    class each kind of DtInteraction's addCallback function should
    // subscribe to.
    // 4. Set up mappings for creating DtInteraction instances.  Indicate
```

```
    // what kind of DtInteraction to create to represent each FOM
    // interaction class.

    // FMB_ROBJB_M_ DtEntityStateRepository ObjectRoot.BaseEntity
   REGISTER_OBJECT_MAPPER("ObjectRoot.BaseEntity",
                          FmbBaseEntityObjEncoder,
                          FmbBaseEntityObjDecoder,
                          "DtEntityPublisher",
                          "DtReflectedEntityList", NULL)
    // FMB_ROBJE_M_

   // FMB_ROBJ_M_

   // FMB_RINT_M_

   // Set object and interaction classes that were previously registered.
   classList.setObjectClassDeclMgmt(this);
   classList.setInteractionClassDeclMgmt(this);

}

FmbVR_LinkMapper::~FmbVR_LinkMapper()
{
}

#endif
```

Decoder

```
/*********************************************************************
** VR_Link BaseEntity Decoder
*********************************************************************/

#if DtHLA

#include "entitySR.h"
#include "FmbBaseEntityObjDecoder.h"
#include "NetStructs.h"
#include "hExerciseConn.h"
#include "fom.h"

#define FmbBaseEntityObjDecoder_SIZE_WARNING(attributeName, type, length
   ) \
DtWarn("\nSize of value decoded from RTI update for attribute %s (%d)\n" \
    " is %s than size of %s (%d)\n", \
     attributeName, length, \
    (length < sizeof(type) ? "smaller" : "larger"), #type, sizeof(type));


// Default constructor
FmbBaseEntityObjDecoder::FmbBaseEntityObjDecoder(
   DtExerciseConn* exConn,
   DtObjClassDesc* classDesc) :
   DtHlaStateDecoder(exConn, classDesc)
{
    // Register the individual decoding functions for BaseEntity
    // attributes with the object.  The decoding functions have names like
```

```
    // decodePosition. The macro expands to look like this:
    // addDecoder("Position", (DtAttributeDecoder) decodePosition);

   DtADD_ATTR_DECODER(Orientation);
   DtADD_ATTR_DECODER(AccelerationVector);

   // FMB_AREG_M_
}

FmbBaseEntityObjDecoder::~FmbBaseEntityObjDecoder()
{
}

// Decoding functions

// FMB_ADFNB_M_ acceleration AccelerationVector
void FmbBaseEntityObjDecoder::decodeAccelerationVector(
   DtEntityStateRepository* stateRep,
   const RTI::AttributeHandleValuePairSet& attrs,
   int pairSetIndex)
{
   RTI::ULong length = 0;
   $ExpectedNetType$* netVal = ($ExpectedNetType$*) decodingBuffer();

   attrs.getValue(pairSetIndex, (char*)netVal, length);

   if (length < sizeof($ExpectedNetType$))
   {
      FmbBaseEntityObjDecoder_SIZE_WARNING("AccelerationVector",
   $ExpectedNetType$, length);
      DtWarn("Attribute was discarded!\n");
   }
   else
   {
      if (length > sizeof($ExpectedNetType$))
         FmbBaseEntityObjDecoder_SIZE_WARNING("AccelerationVector",
   $ExpectedNetType$, length);

      stateRep->setAcceleration(*netVal);
   }
}
// FMB_ADFNE_M_


// FMB_ADFN_M_

#endif
```