VR-Forces

Migration Guide

VT MAK
A company of VT Systems

VR-Forces

Migration Guide

VT MAK
A company of VT Systems

VT MAK
150 Cambridge Park Drive, 3rd Floor
Cambridge, MA 02140 USA

Voice: 617-876-8085
Fax: 617-876-9208

info@mak.com

www.mak.com

Revision VRF-4.5-9-170221

Contents

Preface

This manual is for developers who must migrate applications from previous versions of VR-Forces to the current version.

VR-Forces Documentation

VR-Forces documentation is provided as manuals in PDF format, online help, and HTML class documentation. The PDF files are in the *./doc* directory. The VR-Forces documentation set is as follows:

- *VR-Forces Users Guide* contains all documentation for running VR-Forces, creating and running scenarios, and configuring VR-Forces.
- *VR-Forces Migration Guide* collates API migration information for recent releases.
- Online help. The VR-Forces front-end, the OPD Editor, and the Simulation Object Editor have online help accessible from the Help menu.
- *VR-Forces Developers Guide* and API documentation. Class documentation and developers guides in linked HTML pages.
- *VR-Forces Release Notes*.
- VR-Forces *First Experience Guide*. A brief introduction to the most basic features of VR-Forces.
- *VR-Forces Entity Model Catalog*. A catalog of all of the simulation objects and tactical graphics configured in the Simulation Object Editor, with basic parameter details and a screen capture of the 3D model or icon.

MAK Products

The VR-Forces is a member of the VT MAK line of software products designed to streamline the process of developing and using networked simulated environments. The VT MAK product line includes the following:

- VR-Link® Network Toolkit. VR-Link is an object-oriented library of C++ functions and definitions that implement the High Level Architecture (HLA) and the Distributed Interactive Simulation (DIS) protocol. VR-Link has built-in support for the RPR FOM and allows you to map to other FOMs. This library minimizes the time and effort required to build and maintain new HLA or DIS-compliant applications, and to integrate such compliance into existing applications.

  VR-Link includes a set of sample debugging applications and their source code. The source code serves as an example of how to use the VR-Link Toolkit to write applications. The executables provide valuable debugging services such as generating a predictable stream of HLA or DIS messages, and displaying the contents of messages transmitted on the network.

- MAK RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MAK RTI is optimized for high performance. It has an API, RTIspy®, that allows you to extend the RTI using plug-in modules. It also has a graphical user interface (the RTI Assistant) that helps users with configuration tasks and managing federates and federations.
- VR-Forces®. VR-Forces is a computer generated forces application and toolkit. It provides an application with a GUI, that gives you a 2D and 3D views of a simulated environment.

  You can create and view entities, aggregate them into hierarchical units, assign tasks, set state parameters, and create plans that have tasks, set statements, and conditional statements. You can simulate using entity-level modeling, which focuses on the actions of individual people and vehicles, and aggregate-level modeling, which focuses on the interaction of large hierarchical units.

  VR-Forces also functions as a plan view display for viewing remote simulation objects taking part in an exercise. Using the toolkit, you can extend the VR-Forces application or create your own application for use with another user interface.

- VR-Vantage™. VR-Vantage is a line of products designed to meet your simulation visualization needs. It includes three end-user applications (VR-Vantage Stealth, VR-Vantage PVD, and VR-Vantage IG) and the VR-Vantage Toolkit.
  - VR-Vantage Stealth displays a realistic, 3D view of your virtual world, a 2D plan view, and an exaggerated reality (XR) view. Together these views provide both situational awareness and the big picture of the simulated world. You can move your viewpoint to any location in the 3D world and can attach it to simulation objects so that it moves as they do.
  - VR-Vantage IG is a configurable desktop image generator (IG) for out the window (OTW) scenes and remote camera views. It has most of the features of the Stealth, but is optimized for its IG function.
  - VR-Vantage PVD provides a 2D plan view display. It gives you the big picture of the simulated world.
  - SensorFX. SensorFX is an enhanced version of VR-Vantage that uses physics based sensors to view terrain and simulation object models that have been materially classified. It is built in partnership with JRM Technologies.
  - The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MAK's VR-Vantage applications, or to integrate VR-Vantage capabilities into your custom applications. VR-Vantage is built on top of OpenSceneGraph (OSG). The toolkit includes the OSG version used to build VR-Vantage.
- MAK Data Logger. The Data Logger, also called the Logger, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal and can quickly jump to areas of interest. The Logger has a GUI and a text interface. The Logger API allows you to extend the Logger using plug-in modules or embed the Logger into your own application. The Logger editing features let you merge, trim, and offset Logger recordings.
- VR-Exchange™. VR-Exchange allows simulations that use incompatible communications protocols to interoperate. For example, within the HLA world, using VR-Exchange, federations using the HLA RPR FOM 1.0 can interoperate with simulations using RPR FOM 2.0, or federations using different RTIs can interoperate. VR-Exchange supports HLA, TENA, and DIS translation.
- VR-TheWorld™ Server. VR-TheWorld Server is a simple, yet powerful, web-based streaming terrain server, developed in conjunction with Pelican Mapping. Delivered with a global base map, you can also easily populate it with your own custom source data through a web-based interface. The server can be deployed on private, classified networks to provide streaming terrain data to a variety of simulation and visualization applications behind your firewall.

- DI-Guy™. The DI-Guy product line is a set of software tools for real-time human visualization, simulation, and artificial intelligence. Every DI-Guy software offering comes with thousands of ready-to-use characters, appearances, and motions. DI-Guy enables the easy creation of crowds and individuals who are terrain aware, autonomous, and react intelligently to ongoing events. Save time, money and create outstanding simulations with DI-Guy. The DI-Guy product line includes the following products:
  - The DI-Guy SDK. Embed the DI-Guy library in your real-time application and populate your world with lifelike human characters.
  - DI-Guy Scenario™. Author and visualize human performances in a rich, user-friendly graphical environment. Use DI-Guy Scenario as an end visualization application or save scenarios and load them into your DI-Guy SDK enabled application.
  - ECOSim. Enhanced Company Operations Simulation (ECOSim) is a company-level training simulation that teaches leaders how best to deploy troops, UAVs, convoys, and other assets. ECOSim focuses on ease-of-use, rapid scenario generation, runtime operator control, and realistic and reactive human simulation.
  - DI-Guy AI. Generate crowds of autonomous characters to quickly populate your worlds with hundreds and thousands of terrain-aware, collision avoiding DI-Guys. Used as a module on top of DI-Guy Scenario and DI-Guy SDK.
  - Expressive Faces Module. Enable DI-Guy characters to have faces that display emotion, eyes that look in directions and blink, and lips that sync to sound files.
  - DI-Guy Motion Editor. Create or customize motions to your particular needs in an easy-to-use graphical application.
- RadarFX. RadarFX is a client-server application that can simulates synthetic-aperture radar (SAR). The server application, which is based on VR-Vantage and SensorFX, loads a terrain database and, optionally, connects to simulations. A client application requests SAR images from the server. VR-Forces includes a sample client application.

How to Contact Us

For VR-Forces technical support, information about upgrades, and information about other MAK products, you can contact us in the following ways:

Telephone

Call or fax us at:	Voice:	617-876-8085 (extension 3 for support)
	Fax:	617-876-9208

E-mail

Sales and upgrade information:	info@mak.com
Technical support:	support@mak.com
VR-Vantage support:	

Internet

MAK web site home page:	www.mak.com
License key requests:	www.mak.com/support/licenses/ get-licenses
Product version and platform information:	www.mak.com/support/product-versions
For the free, unlicensed MAK RTI:	www.mak.com/resources/bonus-material
MAK Community Forum:	www.mak.com/connect/forum

Post

Send postal correspondence to:	VT MAK 150 Cambridge Park Drive, 3rd Floor Cambridge, MA, USA 02140

When requesting support, please tell us the product you are using, the version, and the platform on which you are running.

Document Conventions

This manual uses the following typographic conventions:

`Monospaced`	Indicates commands or values you enter.
`Monospaced Bold`	Indicates a key on the keyboard.
`Monospaced Italic`	Indicates command variables that you replace with appropriate values.
Blue text	A hypertext link to another location in this manual or another manual in the documentation set.
`{ }`	Indicates required arguments.
`[ ]`	Indicates optional arguments.
|	Separates options in a command where only one option may be chosen at a time.
(|)	In command syntax, indicates equivalent alternatives for a command-line option, for example, `(-h | --help)`.
/	Indicates a directory. Since MAK products run on both Linux and Windows PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Italic	Indicates a file name, pathname, or a class name.
sans Serif	Indicates a parameter or argument.
➤	Indicates a one-step procedure.
Menu → Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose **File → Save**.
🎥	Click the icon to run a tutorial video in the default browser.
ℹ	Indicates supplemental or clarifying information.
❗	Indicates additional information that you must observe to ensure the success of a procedure or other task.
▣	Indicates that a section is valid only for entity-level scenarios.
▭	Indicates that a section is valid only for aggregate-level scenarios.

Directory names are preceded with dot and slash characters that show their position with respect to the VR-Forces home directory. For example, the directory *vrforces4.5/doc* appears in the text as *./doc*.

DI-Guy Conventions

Table 1-1 lists the conventions used for entity coordinates in DI-Guy documentation.

Table 1-1: Coordinate system conventions

Convention	Indicates

XYZ	X = forward,
	Y = to the left
	Z = up

Yaw, Roll, Pitch	Orientation is applied in the order YRP. Yaw = rz – orientation about the vertical axis Roll = rx – orientation about the fore-aft axis Pitch = ry – orientation nose down, nose up

Mouse Button Naming Conventions

An instruction to click the mouse button, refers to clicking the primary mouse button, usually the left button for right-handed mice and the right button for left-handed mice. The context-sensitive menu, also called a popup menu or right-click menu, refers to the menu displayed when you click the secondary mouse button, usually the right button on right-handed mice and the left button on left-handed mice.

Third Party Licenses

MAK software products may use code from third parties. This section contains the license documentation required by these third parties.

Boost License

VR-Link, and all MAK software that uses VR-Link uses some code that is distributed under the Boost License. All header files that contain Boost code are properly attributed. The Boost web site is: www.boost.org.

libXML and libICONV

VR-Link and all MAK software that uses VR-Link, links in libXML and libICONV. On some platforms the compiled libraries and header files are distributed with MAK Products. MAK has made no modifications to these libraries. For more information about these libraries please see the following web sites:

- The LGPL license is available at: http://www.gnu.org/licenses/lgpl.html.
- Information about IconV is at: http://www.gnu.org/software/libiconv/.
- Information about LibXML is at: http://xmlsoft.org/.

Lua

Some MAK products use the Lua programming language (www.lua.org). Its license is as follows:

LizardTech

Freefont OpenType Font Set

VR-Vantage applications and VR-Forces use the Freefont OpenType font set from the Free Software Foundation. It is covered by the General Public License (GPL). For details, please see: http://www.gnu.org/licenses/gpl.html

Autodesk Gameware Navigation

osgoculusviewer Library

Third-Party Licenses for VR-Vantage Applications

VR-Vantage applications use a variety of third-party libraries. Developers who want to use these libraries may be required to purchase developer's licenses. Please see section 1.2 in *VR-Forces Front-End Developers Guide*.

1. Migrating from VR-Forces 3.12 to 4.0.3

The VR-Forces 4.x GUI is built with the VR-Vantage Toolkit, which is significantly different from the VR-Forces 3.12 GUI API. This migration guide describes the major differences between VR-Forces 3.12 and VR-Forces 4.0 through 4.0.3.

1.1. Getting Information About Objects

In VR-Forces 3.12 *DtModelKey* was used to find object information in various dictionaries, most notable the *DtModelDataDictionary*. In VR-Forces 4.x this has been replaced with *DtElementId*. The *DtElementId* is the main key to use to look up data in the system. For details, please see "Storing Data," on page 1-2.

The API extends the element ID with the scene object ID. The scene object ID (*DtUniqueId*) represents an individual screen representation of an object, given a particular model set. For example, the 2D representation of an object has a different scene object ID than the 3D representation. If you have a scene object ID, you can use the *DtElementData* object to retrieve the element ID based on scene object ID. If you need a scene object ID from an element ID, the *DtElementData* can also do that mapping.

Selection sets (discussed in "Managing Object Selection," on page 1-3) contain the *DtElementId* of the objects selected. You can use these IDs to retrieve information about selected objects.

1.2. Storing Data

In VR-Forces 3.12, *DtModelData* was used as the foundation for data storage, symbol creation, and symbol updating. The *DtModelData* class does not exist in VR-Forces 4.x. It has been replaced with the *DtStateView*. A *DtStateView* is a template class that takes a *DtSimState* object as a template. This object is implemented as a subclass, and those subclasses contains the data that represents the information about the object. In VR-Forces 4.x, the main *DtSimState* objects are:

- *DtVrlinkSimulatedBaseState*. Information about any VR-Link-based object, such as marking text, type, force
- *DtVrlinkSimulatedEntityState*. Information about VR-Link entities.
- *DtVrlinkSimulatedAggregateState*. Information about VR-Link aggregates.
- *DtVrlinkSimulatedEnvironmentProcess*. Information about VR-Link environmentals.
- *DtVrfObjectDataState*. Information about an object that is VR-Forces-specific

When an object is discovered on the network, information about that object from its VR-Link reflected object is placed into the appropriate object state. This state is added to a queue for updating.

In VR-Forces 3.12, as soon as an object update was received over the network, it was immediately updated. In VR-Forces 4.x this is not the case. Each object is updated at a certain frame rate. This allows for better performance and control over when object information is updated. For example, if an object is selected, its object update rate is increased, assuming the user might want to know more information about this object. This means all other objects continue to update at a default rate, or not at all. This does not effect the position, speed, or orientation of the object, simply the additional data that is transmitted to the front-end for that object.

This information is stored in *DtStateViewCollection* classes. Each *DtStateViewCollection* keeps a list of all particular objects of a state view type and can be used to retrieve information about that object type. So, for example, there are collections of entities, aggregates, and environmental processes.

In VR-Forces the *DtVrfStateViewCollectionManager* manages all these collections. It is your way of retrieving information about a particular object and its *DtStateView* information. The *DtVrfStateViewCollectionManager* is analogous to the VR-Forces 3.12 *DtModelDataDictionary* class.

In the *DtVrfStateViewCollectionManager*, all objects that are in the system are stored in the mySimulatedBaseStateViewCollection member. This member holds all structures that are of a *DtVrlinkSimulatedBaseState* (which all state view objects subclass from). Using the findStateView() methods of the *DtVrfStateViewCollectionManager*, you can find the basic state view information about any object. This information can be retrieved by marking text name or by *DtElementId* of the object. If you know that an item is specifically an entity, aggregate, or environmental, you can use the state view collection that is appropriate to find the state data for that object.

1.3. Managing Object Selection

In VR-Forces 3.12 you would use the map area of the currently active window to get a list of objects that were currently selected. In VR-Forces 4.x you use the *DtSelection-Manager* class to get the list of object selected, as follows:

```
DtSelectionManager::IdSet currentSelections =
  DtSelectionManager::instance(myDe).currentSelection();
```

You can then iterate over the list to get information about these objects from the *DtVrf-StateViewCollectionManager*:

```
DtSelectionManager::IdSet::const_iterator iter =
  currentSelections.begin();

while (iter != currentSelections.end())
{
  DtStateView<DtVrlinkSimulatedEntityState>* stateView =
    const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
    DtVrfStateViewCollectionManager::instance(myDe).entityStateView()
    ->findStateView( *iter ) );
  ++iter;
}
```

VR-Forces 4.x also supplies a convenience class called the *DtVrfSelectionHandler*. This class has methods for retrieving information about current selections. For example, if you want to get data for all the currently selected objects, you can use the code in the previous example or you could do the following:

```
std::vector<makVrv::DtSimEntry*> entries =
  DtVrfSelectionHandler::instance(myDe).getSelectedEntries();
```

Remember that each *DtSimEntry* contains the list of all available data for that object. Most objects in VR-Forces have two *DtSimEntry* views of data: the VR-Link State Repository data (*DtVrlinkSimulatedEntityState*) and the VR-Forces-specific object data (*DtVrfObjectDataState*). You can iterate over the simStates() of the state entry, dynamically casting to the object in question that you want. Once the dynamic cast succeeds, you can use that data.

1.4. Creating Symbols

In VR-Forces 3.12 if you wanted to affect the way that symbols were created, you would override the *DtMtlSymbolMapper*, and install your own version.

In VR-Forces 4.x, symbols are created from *DtVisualDefinitionSets*. These visualizer sets are organized in *DtObjectDictionary* classes. Each class of object (entity, aggregate, environmental, interaction) has its own object dictionary. In order to affect the look of a symbol, you can modify the object dictionary, adding your own *DtVisualDefinition* or *DtVisualDefinitionSet*. The exampleObjectDictionary, exampleRangeLine and exampleVisualDefinitionSet examples show how this is done.

Mapping entity type enumerations to models is now done in the Entity Type Mappings dialog box rather than in symbol map files. Entity types are mapped to entity definitions that contain the correct visualizer information for that entity type.

In VR-Forces 4.x, the actual visual definitions are not created in the main GUI thread. A *DtElement* object (or a subclass) is created. It contains all the visualizers needed to visualize an object. *DtElement* objects are created in the network thread and are protected from the main thread. If you want to, in code, change how these items look, you will need to do that before creation by changing the visual definition set for that particular object. Individual visualizers, however, can be created to self configure given certain conditions. The point being stressed here is that there has been an attempt to move away from direct manipulation of the symbols and move towards a more object oriented approach for symbol manipulation.

If you need to get access to an individual symbol, you can do this through the *DtAgent-Manager*. Since each object is scene dependent, you will need to know the model set you are referencing to get the scene object ID of the object you care about:

```
DtElementData::SceneObjectIdList ids;

myDe.dataBank().elementData().getSceneObjectIds(elementId, ids,
  myDe.driverManager().inputDriver().
  currentChannel()->currentModelSet());
DtUniqueID idToUse = elementId;

if (ids.size())
{
  idToUse = ids[0];
}

DtAgentUpdateResolverInterface* updater =
  myDe.agentManager().findUpdater( idToUse );
```

```
if(updater)
{
  DtSceneObject* sceneObject =
    updater->castObjectTypeFromUpdater<DtSceneObject>( "DtSceneObject");

  if ( sceneObject )
  {
    sceneObject->getPosition(0, dbLocation);
  }
}
```

Please refer to the *DtSceneObject* class for the information you can retrieve.

The scene objects are screen representations only and do not contain simulation data.

1.5. Updating Symbols

In VR-Forces 3.12, in order to affect how a symbol was updated, you would have either subclassed and installed your own version of the *DtVrfGuiSymbolUpdater* class or added post update callbacks.

In VR-Forces 4.x there is no centralized symbol updater. Each visualizer is responsible for doing its own updating. When a visualizer is created, it is handed a subclass of a *DtStateListener*. This class contains all the simulation data necessary to drive the visuals of an object. Signals are defined in these state listener subclasses that visualizers can connect to in order to update their state.

If you want to change how an existing object is updating, you must subclass and install that object into the *DtVrlinkStateVisualizerFactory* if this object has a VR-Link representation, or the *DtStateVisualizerFactory* if it does not.

To add subclasses to the VR-Link state visualizer factory you must create driver accessories for the VR-Link protocol you want to change. (Please see the addAttr example for an example of how to create an accessory.) You can then reference the VR-Link state visualizer factory as in the slot_onSimCreated() method (in *DtAddAttrGuiAccessory.inl*):

```
DtVrlinkStateVisualizerFactory& visualizerStateFactory =
  DtVrlinkStateVisualizerFactory::instance(
  (DtVrlinkConnection*)connection );
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

The same methodology can be used for the state visualizer factory:

```
DtWriteSettingsLock<DtSharedStateVisualizerFactory>
  visualizerStateFactory(DtSharedSettingsManager::instance(myDe));
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

1.6. Sending Messages to the Back-end

In VR-Forces 3.12, to communicate with the back-end (sending sets, tasks or interface content messages), the remote controller was used: DtVrfGuiAppEventController::remoteController().

In VR-Forces 4.x there is no direct access to the remote controller from the GUI thread. Since the GUI is not network dependent, the *DtVrfSimMessager* class was created to interface the GUI to the back-end. This class should be used to send all messages. The interface content example shows how to send interface content messages using *DtVrfSimMessager*.

1.7. Adding Callbacks to Receive Messages

In VR-Forces 3.12 the *DtVrfMessageInterface* class was used to register callback messages for message types. In VR-Forces 4.x the *DtVrfSimMessageHandler* is used. The *DtVrfSimMessager* also interfaces with this class, and that can be used as well. The interface content example shows how to register for a callback message using *DtVrfSimMessageHandler*.

1.8. Keyboard and Mouse Handling

In VR-Forces 3.12 you would subclass and install an event handler to handle mouse events and keyboard events. You might also have subclassed and installed your own version of the *DtPvdMapArea* or connected to signals to handle events.

In VR-Forces 4.x you subclass and install a *DtEventProcessor*. The *DtEventProcessor* subclasses have methods for mouse and keyboard messages. If you want to handle a mouse or keyboard message, override the appropriate method (processKeyboardEvent() or processMouseEvent()). If you handle the message you can return true or false. You can also be notified of events that were handled by other event processors before your event processor got a chance, by overriding the above methods and handling the case of a MouseLost event.

Add your event processor subclass into the *DtInputDriver* class. The *DtInputDriver* object is referenced through the driver manager:

```
DtInputDriver& inputDriver = myDe.driverManager().inputDriver();
```

The event processor can be added to the front or the back of the list (addEventProcessorToFront() or addEventProcessorToBack()). You can also get the list of event processors and insert it directly (eventProcessorList()). The exampleEventProcessor example has an example of how to use an event processor.

1.9. Changes to Signal Usage

VR-Forces 3.12 mostly used Qt signals. VR-Forces 4.x mostly uses Boost signals. While some Qt signals are used in the Qt layer, they are usually translated back into Boost signals.

To use a Boost signal you must connect to a signal and give a method to call when the signal is invoked.

The following examples show how to connect to signals without parameters (preTick), and with parameters (postTick). The binding is done to an object class and an instance of that object.

```
myDe.signal_preTick.connect(boost::bind(&MyObject::method, this));
myDe.signal_postTick.connect(boost::bind(&MyObject::method, this, _1));
```

! It is very important to disconnect signals when they are no longer needed or when the object is freed. Failing to disconnect signals can lead to application instability, as, unlike Qt signals, when the object is destroyed the signal is not disconnected.

1.9.1. Application Tick Notification

In VR-Forces 3.12 there was no way to be notified when the application was about to tick or had completed a tick. In VR-Forces 4.x, you can get this information. The main class of a VR-Vantage application is *DtDe*. The *DtDe* class is typically passed to any object that might need to use it, and is available to any plug-in on startup.

To be notified when the application is about to tick, connect to the signal_preTick signal.

On a post tick, connect to the signal_postTick signal.

1.9.2. Selection Management

In VR-Forces 3.12, to find out when selections had changed, you would connect to the Qt signal sent by the map area on selection changed (selectionUpdated, selection-Changed, and so on). In VR-Forces 4.x, you use the Selection Manager and its signals to be notified when selections have changed. You can get a reference to the Selection Manager as follows:

```
DtSelectionManager& selManager = DtSelectionManager::instance(myDe);
```

The signal_currentSelectionChanged signal is sent any time the current selection is changed. It supplies the following parameters:

- The current selection.
- What was unselected.
- The new selection type.
- The old selection type.

The IDs supplied are element IDs, which can be used to retrieve data about the selected objects. "Managing Object Selection," on page 1-3 explains how to use this information.

The signal_vertexSelected and signal_vertexDeselected signals are sent when vertices on a line are selected or unselected

1.9.3. Tracking New Objects

In VR-Forces 3.12, to find out when an object was added, you would have used the modelDataAdded signal. In VR-Forces 4.x you use the *DtElementData* object. You can get to the *DtElementData* as follows:

```
DtElementData& elementData = myDe.dataBank().elementData();
```

The signal signal_elementsAdded is sent for any elements added during that tick.

You can then use the State View Collection Manager to find information about the elements added. Due to the nature of the multi-threading of the application, it is possible that the state view data does not yet exist for those elements. In that case, for the state view collection you want to retrieve information for, connect to the signal_stateViewAdded signal. For example:

```
DtVrfStateViewCollectionManager::instance(myDe).
  baseStateView()->signal_stateViewAdded
```

When the state view is added you can then get information about that object.

1.9.4. Tracking Object Updates

In VR-Forces 3.12, to be notified when an object was updated, you might have installed your own version of the symbol updater or connected to the model data dictionary's modelDataUpdated signal. In VR-Forces 4.x you use the state view itself to be notified when state view data is updated.

You first need to find the state view you are interested in:

```
DtStateView<DtVrlinkSimulatedEntityState>* stateView =
  const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
  selHandler.entityStateView()->findStateView( principalSimEntryId ) );
stateView->signal_stateViewUpdated.connect( boost::bind(
  &MyObject::slot_onPrincipalStateViewUpdated, this, _1));
```

You will now be notified when that object is updated. This is more efficient than the way it was done in VR-Forces 3.12, because now you can be notified when a very specific instance of the data you are interested in is updated.

1.9.5. Tracking Object Removals

In VR-Forces 3.12 you would have used the modelDataAboutToBeRemoved signal. In VR-Forces 4.x you use the element data's signal_elementsToBeRemoved signal. This provides a list of elements that are being removed in this tick. Please see "Tracking New Objects," on page 1-8 for how to retrieve a reference to the *DtElementData*.

2. Migrating from VR-Forces 3.12 to 4.0.4

The VR-Forces 4.x GUI is built with the VR-Vantage Toolkit, which is significantly different from the VR-Forces 3.12 GUI API. This migration guide describes the major differences between VR-Forces 3.12 and VR-Forces 4.0.4.

2.1. Getting Information About Objects

In VR-Forces 3.12 *DtModelKey* was used to find object information in various dictionaries, most notable the *DtModelDataDictionary*. In VR-Forces 4.x this has been replaced with *DtElementId*. The *DtElementId* is the main key to use to look up data in the system. For details, please see "Storing Data," on page 2-2.

The API extends the element ID with the scene object ID. The scene object ID (*DtUniqueId*) represents an individual screen representation of an object, given a particular model set. For example, the 2D representation of an object has a different scene object ID than the 3D representation. If you have a scene object ID, you can use the *DtElementData* object to retrieve the element ID based on scene object ID. If you need a scene object ID from an element ID, the *DtElementData* can also do that mapping.

Selection sets (discussed in "Managing Object Selection," on page 2-3) contain the *DtElementId* of the objects selected. You can use these IDs to retrieve information about selected objects.

2.2. Storing Data

In VR-Forces 3.12, *DtModelData* was used as the foundation for data storage, symbol creation, and symbol updating. The *DtModelData* class does not exist in VR-Forces 4.x. It has been replaced with *DtObjectDataInterface*. This class (and subclasses) let you access data for any simulated object in the system:

- *DtEntityDataInterface*
- *DtAggregateDataInterface*
- *DtTacticalGraphicDataInterface*
- *DtSpotReportDataInterface*.

To retrieve information about objects, you can either use the object's element ID or the object's name (marking text) from the *DtVrfStateViewCollectionManager*. The *DtVrfStateViewCollectionManager* class is analogous to the *DtModelDataDictionary* in VR-Forces 3.12. In the *DtVrfStateViewCollectionManager* class you will use the lookupAndCastObjectData() method to return a shared data pointer to the data you need. For example, if you want to find an entity data:

```
DtEntityDataInterfacePtr data =
  DtVrfStateViewCollectionManager::instance(myDe).lookupAndCastObjectDat
  a<DtEntityDataInterface>("M1A2 1");
```

Also, the IDs in the current selection set can be used to look up data for selected objects.

In VR-Forces 3.12, as soon as an object update was received over the network, it was immediately updated. In VR-Forces 4.x this is not the case. Each object is updated at a certain frame rate. This allows for better performance and control over when object information is updated. For example, if an object is selected, its object update rate is increased, assuming the user might want to know more information about this object. This means all other objects continue to update at a default rate, or not at all. This does not effect the position, speed, or orientation of the object, simply the additional data that is transmitted to the front-end for that object.

This information is stored in *DtStateViewCollection* classes. Each *DtStateViewCollection* keeps a list of all particular objects of a state view type and can be used to retrieve information about that object type. So, for example, there are collections of entities, aggregates, and environmental processes.

In VR-Forces the *DtVrfStateViewCollectionManager* manages all these collections. It is your way of retrieving information about a particular object and its *DtStateView* information. The *DtVrfStateViewCollectionManager* is analogous to the VR-Forces 3.12 *DtModelDataDictionary* class.

In the *DtVrfStateViewCollectionManager*, all objects that are in the system are stored in the mySimulatedBaseStateViewCollection member. This member holds all structures that are of a *DtVrlinkSimulatedBaseState* (which all state view objects subclass from). Using the findStateView() methods of the *DtVrfStateViewCollectionManager*, you can find the basic state view information about any object. This information can be retrieved by marking text name or by *DtElementId* of the object. If you know that an item is specifically an entity, aggregate, or environmental, you can use the state view collection that is appropriate to find the state data for that object.

The guiObjectDataInterface example details how to use these data interface classes.

2.3. Managing Object Selection

In VR-Forces 3.12 you would use the map area of the currently active window to get a list of objects that were currently selected. In VR-Forces 4.x you use the *DtSelection-Manager* class to get the list of object selected, as follows:

```
DtSelectionManager::IdSet currentSelections =
  DtSelectionManager::instance(myDe).currentSelection();
```

You can then iterate over the list to get information about these objects from the *DtVrf-StateViewCollectionManager*:

```
DtSelectionManager::IdSet::const_iterator iter =
  currentSelections.begin();

while (iter != currentSelections.end())
{
  DtStateView<DtVrlinkSimulatedEntityState>* stateView =
    const_cast<DtStateView<DtVrlinkSimulatedEntityState>*>(
    DtVrfStateViewCollectionManager::instance(myDe).entityStateView()
    ->findStateView( *iter ) );
  ++iter;
}
```

VR-Forces 4.x also supplies a convenience class called the *DtVrfSelectionHandler*. This class has methods for retrieving information about current selections. For example, if you want to get data for all the currently selected objects, you can use the code in the previous example or you could do the following:

```
std::vector<makVrv::DtSimEntry*> entries =
  DtVrfSelectionHandler::instance(myDe).getSelectedEntries();
```

Remember that each *DtSimEntry* contains the list of all available data for that object. Most objects in VR-Forces have two *DtSimEntry* views of data: the VR-Link State Repository data (*DtVrlinkSimulatedEntityState*) and the VR-Forces-specific object data (*DtVrfObjectDataState*). You can iterate over the simStates() of the state entry, dynamically casting to the object in question that you want. Once the dynamic cast succeeds, you can use that data.

2.4. Creating Symbols

In VR-Forces 3.12 if you wanted to affect the way that symbols were created, you would override the *DtMtlSymbolMapper*, and install your own version.

In VR-Forces 4.x, symbols are created from *DtVisualDefinitionSets*. These visualizer sets are organized in *DtObjectDictionary* classes. Each class of object (entity, aggregate, environmental, interaction) has its own object dictionary. In order to affect the look of a symbol, you can modify the object dictionary, adding your own *DtVisualDefinition* or *DtVisualDefinitionSet*. The exampleObjectDictionary, exampleRangeLine and exampleVisualDefinitionSet examples show how this is done.

Mapping entity type enumerations to models is now done in the Entity Type Mappings dialog box rather than in symbol map files. Entity types are mapped to entity definitions that contain the correct visualizer information for that entity type.

In VR-Forces 4.x, the actual visual definitions are not created in the main GUI thread. A *DtElement* object (or a subclass) is created. It contains all the visualizers needed to visualize an object. *DtElement* objects are created in the network thread and are protected from the main thread. If you want to, in code, change how these items look, you will need to do that before creation by changing the visual definition set for that particular object. Individual visualizers, however, can be created to self configure given certain conditions. The point being stressed here is that there has been an attempt to move away from direct manipulation of the symbols and move towards a more object oriented approach for symbol manipulation.

If you need to get access to an individual symbol, you can do this through the *DtAgentManager*. Since each object is scene dependent, you will need to know the model set you are referencing to get the scene object ID of the object you care about:

```
DtElementData::SceneObjectIdList ids;

myDe.dataBank().elementData().getSceneObjectIds(elementId, ids,
  myDe.driverManager().inputDriver().
  currentChannel()->currentModelSet());
```

```
DtUniqueID idToUse = elementId;

if (ids.size())
{
  idToUse = ids[0];
}

DtAgentUpdateResolverInterface* updater =
  myDe.agentManager().findUpdater( idToUse );

if(updater)
{
  DtSceneObject* sceneObject =
    updater->castObjectTypeFromUpdater<DtSceneObject>( "DtSceneObject");

  if ( sceneObject )
  {
    sceneObject->getPosition(0, dbLocation);
  }
}
```

Please refer to the *DtSceneObject* class for the information you can retrieve.

i The scene objects are screen representations only and do not contain simulation data.

2.4.1. Creating Local (Unpublished) Symbols

In VR-Forces 3.12 you could create unpublished symbols by creating an appropriate *DtModelData* subclass and adding that model data into the application's Model Data Dictionary. You would then update that model data and mark it for update in the application to change the symbol.

In VR-Forces 4.x, there is now a *DtLocalObjectManager* class that does this for you. The guiLocalCreateObject example shows how to use this new class.

2.5. Updating Symbols

In VR-Forces 3.12, in order to affect how a symbol was updated, you would have either subclassed and installed your own version of the *DtVrfGuiSymbolUpdater* class or added post update callbacks.

In VR-Forces 4.x there is no centralized symbol updater. Each visualizer is responsible for doing its own updating. When a visualizer is created, it is handed a subclass of a *DtStateListener*. This class contains all the simulation data necessary to drive the visuals of an object. Signals are defined in these state listener subclasses that visualizers can connect to in order to update their state.

If you want to change how an existing object is updating, you must subclass and install that object into the *DtVrlinkStateVisualizerFactory* if this object has a VR-Link representation, or the *DtStateVisualizerFactory* if it does not.

To add subclasses to the VR-Link state visualizer factory you must create driver accessories for the VR-Link protocol you want to change. (Please see the addAttr example for an example of how to create an accessory.) You can then reference the VR-Link state visualizer factory as in the slot_onSimCreated() method (in *DtAddAttrGuiAccessory.inl*):

```
DtVrlinkStateVisualizerFactory& visualizerStateFactory =
  DtVrlinkStateVisualizerFactory::instance(
  (DtVrlinkConnection*)connection );
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

The same methodology can be used for the state visualizer factory:

```
DtWriteSettingsLock<DtSharedStateVisualizerFactory>
  visualizerStateFactory(DtSharedSettingsManager::instance(myDe));
visualizerStateFactory.addStateVisualizerCreator(
  "DtEntityKinStateVisualizer", new MyEntityKinStateVisualizerCreator);
```

2.6. Sending Messages to the Back-end

In VR-Forces 3.12, to communicate with the back-end (sending sets, tasks or interface content messages), the remote controller was used: DtVrfGuiAppEventController::remoteController().

In VR-Forces 4.x there is no direct access to the remote controller from the GUI thread. Since the GUI is not network dependent, the *DtGuiThreadVrfRemoteController* class interfaces the GUI to the back-end. Use this class to send all messages. The interface content example and the guiThreadVrfRemoteController example show how to send interface content messages using *DtGuiThreadVrfRemoteController*.

2.7. Adding Callbacks to Receive Messages

In VR-Forces 3.12 the *DtVrfMessageInterface* class was used to register callback messages for message types. In VR-Forces 4.x the *DtNetworkMessageCallbackManager* class is used. The *DtGuiThreadVrfRemoteController* also interfaces with this class, and that can be used as well. The interface content example and the networkCallbackManager example show how to register for a callback message using *DtNetworkMessageCallbackManager*.

2.8. Keyboard and Mouse Handling

In VR-Forces 3.12 you would subclass and install an event handler to handle mouse events and keyboard events. You might also have subclassed and installed your own version of the *DtPvdMapArea* or connected to signals to handle events.

In VR-Forces 4.x you subclass and install a *DtEventProcessor*. The *DtEventProcessor* subclasses have methods for mouse and keyboard messages. If you want to handle a mouse or keyboard message, override the appropriate method (processKeyboardEvent() or processMouseEvent()). If you handle the message you can return true or false. You can also be notified of events that were handled by other event processors before your event processor got a chance, by overriding the above methods and handling the case of a MouseLost event.

Add your event processor subclass into the *DtInputDriver* class. The *DtInputDriver* object is referenced through the driver manager:

```
DtInputDriver& inputDriver = myDe.driverManager().inputDriver();
```

The event processor can be added to the front or the back of the list (addEventProcessorToFront() or addEventProcessorToBack()). You can also get the list of event processors and insert it directly (eventProcessorList()). The exampleEventProcessor example has an example of how to use an event processor.

2.9. Changes to Signal Usage

VR-Forces 3.12 mostly used Qt signals. VR-Forces 4.x mostly uses Boost signals. While some Qt signals are used in the Qt layer, they are usually translated back into Boost signals.

To use a Boost signal you must connect to a signal and give a method to call when the signal is invoked.

The following examples show how to connect to signals without parameters (preTick), and with parameters (postTick). The binding is done to an object class and an instance of that object.

```
myDe.signal_preTick.connect(boost::bind(&MyObject::method, this));
myDe.signal_postTick.connect(boost::bind(&MyObject::method, this, _1));
```

!

It is very important to disconnect signals when they are no longer needed or when the object is freed. Failing to disconnect signals can lead to application instability, as, unlike Qt signals, when the object is destroyed the signal is not disconnected.

2.9.1. Application Tick Notification

In VR-Forces 3.12 there was no way to be notified when the application was about to tick or had completed a tick. In VR-Forces 4.x, you can get this information. The main class of a VR-Vantage application is *DtDe*. The *DtDe* class is typically passed to any object that might need to use it, and is available to any plug-in on startup.

To be notified when the application is about to tick, connect to the signal_preTick signal.

On a post tick, connect to the signal_postTick signal.

2.9.2. Selection Management

In VR-Forces 3.12, to find out when selections had changed, you would connect to the Qt signal sent by the map area on selection changed (selectionUpdated, selection-Changed, and so on). In VR-Forces 4.x, you use the Selection Manager and its signals to be notified when selections have changed. You can get a reference to the Selection Manager as follows:

```
DtSelectionManager& selManager = DtSelectionManager::instance(myDe);
```

The signal_currentSelectionChanged signal is sent any time the current selection is changed. It supplies the following parameters:

- The current selection.
- What was unselected.
- The new selection type.
- The old selection type.

The IDs supplied are element IDs, which can be used to retrieve data about the selected objects. "Managing Object Selection," on page 2-3 explains how to use this information.

The signal_vertexSelected and signal_vertexDeselected signals are sent when vertices on a line are selected or unselected

2.9.3. Tracking New Objects

In VR-Forces 3.12, to find out when an object was added, you would have used the modelDataAdded signal. In VR-Forces 4.x you use the *DtElementData* object. You can get to the *DtElementData* as follows:

```
DtElementData& elementData = myDe.dataBank().elementData();
```

The signal signal_elementsAdded is sent for any elements added during that tick.

You can then use the State View Collection Manager to find information about the elements added. Due to the nature of the multi-threading of the application, it is possible that the state view data does not yet exist for those elements. In that case, for the state view collection you want to retrieve information for, connect to the signal_stateViewAdded signal. For example:

```
DtVrfStateViewCollectionManager::instance(myDe).
  baseStateView()->signal_stateViewAdded
```

When the state view is added you can then get information about that object.

2.9.4. Tracking Object Updates

The object data interface allows you to change how often you want to have data refreshed into the object and to be notified when that data is updated. The setUpdat-eRate() method lets you set the update frequency. The boost signal_dataUpdated signal is sent when data has been updated.

2.9.5. Tracking Resources

To track resources for entities, you can connect to the signal_resourcesChanged() method of the data interface class and then inspect the resources() method to see the resources. You can request new resources by calling the requestResources() method.

2.9.6. Tracking Object Removals

In VR-Forces 3.12 you would have used the modelDataAboutToBeRemoved signal. In VR-Forces 4.x, you use the element data's signal_elementsToBeRemoved signal. This provides a list of all the elements that are being removed in this tick. Please see "Tracking New Objects," on page 2-8 for how to retrieve a reference to the *DtElement-Data*.

You can also track the removal of specific objects. The boost signal_dataRemoved is sent from the data interface when a simulated object is removed.

3. Simulation Model Set Changes in VR-Forces 4.0.4

This chapter summarizes changes to system definitions and OPE files.

3.1. Simulation Model Set Changes

Changes to the SMS are organized by component or system type.

3.1.1. Flight Command Controllers

Air domain Movement Systems. The flight-command-controller was added to Fixed-wing and rotary-wing entities to handle new flight tasks. (For details, please see fixed-WingFlightCommandController.h and rotaryWingFlightCommandController.h.) If you have created your own air domain movement systems, you must add this controller to take advantage of these new tasks and behavior.

3.1.2. Weapon Interface

The target-priorities parameters have moved. They are no longer part of the target selection controller. They are now in the weapon controllers inside the weapon systems, as follows:

- **target-selection-controller:**
  - Target priorities have been removed from the target selection controller (moved into weapons).
  - The `target-select-controller:weapon-system` connections have been removed. The **target-selection-controller** to sensor connections remain.
  - The controller uses *DtComponentDescriptor* (component-descriptor) for the descriptor type).
- Weapon systems:
  - Target priorities have been added into the weapon controllers. The **target-selection-criteria** parameter has been renamed to **targeting-control**.
  - Target priority metadata has been removed.
  - For ballistic weapons, the **system:target-list** connection is now `<controllerName>:target-to-acquire`. Please see *125mm-gun.sysdef* for an example.
  - Missile Weapons. The `system:target-list` input connection has been removed.

Laser designators have the same changes as weapon systems. They also now have the parameter **integrated-with-launcher**, which determines how target input works (either from the launcher controller, if integrated, or the target selection controller directly if not).

Apache. The laser designator that went along with the laser-guided-hellfire-missile launcher has been moved into the launcher.

The laser-guided-hellfire-missile-launcher system has the following changes:

- Uses the new controller integrated-laser-guided-launcher-controller.
- The laser designator that used to be on the entity is now in this system.
- A new connection (connect missile-launcher:target-to-acquire laser-controller:target-list) was added to represent the missile launcher providing targets to the laser designator.

3.1.3. Radar Modes

Emitter systems now defines a radar-mode-list, each one of which defines a beam-list so these can be switched at runtime. For more information, please see Section 10.5, "Configuring Emitters", in *VR-Forces Configuration Guide*.

3.1.4. Weapon Enhancements

In weapon systems the munition-wrapper resource "ammo" has been removed. Individual munitions are now regular resources.

Ballistic guns have two changes:

- Burst fire. Some ballistic guns now fire in bursts. The rounds-per-minute and extra-time-between-bursts parameters were added to the ballistic gun controller.
- Magazine-based reloading. The rounds-per-magazine parameter was added for magazine-based reloading.

3.1.5. Range Rings

- Weapon systems. The range-name parameter was added to ballistic weapon actuators and missile launcher controllers. This is the display name for range items associated with this weapon displayed in the GUI.
- Aggregate OPE files. The range-name parameter was added. The combat-range-controller was added to define range rings that can represent disaggregation ranges or subordinate weapon systems.
- Munition OPE files. The range-name parameter was added to the missile/torpedo entity definition to display missile range.

3.1.6. Ammo Select Tables

For weapons that fire a burst, hit probability is for the whole burst.

New weapons have been added, so if you have made your own damage system, you may have to make a new damage table to add new mappings.

4. Migration to VR-Forces 4.1

This chapter lists changes to the APIs for VR-Forces 4.1. For occasional updates to migration information, please check http://www.mak.com/support/migration-support.html.

4.1. API Changes

VR-Forces has the following changes to its APIs:

- Tasks, Sets and Reports now use a string as a type identifier instead of an integer.
- Simulation engine:
  - The simulation engine now supports streaming and paging of feature data. The API has been updated to add this support. (For details, please see section 15.6 in VR-Forces Developers Guide.)
  - *DtSimComponent* now has a virtual preFirstTickInit() method, called before the first call to tick().
- GUI:
  - The *DtNetworkMessageCallbackManager* class was renamed to *DtGuIThreadNetworkCallbackManager*.
  - For attributes and capabilities settings, the *attributes.mtl* and *capabilities.mtl* files are no longer used. The values are read from the SISO *.xml* file. All items should now be SISO compliant.
  - *dis-entity-types.csv* is no longer used in the Entity Editor or //front-end for selecting entity types. The SISO *.xml* file is now used.
  - Many GUI elements are now available to create via the *DtVrfTaskSetVariableWidgetManager*. Any parameter that is added to a scripted tasked is created using this factory. The list of GUI elements that can be created are in *vrfScriptedTasks.h*.
  - Task, Set and Create menu configuration is now initially set from configurations found in the *DtVrfSimulationManager*. The menuManipulation example has been modified to show the new usage.

The VR-Vantage Control Toolkit has the following changes:

- View control messages are sent using *DtDataInteractions*. *DtDataInteractions* provide a receiverId field. The recieverId's siteId and applicationId are now used when processing view control messages. If these match the siteId and applicationId of the connection receiving the message, the message is processed. The message is also processed if siteId and applcationId are both -1's (wildcards). For backwards compatibility, siteid/applicationId of 0/0 is also processed.
- Some view control messages control observer-specific settings. These messages now let you specify the name of an observer mode to modify. If no observer mode is specified, then these commands affect all observer modes currently in use. If an observer mode is specified, then only that observer mode is affected.

4.2. Simulation Model Set Changes

Most entity OPE files now have a the new script-controller controller that enables the Lua scripting feature on the entity. See the M1A2_1_1_1_255_1_1_3_-1.ope file in .\data\simulationModelSets\default for an example.

The default-task-rules.tsk file, which controls task concurrency, has changed in format. If you created your own tasks and needed edited this file, you will need to update it.

If you modified any files in the gui folder of the simulation model set, these files have been altered a bit and will need to be updated.

If you created movement system definition (sysdef) files or modified the defaults, note the following changes:

- The path-movement controller has been removed from all systems and OPE files and should not be in any of the files.
- The convoy-task-controller has changed how it finds roads. See ground-disaggregated-movement.ssydef in \data\simulationModelSets\default\systems\movement for an example.
- Some movement sysdef files now have a compatibility-controller, This is because we have removed the plan-and-move-to tasks. Any scenarios that had plans with these tasks will still work if those entities have the compatibility-controller. See ground-wheels-off-road.sysdef in \data\simulationModelSets\default\systems\movement for an example.
- The rail-path-movement controller has been simplifed a bit and some parameters have changed. See ground-wheels-off-road.sysdef in \data\simulationModelSets\default\systems\movement for an example.
- The obstruction-sensor sensor has some different parameters for obstacle avoidance. See air-cushion-default.sysdef in \data\simulationModelSets\default\systems\movement for an example.

5. Migration to VR-Forces 4.2

This chapter lists changes to the APIs for VR-Forces 4.1. For occasional updates to migration information, please check http://www.mak.com/support/migration-support.html.

5.1. Changes to the Object Parameters Database

The object parameters database has been significantly changed. Going forward, these changes should make it much easier to migrate customized simulation model sets to new releases. It will also make it easier to add new entity simulation models.

The "higher level" organization of the OPD now consists of a set of "platforms" and entity files. The platforms are generic OPE files that represent the basic entity and object types, such as ground platforms, surface platforms, line objects, area objects, and so on. These files use the familiar MTL format. It is expected that platforms will rarely change. Any changes to a platform affect all entities of that platform type.

Individual entities no longer have OPE files. Entity specific parameters are saved in XML files with the *.entity* extension. As with the OPE files used in the past, entity files reference the systems that an entity supports. Systems continue to be specified in system definition files in MTL format. Other SMS files such as damage files, detection files, formation files, and so on have not changed from previous releases.

The Entity Editor has been updated to use these new entity files. Although the physical layout has changed, for the most part its usage remains the same as in the previous release. When you edit an entity you now just edit its entity file. When you create a new entity, you create a new entity file. The entity file also contains the object type, which used to be in *vrfSim.opd* and the menu details that used to be in *entity.mst*.

The OPD Editor no longer edits individual entity parameters. It lets you edit platforms and systems. Changes to platforms and systems affect all entities that use them.

The result of these changes is that to add a new entity, you create a new entity file in the Entity Editor. To migrate to a new release, you simply copy your custom entity files (and any files that they reference) into the SMS in the new release. The entity file will pick up any changes made to the platforms for the new release. You no longer have to update OPE files and *vrfSim.opd*. *vrfSim.opd* still exists, but simply contains variable bindings for the platforms.

You will still also have to update entity type mappings and model definitions for your custom entities.

5.1.1. Migrating Legacy Simulation Model Sets to the VR-Forces 4.2 Format

The Entity Editor has a built-in tool that can upgrade simulation model sets from VR-Forces 3.12 through 4.1.1 to the new format.

To migrate a simulation model set to VR-Forces 4.2:

1. Open the Entity Editor.
2. Choose **File** → **Upgrade Old Simulation Model Set**. The Select SMS to Upgrade dialog box opens.
3. Select the SMS that you want to upgrade.
4. In the New SMS Name text box, type a name for the upgraded SMS.
5. Click Finish.

The Entity Editor displays a status window. It copies files from the current default SMS to the new SMS. Then it converts files from the old SMS to the new format.

5.2. Changes to the Lua API

The Lua function vrf::getSimObjectsNear() no longer includes the calling entity in the list. Scripts that used this function might not work correctly any more. For example, if a script checks the size of the returned list and considers a list size of 1 as being empty (because it wants to ignore itself), it will now incorrectly think the list is empty when there is one other entity in the area. If you have a script that uses this function you should examine it to determine whether or not you need to change the code to account for the new behavior.

6. Migration to VR-Forces 4.3

This chapter describes migration issues from VR-Forces 4.2 to 4.3.

6.1. Changes to Simulation Model Sets

Prevous versions of VR-Forces provided one simulation model set (SMS) – *default.sms*. VR-Forces 4.3 introduces an aggregate warfare model. Because entity modeling for aggregate warfare is quite different from that in prevous versions of VR-Forces, a new SMS, *AggregateLevel.sms*, was required. Entity-level modeling, which descrbes the entity modeling in previous versions of VR-Forces, is now provided by *EntityLevel.sms*, which essentially replaces *default.sms*.

Although aggregate-level modeling and entity-level modeling are quite different, they do share some common objects. To minimize duplication of common objects in multiple SMSs and to support greater ease of SMS customization, VR-Forces 4.3 introduces the ability for SMSs to include other SMSs. The objects that are common to *EntityLevel.sms* and *AggregateLevel.sms* are contained in *base.sms*, which is included in both of the other SMSs.

If you load a scenario in VR-Forces 4.3 that references *default.sms*, VR-Forces automatically loads *EntityLevel.sms*. When you save the scenario, it updates it to the new SMS. Therefore, most legacy scenarios should migrate easily to VR-Forces 4.3. If you are using a customized SMS, you will have to evaluate your customizations and decide how to upgrade.

For general details about including SMSs in other SMSs, please see Section 5.3, "Simulation Model Sets", in *VR-Forces Configuration Guide*. For specific migration options, please see Section 5.3.7, "Migrating a Simulation Model Set to a New Release", in *VR-Forces Configuration Guide*.

6.2. API Changes

VR-Forces has the following changes to its APIs. For occasional updates to migration information, please check http://www.mak.com/support/migration-support.html.

- We have renamed a lot of header files for clarity, so that they will better match class names and make things easier to find. This release includes tools to help make this part of the upgrade process easy:
  - In the compatibility folder, there is a command line tool you can run to upgrade the header file references in your code. Instructions for how to do this are in *./compatibility/readme.txt*.
  - Alternatively, header files with the old names have been added to *./include/compatibility*. They point to the new files. You can add this directory to your project's include paths.
- In HLA, VR-Forces can change the transport type Data interactions at run time based on the content type of the message. Before sending a message, VR-Forces checks to see if it should be best-effort or reliable. If the current setting does not match what is required for the content type of the message, it makes an RTI service call to update the transport type.
- *DtVrfMessageInterface* has new methods to allow a plug-in to change transport types as well.
- The DtIfServerStatus message now contains, by back-end, the number of entities (by kind and domain) that are being simulated on that back-end. When a scenario is loaded, the signal_allScenarioObjectsDiscovered signal is sent from the *DtVrfScenarioManager*.
- By default, the following content types are sent reliable. Everything else is best-effort.
  - DtRemoveFromOrganizationType
  - DtSetSubordinateOrderType
  - DtAddToOrganizationType
  - DtIfObjectIndexMessageType
  - DtScriptedTaskResponseMessageType.

7. Migration to VR-Forces 4.4

This chapter describes migration issues from VR-Forces 4.3 to 4.4.

7.1. UUID Replaces Object Name for Identifying Objects

VR-Forces 4.4 uses a VR-Forces UUID as a unique identifier. Simulation objects and tactical graphics can now have the same names, but can still be identified as unique objects. This makes collaborative creation of scenarios and merging scenarios an easy process. As part of this transition, the Scenario Merge tool has been dropped. You can now merge scenarios simply by importing them into an open scenario in the VR-Forces front-end.

7.1.1. Loading Legacy Scenarios

When you load or import a scenario created in VR-Forces 4.3.x or earlier, all simulation objects are given new unique IDs. These unique IDs replace simulation object names in all locations where they are used, such as plans and the order of battle file.

When a legacy scenario is loaded (or imported) the DtUUID, DtRwUUID, and DtUUIDMarkingTextResolution managers convert object names to new UUIDs. A marking text to UUID mapping is created in the marking text, and all the marking text entries are then remapped to their new UUIDs. This will occur in any place that an object name is used as an identifier (plans, tasks, sets, and so on).

This process is transparent to the end user. Simulation objects are still be referenced by object name and all operations still work on object names from the users perspective.

7.1.2. API Changes

Wherever marking text used to be used to address objects, the object's UUID will be used. If you are sending messages to objects and are using the objectName() in a message, for example:

```
msg.setObjectName(obj->objectName());
```

you will now use the object's UUID:

```
msg.setUUID(obj->uuid());
```

It is still possible to use the marking text of the object, for example:

```
msg.setUUID(obj->markingText());
```

However, using old style code is not recommended. If you do not use the UUID, scenarios cannot use duplicate simulation object names and end users will lose the benefit of this change. For example, importing scenarios will not be guaranteed to work.

If you have written controllers, process state repositories, tasks, sets, and so on, that use DtRwString as an identifier for an object to apply an operation to, all you need to do to migrate your code is change `DtRwString` to `DtRwUUID`.

If you make this change, when you load a scenario, VR-Forces remaps what used to be the marking text to the new UUID representation. In the debugger you will see that the UUID also has a text pointer to the object (via marking text) that this represents.

If you have created methods that take an object name, such as:

```
void MyClass::setObjectName(const DtString&);
```

when you change your member to a DtRwUUID you should also change this prototype:

```
void MyClass::setUUID(const DtUUID&);
```

When looking up *DtVrfObjects* in the object manager you can now use:

```
lookupVrfObjectByUUID
```

i If you pass in marking text, this call also tries to look up the object by marking text. However, this is not recommended, because it will not allow the controller to work with similarly named objects.

7.2. Changes to VR-Link's DtVector

In VR-Link 5.2, *DtVector* has been split into *DtVector32* and *DtVector64*. (*DtVector64* is aliased to *DtVector*.) All functions that take velocity, acceleration, and embedded position information have been converted into *DtVector32*. This was done because DIS and the RPR FOM use 32-bit values for those fields and customers using our standard *DtVector* were seeing small rounding issues due to conversion to and from 32-bits when the data was passed over the network. Unfortunately, this also affects a large number of classes in VR-Link.

This change is unlikely to require changes to VR-Forces code. However, it is pointed out here just in case. For more information, please see *VR-Link 5.2 Release Notes* and class documentation.

8. Migration to VR-Forces 4.5

This chapter describes migration issues from VR-Forces 4.3 to 4.4.

8.1. API Changes

If your application or plug-in adds visualizer definitions to entity element definitions in the initDeModule, they now need to be added after the signal_postLoadVisual boost signal is sent from the DtVrfScenarioManager.

Scenario rewind now uses the scenario rollback functionality. Previously you would add callbacks for scenario rewound when a scenario was rewound from the GUI. You must now add callbacks for addPreLoadScenarioCallback() and addPostLoadScenario-Callback() in the DtCgf class.

Index

VT MAK
A company of VT Systems

Link - Simulate - Visualize

VRF-4.5-9-170221