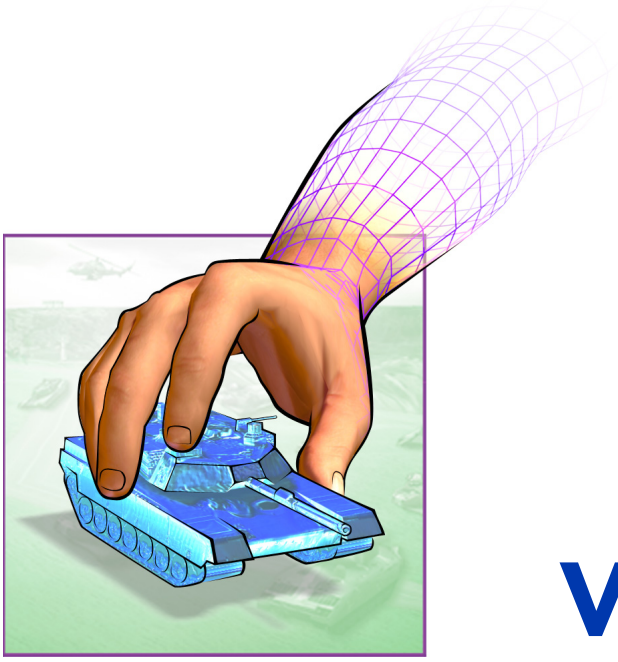
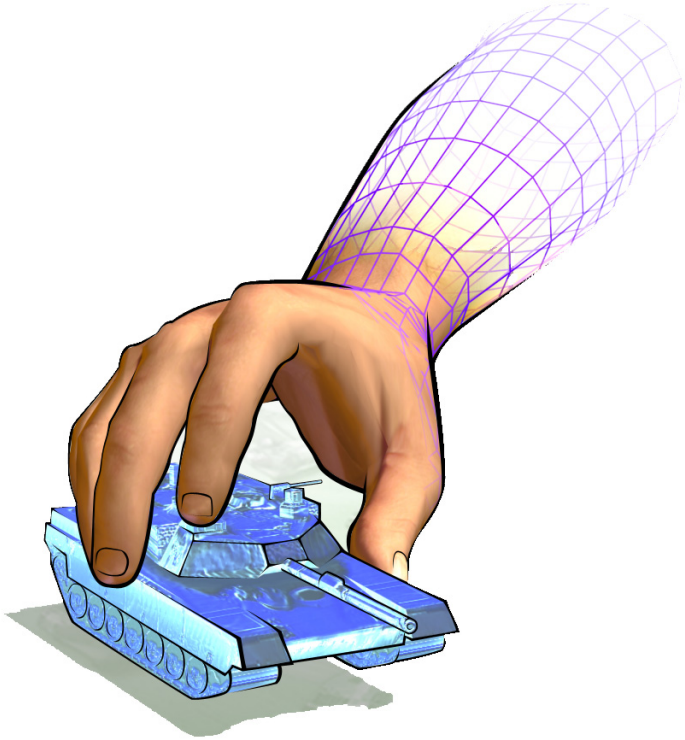


VR-Forces

Developers Guide

VT MÄK
A company of VT Systems



VR-Forces

Developers Guide

VT MÄK
A company of VT Systems

For third-party license information, please see "Third Party Licenses," on page xix.

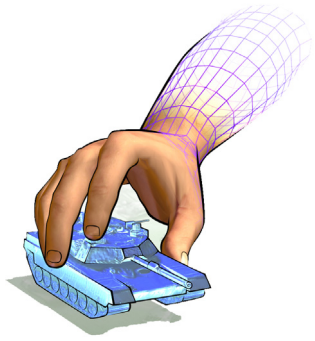
VT MÄK
68 Moulton St.
Cambridge, MA 02138 USA

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

info@mak.com

www.mak.com

Revision VRF-4.0-2-110120



Contents

Preface

Chapter 1 Introduction to the APIs

Chapter 2 The VR-Forces Simulation Engine

Chapter 3 Objects

Chapter 4 Entities

Chapter 5 Component Concepts

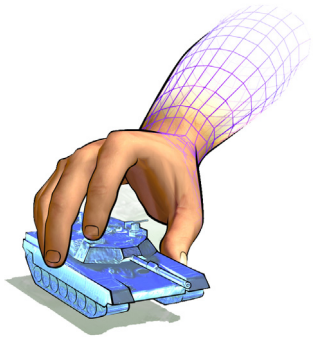
Chapter 10 Tasks, Sets, Commands, and Reports

Chapter 11 Plans

Chapter 16 Building the VR-Forces Back-End

Index

Index of Classes, Functions, and Header Files



Preface

This manual is written for persons who will develop software using the VR-Forces SDK. The manual assumes that you are familiar with your operating system, development environment, and C++.

Please see release documentation for updates to this manual and the latest information about your version of VR-Forces.

How This Manual is Organized

The chapters are organized as follows:

Chapter 1, *Introduction to the APIs* introduces you to the VR-Forces toolkit.

Chapter 2, *The VR-Forces Simulation Engine* describes the *DtCgf* class, which is the API to the simulation engine.

Chapter 3, *Objects* provides more conceptual detail about objects and explains how to create and manager them.

Chapter 4, *Entities* describes entities and addresses subjects such as echelon IDs, state repositories, parameters, and resources.

Chapter 5, *Component Concepts* explains explains component concepts and classes.

Chapter 6, *Creating Actuators and Controllers* explains how to create new actuators and controllers.

Chapter 7, *Sensors* explains how sensors work and how to create a new one.

Chapter 8, *Messages* describes the messaging mechanisms.

Chapter 9, *Entity Communication* explains VR-Forces radios and radio messages.

Chapter 10, *Tasks, Sets, Commands, and Reports* explains how to create new tasks and set commands for use in plans or tasks.

Chapter 11, *Plans* describes the Plan API.

Chapter 12, *The VR-Forces Remote Control API* describes this API, which allows you to control the simulation engine from a remote application, such as a graphical user interface.

Chapter 13, *The Terrain API* explains how VR-Forces accesses the terrain.

Chapter 14, *Reading and Writing Files* describes the mechanisms that allow VR-Forces objects to write themselves to file and read themselves from a file.

Chapter 15, *Miscellaneous Classes and Utilities* describes additional subjects you should be aware of.

Chapter 16, *Building the VR-Forces Back-End* explains how to use the sample project files provided with VR-Forces to build the back-end. It also explains how to deploy applications built using the VR-Forces SDK.

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 Getting Started Guide* is a quick introduction to VR-Forces. It covers the basics of installing VR-Forces, running a scenario, and creating a scenario. It focuses on helping new users avoid common mistakes.
- *VR-Forces Documentation Center* is your central starting point for accessing the VR-Forces manual set. It has a documentation roadmap, tables of contents for all manuals, and a master index, including the TDB Tool, to help you find references to subjects that are covered in two or more manuals or locate the manual in which a particular topic is discussed. All table of contents entries and index entries are live links to the manuals.
- *VR-Forces Users Guide* describes how to install VR-Forces and configure license management. It explains how to use the VR-Forces graphical user interface to view simulations and how to manage aspects of VR-Forces that are not directly related to creating and running scenarios.
- *VR-Forces Scenario Management Guide* explains how to create and run scenarios.
- *VR-Forces Configuration Guide* explains advanced features for configuring performance and how to edit VR-Forces configuration files. It includes documentation of the Entity Editor, OPD Editor, and Scenario Merge tools.
- *VR-Forces Developers Guide* explains how to use the VR-Forces simulation and remote control APIs. It focuses primarily on the simulation engine.
- *VR-Forces Front-End Developers Guide* explains how to extend or modify the VR-Forces graphical user interface (GUI).
- *VR-Forces Migration Guide* collates API migration information for each release in the VR-Forces 3.x series.

- Online help. The VR-Forces front-end, the OPD Editor, the Entity Editor, and the TDB Tool have online help accessible from the Help menu.
- Class documentation. Classes are documented in linked HTML pages.
- *VR-Forces Release Notes*.

MÄK Products

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

- MÄK RTI. An RTI (Run-Time Infrastructure) is required to run applications using the High Level Architecture (HLA). The MÄK 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 local entities, aggregate them into hierarchical units, assign tasks, set state parameters, and create plans that have tasks, set statements, and conditional statements. VR-Forces also functions as a plan view display for viewing remote entities 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 XR, and VR-Vantage IG — and the VR-Vantage Toolkit. VR-Vantage Stealth displays a realistic, 3D view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway.

  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 XR provides a common operational picture using the same 3D views as VR-Vantage Stealth, 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.

  The VR-Vantage Toolkit is a 3D visual application development toolkit. Use it to customize or extend MÄK'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.

- MÄK 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.
- MÄK Plan View Display. The Plan View Display (PVD) provides a 2D, "big picture" of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display.
- 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.

How to Contact Us

For VR-Forces technical support, information about upgrades, and information about other MÄK 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:	vrv-support@mak.com

Internet

MÄK web site home page:	www.mak.com
License key requests:	www.mak.com/support/licenses.php
Product version and platform information:	www.mak.com/support/productversions.php
For the free, unlicensed MÄK RTI:	www.mak.com/products/rti.php
Support FAQ:	www.mak.com/support/faq.php

Post

Send postal correspondence to:	VT MÄK 68 Moulton St. Cambridge, MA, USA 02138

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.
`{ }`	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 MÄK products run on both UNIX 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**.
	Indicates supplemental or clarifying information.
	Indicates additional information that you must observe to ensure the success of a procedure or other task.

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 *vrforces/doc* appears in the text as *./doc*.

Hardware and Operating System Naming Conventions

Unless otherwise specified, references to UNIX include Linux, regardless of the type of computer it is running on. References to PCs refer to Intel processor-compatible computers running a version of Microsoft Windows.

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 popup 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

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

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

libXML and libICONV

VR-Link and all MÄK software that uses VR-Link, links in libXML and libICONV. On some platforms the compiled libraries and header files are distributed with MÄK Products. MÄK 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/

pThreads Library

VR-Exchange links with the pThreads win32 library. The library is distributed under the GNU Lesser General Public License (LGPL). MÄK has made no modification to this library. For information about the pThreads win32 library please see: http://source-ware.org/pthreads-win32/

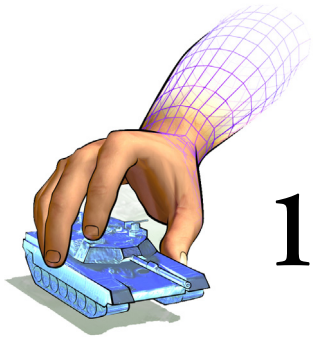
EHS, PCRE, and PME

The MÄK RTI links with the EHS, PCRE, and PME libraries. These libraries are distributed under the GNU Lesser General Public License (LGPL). MÄK has made modification to these libraries only to allow them to compile on the supported plat-forms. For more information about these libraries please see the following web sites:

- EHS: http://xaxxon.slackworks.com/ehs/
- PCRE: http://www.pcre.org/
- PME: http://xaxxon.slackworks.com/pme/index.html

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



1

Introduction to the APIs

This chapter introduces you to the VR-Forces toolkit APIs.

! Use of the VR-Forces API requires installation of VR-Link. To rebuild the GUI or to build any API examples that have a GUI, you must install Qt. Please see *VR-Forces Release Notes* or the product versions page on the MÄK web site for a list of the versions required. Please consult the VR-Link documentation for information about VR-Link.

1.1. Introduction to the VR-Forces Toolkit

VR-Forces consists of several APIs that you can use together or separately to build computer generated forces (CGF) applications and other distributed simulation applications. The APIs are:

- VR-Forces Simulation API
- VR-Forces Remote Control API
- VR-Forces Terrain API
- VR-Forces Graphical User Interface (GUI) API.

VR-Forces includes two executable programs, *vrfSim* and *vrfGui*, that use these APIs. The *vrfGui* executable is a graphical user interface that provides a two-dimensional plan view display and the ability to create and manage individual entities, aggregate entities, and control objects. The *vrfSim* executable is the simulation engine that responds to user actions in *vrfGui* and sends out the data that *vrfGui* displays. *VR-Forces Users Guide* explains how to use the VR-Forces executable programs.

Figure 1-1 illustrates how the VR-Forces APIs are used in the VR-Forces programs.

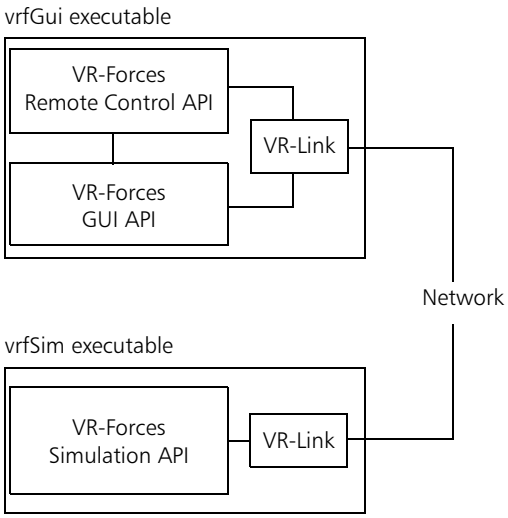


Figure 1-1. VR-Forces API implementation

1.1.1. The VR-Forces Simulation API

The VR-Forces Simulation API, also known simply as the VR-Forces API, provides a wide variety of classes and functions needed to implement a CGF simulation engine. The API gives you access to libraries of code for the elements required to execute a simulation, such as:

- Vehicle dynamics
- Control objects
- Sensors
- Controllers
- Weapons systems
- Radios
- Terrain representation
- Coordinate conversions
- Network messages
- Tasks
- Plans.

Furthermore, it provides a simulation engine infrastructure that ties all of this functionality together. The *vrfSim* application that comes with VR-Forces uses the VR-Forces Simulation API.

You can use the VR-Forces Simulation API to embed the CGF simulation engine into custom applications. You can customize or extend the functionality of the simulation engine, whether you are creating a plug-in to extend *vrfSim*, building your own extended *vrfSim* application, or as part of a larger application. An application that uses the VR-Forces Simulation API is said to be a VR-Forces-based application, or a VR-Forces application.

Chapter 2, *The VR-Forces Simulation Engine* provides an introduction to the VR-Forces API. It describes how to embed CGF functionality into a custom application, extend the *vrfSim* application through plug-ins, or rebuild the *vrfSim* application that is delivered with VR-Forces. The focus is on the *DtCgf* class – the top-level class in the VR-Forces API. Other chapters describe the other elements of the VR-Forces API – the classes that represent objects, simulation components, tasks, messages, and so on. The *DtCgf* class uses these classes to perform most of its work, and you will often need to subclass these classes if you want to extend or customize the behavior of the VR-Forces simulation engine.

1.1.2. The VR-Forces Graphical User Interface API

The VR-Forces Graphical User Interface (GUI) API provides the code libraries that implement the VR-Forces GUI. You can use this API to customize or extend *vrfGui*, or to embed its functionality into an instructor/operator station or other application.

The VR-Forces front-end is built using the VR-Vantage SDK. For complete details, please see *VR-Forces Front-End Developers Guide*.

1.1.3. The VR-Forces Terrain API

The Terrain API is a set of classes that enable applications to read, write, and query terrain databases. Through the Terrain API you can:

- Load terrain databases in MÄK GDB (native) format as well as a variety of external formats (such as OpenFlight, ESRI Shapefile, and CTDB)
- Save terrain databases to MÄK GDB format
- Access and manipulate the terrain geometry (polygon) and vector data
- Perform intersection tests with the terrain geometry and vector network
- Query the vector network for information about feature data.

The VR-Forces GUI uses the terrain API to access the geometry and vector data it needs to display the terrain data. The VR-Forces simulation engine uses the terrain API to perform terrain intersection tests for modeling ground vehicle movement, and line of sight in its sensor model.

Chapter 13, *The Terrain API* describes the Terrain API.

1.1.4. The VR-Forces Remote Control API

The VR-Forces Remote Control API is a set of classes that allow an application to control one or more remote VR-Forces applications. The VR-Forces GUI (the *vrfGui* executable) uses the VR-Forces Remote Control API to control the *vrfSim* application or other VR-Forces applications. The Remote Control API can also be used in custom VR-Forces front-ends, simulation managers, or instructor/operator stations.

The Remote Control API contains no GUI-related code. You can use it as easily in an application that uses scripts or command-line input as you can in a GUI-based application. You can use it in any application that needs to control remote VR-Forces simulation engines.

Chapter 12, *The VR-Forces Remote Control API*, describes the VR-Forces Remote Control API.

1.1.5. The VR-Forces APIs Share Classes and Libraries

Some classes and libraries are shared among two or more of the VR-Forces APIs. For example, both the GUI API and the Simulation API use the same set of classes to represent terrain, and both the Simulation API and the Remote Control API use the same classes to represent tasks and plans.

1.1.6. VR-Forces and VR-Link

The VR-Forces APIs rely on VR-Link to handle DIS and HLA networking. For example, the Remote Control API uses VR-Link to send control messages to VR-Forces. The Simulation API uses VR-Link to send out information about the objects it is simulating, and to find out about other objects in the simulated world that it needs to interact with. The GUI API uses VR-Link to find out about all of the objects in the exercise so that it can depict them on the map. A VR-Forces developer's license includes a license for VR-Link so that you can use VR-Link functions and classes directly from applications that use any of the VR-Forces APIs.

VR-Link provides a protocol-independent interface to DIS and HLA. If you use it, you do not need to know the lower level details of these protocols to build VR-Forces applications. VR-Forces applications can take advantage of VR-Link's tools for managing objects and interactions, such as reflected entity lists.

If you are not already familiar with VR-Link, we recommend that you review the conceptual material in *VR-Link Developer's Guide* and the description of the protocol independent interface.

1.2. Extending VR-Forces: Plug-ins or Stand-Alone Applications

If you want to modify or extend VR-Forces, there are two main approaches you can take: create a plug-in, or create a new stand-alone application (that is, rebuild *vrfSim* or *vrfGui*.)

Create a plug-in if you want to:

- Modify or upgrade features without changing the base installed version of VR-Forces. This lets you deliver changes to customers without requiring them to re-install the entire application.
- Integrate your extensions with the work of other developers.
- Easily include or exclude the new features depending on the needs of a particular simulation. Ideally, plug-in modules should be constrained to a specific, single piece of functionality. For example, if you want to add a new toolbar and an unrelated new menu command, use separate plug-ins. However, if the new features work together, such as a new menu and a toolbar with icons for the same set of features, use one plug-in.

Extend VR-Forces by rebuilding the application if you want to:

- Remove functionality from the system
- Make changes that affect classes that you may need to override
- Make extensive changes that require subclassing and installation of new objects using a factory rather than adding to the default system features.

The principal difference in code between a plug-in and a stand-alone application is whether you modify *main.cxx* or a *plugin.cxx*. The code you write to implement the new functionality will be the same. Switching between the plug-in approach and extending the application is straightforward.

1.3. Simulation, Terrain, and Remote Control API Examples

VR-Forces includes the following examples of how to use the Simulation, Remote Control, and Terrain APIs:

Adding New Tasks and Behaviors to Entities

- addActuator – shows how to add a new actuator component. Please see "Adding a New Entity Behavior (Creating an Actuator)," on page 6-2.
- addTaskSim – shows how to add a task that uses the User Task dialog box. Please see "Adding a User Task," on page 10-10.
- decideToGiveUpTask - shows how to extend an entity's behavior with the ability to determine when it ought to give up on carrying out a task.
- radarWarnRx – shows how to add a sensor component. Please see "Adding a Sensor Component," on page 7-5.
- stopToShoot – shows how to add a new controller component. Please see "Creating a New Controller," on page 6-6.
- subtask – demonstrates how to use subtasks.

Extending Entity State Information

- addAttr – shows how to extend the published state for an entity by extending the FOM. It is similar to the VR-Link addAttr example, but is in the context of extending a VR-Forces entity.
- addPSR – demonstrates how to add a process state repository to a component to allow its internal state to be checkpointed to the order of battle file.
- extendStateRepository - shows how to extend state repositories at the base class level (at the *DtVrfObjectStateRepository* level), so you do not have to derive from multiple leaf-level state repository classes to extend the state repositories of different kinds of entities.

Adding a Completely New Kind of Entity

- addEntity – shows how to add a new entity.

Accessing the Object Manager

- objectManager – shows how to override a high-level manager that maintains all simulated entities, aggregates, and control objects in VR-Forces. Please see the comments in the source files for documentation.

Creating a Simulation Engine in an Application

- simpleCGF – shows how to create an instance of the *DtCgf* in an application and use it to load and run a VR-Forces scenario.
- simplePlan – shows how to create an instance of the *DtCgf* in an application and use it to create an entity, construct a plan for the entity, and run the simulation.

Extending the Plan Language

- condition – explains how to add a new conditional expression to the VR-Forces plan editor. It adds a condition that tests to see if an entity is facing north.

Creating and Changing Terrain Programmatically

- addTerrainFeatures – shows how to add terrain features (vector data) to a terrain database using the terrain API.
- createTerrainDb – shows how to create a simple terrain database using the terrain API.
- modifyTerrainDb – shows how to modify (deform the terrain skin) an existing terrain database using the terrain API.

Using the Remote Control API

- remoteControl – shows how to implement an application that uses the Remote Control API.

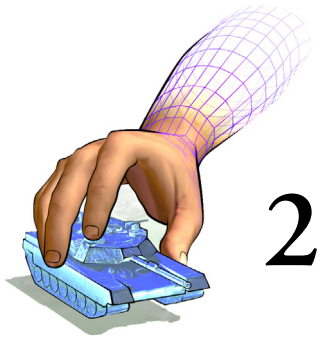
Network Utility

- vrfMsgDump – provides complete source code for a debugging utility that displays the contents of incoming VR-Forces interface messages. You can extend it to display any custom VR-Forces messages that you create.

! If you build your own version of *vrfMsgDump*, you can register messages that you have added to VR-Forces. However, while the binary version of *vrfMsgDump* provided with VR-Forces does not require a license, if you rebuild *vrfMsgDump*, your executable will require an additional VR-Forces license. Also, when you build your own *vrfMsgDump*, save the original version to a different directory or it will be overwritten.

VR-Forces also includes examples of how to use the GUI API. Please see the class documentation for descriptions of the examples.

The examples are in *./examples* (and in the workspace *examples.dsw*). They are described in this manual in the context of the activities they were designed to demonstrate. For example, the addActuator example shows how to add an actuator component, so it is described in Chapter 6, *Creating Actuators and Controllers*.



2

The VR-Forces Simulation Engine

This chapter describes the *DtCgf* class and shows how to embed the VR-Forces simulation engine in an application.

2.1. Getting Started with the VR-Forces Simulation API

The main entry point to the VR-Forces Simulation API is the *DtCgf* class, defined in *cgf.h*. An instance of *DtCgf* represents a simulation engine that can simulate objects, execute scenarios, and communicate with other HLA or DIS simulations. Its member functions allow you to create, load, or save scenarios, control scenario execution, and create, delete or modify objects. *DtCgf* relies on various other classes to perform most of its work, such as *DtVrfObjectManager*, *DtPhysicalWorld*, and so on. You need to understand these classes to customize or extend the functionality of the simulation engine, and we describe them in later chapters. However, to build default VR-Forces functionality into your applications, the *DtCgf* class is the only class you need to understand.

The source code for *vrfSim* (*./appsrc/vrfSimUser/main.cxx*) and for many of the VR-Forces API examples, does not appear to instantiate or use *DtCgf* directly. That is because it uses a higher-level class called *DtVrfApp* to handle creation and use of the *DtCgf*, based on command-line arguments, remote control messages, and so on. The *DtVrfApp* class is described in "Customizing or Extending the Simulation Engine," on page 2-8.

There are three primary approaches you can take to extend the VR-Forces simulation engine:

- Create a plug-in
- Build an extended *vrfSim* application
- Embed VR-Forces functionality in a third-party application.

2.1.1. Creating a Plug-in

A plug-in lets you extend VR-Forces without rebuilding the entire application. For a general discussion of using plug-ins, please see "Extending VR-Forces: Plug-ins or Stand-Alone Applications," on page 1-6. You can extend much of the *vrfSim* application through plug-ins. Through the plug-in interface you have access to:

- Vehicle dynamics
- Control objects
- Sensors
- Controllers
- Weapons systems
- Radios
- Terrain representation
- Coordinate conversions
- Network messages
- Tasks
- Plans.

For more information about creating plug-ins for the simulation engine, please see "Creating a Plug-in," on page 2-6.

2.1.2. Building an Extended *vrfSim* Application

While plug-ins provide a flexible way to deploy extensions to *vrfSim*, there may be cases in which you need to extend the *vrfSim* application. For example, you may need to extend the *DtCgf* class directly. "Customizing or Extending the vrfSim Application," on page 2-12, describes how to build an extended version of *vrfSim*.

2.1.3. Embedding VR-Forces in a Third Party Application

You can incorporate VR-Forces functionality in a third-party application (an application other than *vrfSim*) by creating an instance of a *DtCgf* application. The simpleCgf and simplePlan examples demonstrate how to do this. For more information, please see Section 2.2, "Using the DtCgf Class".

2.2. Using the DtCgf Class

The following example is a minimal VR-Forces application. This example can execute whatever arbitrarily complex scenario is described by *foo.scn*, and will act as a full-fledged CGF participating in an HLA federation.

Example 2.1

```
#include "cgf.h"

int main()
{
   DtCgf cgf(DtSimulationAddress(1, 3001));
   DtExerciseConn exConn("VR-Link", "Minimal");
   cgf.init(&exConn);

   cgf.loadScenario("foo.scn");
   cgf.run();
   while (1)
   {
      cgf.tick();
   }
}
```

This example:

- Creates and initializes an instance of *DtCgf*
- Uses its loadScenario() function to instruct it to load a VR-Forces scenario
- Calls its run() function to tell it to start advancing simulation time
- Calls tick() periodically.

The tick() member function is where *DtCgf* does its real work – performing the operations required for a single simulation frame. Here, information about remotely simulated objects is gathered, sensor input is computed, tasks and behaviors are executed, vehicle dynamics are calculated, and current state is sent to other simulations. Every VR-Forces application needs to tick *DtCgf* once per simulation frame.

2.2.1. Calling the *DtCgf* Constructor

DtCgf's constructor expects a *DtSimulationAddress* as an argument. The simulation address is an identifier used to distinguish a VR-Forces application from other simulations that may be playing in the same exercise. We need to be able to identify a particular VR-Forces application so that VR-Forces remote control applications like *vrfGui* can address a specific simulation engine when it asks for a new object to be created. Simulation addresses are also used in a scenario's object map file to indicate which simulation engine is responsible for simulating each object in the scenario.

DtSimulationAddress is a VR-Link class (defined in *hostStructs.h*) that consists of a site ID and an application ID, both integers.

The second, optional, argument to the *DtCgf* constructor is a session ID. Passing a value other than the default is required only when you have multiple VR-Forces applications in the same exercise, and they are using different terrains. Please see Section 3.2.3, "VR-Forces Sessions", in *VR-Forces Users Guide* for a discussion of how to use session IDs. Please see "VR-Forces Sessions," on page 15-15, for information about how session IDs are implemented in the VR-Forces API.

2.2.2. Initializing A DtCgf

Before you can use an instance of *DtCgf*, you must call its init() function. The init() function requires a pointer to a *DtExerciseConn* – the VR-Link class that represents an application's connection to a DIS or HLA exercise. *DtCgf* and its child classes use the *DtExerciseConn* for all communication with other simulations in the exercise. *DtExerciseConn* has several constructors, and a variety of constructor arguments, whose use depends on whether you are building a DIS or HLA application. Please see VR-Link documentation for details. In Example 2.1, we passed the name of an HLA federation execution (VR-Link), and a name for the federate (Minimal).

The second, optional, argument to init() is a pointer to a *DtVrfCreator* (defined in *vrfCreator.h*). An instance of *DtVrfCreator* is used by *DtCgf* to create instances of the various classes that it needs. If you are not customizing the simulation engine (adding custom vehicle dynamics code, new messages or tasks, and so on), then you can just pass the default value of NULL, which is an indication to the *DtCgf* that it should use the default creator. Please see "The VR-Forces Creator," on page 2-11, for more information about *DtVrfCreator*.

2.2.3. DtCgf Member Functions

This section describes some of the important *DtCgf* member functions.

The *./examples* directory has two examples of using *DtCgf*. The simpleCgf example shows how to create a scenario programmatically. The simplePlan example shows how to create a plan programmatically.

Functions for Managing Scenarios

DtCgf has scenario management functions, such as loadScenario(), closeScenario(), newScenario(), and saveScenario(). Calling saveScenario() for example, causes your application to save a scenario file, along with an order of battle file, a plan file, and an object map file that contain information about the objects that your application is simulating.

These functions apply only to the local VR-Forces simulation engine. Other VR-Forces applications in your exercise are not affected when you call these functions. If you are using multiple simulation engines to simulate a scenario, an external application must coordinate them (usually through the remote control API). Please see Chapter 12, *The VR-Forces Remote Control API*, for more information.

Functions for Running a Scenario

A VR-Forces simulation engine is always either in a running or paused state of execution. Calling run() and pause() toggles between these states. When in paused mode, very little happens during *DtCgf*'s tick(). Time does not advance, and therefore simulation does not need to occur.

Calling rewind() causes the *DtCgf* to reinitialize to the state dictated by the scenario file you have loaded (or restart from a blank slate if you were not working from an existing scenario.)

Functions for Managing Objects

DtCgf contains functions that instruct the simulation engine to create, delete, or task various simulation objects. For example, when the *vrfSim* application gets a message from a *vrfGui* application indicating that a user has created an entity from the GUI, it calls DtCgf::createEntity(). Similarly if you want to programmatically create a locally simulated entity, you can call createEntity() from within your application. (Modifying an entity's state is typically done through the entity object itself rather than through the *DtCgf*. Please see Chapter 10, *Tasks, Sets, Commands, and Reports* for details.)

2.3. Creating a Plug-in

When you create a back-end plug-in, you must include *vrfPluginExtension.h* in the plug-in module. The *vrfPluginExtension.h* header file contains a description of the entry points available to plug-ins for initialization and cleanup.

Use the following plug-in callbacks when you create a plug-in. At least one of these callbacks must be present in a plug-in for it to be loaded:

```
extern "C" {
  // Implement this plugin routine in order to place your object
  // subclasses into the factory.  This method
  // should be used for assigning controller, main objects
  // (DtVrfObjectManager subclasses, and so on), actuators
  // sensors. This function should be implemented if you wish to
  // override or replace functionality
  // that is defined by default within the system.
  DT_VRF_DLL_PLUGIN bool DtInitializeVrfPlugin(DtCgf* cgf,
    DtVrfPluginInformation&)
  {
    return true;
  }

  // Implement this call if you want to get access to those objects
  // created by the DtCgf.  This is called as the last step of the
  // DtCgf init() method  This function should be implemented if
  // you want to work with objects that have been created from the
  // factory as part of the DtCgf init() method call (for example,
  // DtVrfObjectManager)
  DT_VRF_DLL_PLUGIN bool DtPostInitializeVrfPlugin(DtCgf* cgf,
    DtVrfPluginInformation&)
  {
    return true;
  }
}
```

The DtVrfPluginInformation parameter supplied to these entry points allows you to identify your plug-in. This information is recorded to the plug-in's log file and *vrfSim.log* file as each plug-in is loaded and initially used. This structure has the following information:

```
typedef struct _DtVrfPluginInformation
{
  DtString    pluginName;
  DtString    pluginVersion;
  DtString    pluginDescription;
  DtString    pluginCreator;
  DtString    pluginCreatorEmail;
  DtString    pluginContactWebPage;
  DtString    pluginContactMailingAddress;
  DtString    pluginContactPhone;
  DtString    vrfCurrentVersion;
} DtVrfPluginInformation;
```

Filling in this structure is optional. If it is not filled in, the plug-in will still load and be used.

The following entry point is optional and should be used only if the plug-in needs to clean up memory when it is unloaded:

```
extern "C" {
  // Implement this method to be notified when the plugin is about to be
  // unloaded so that you can clean up statically allocated memory
  DT_VRF_DLL_PLUGIN void DtUnloadVrfPlugin()
  {
  }
}
```

The following example is a plug-in for *vrfSim*. The plug-in overrides the normal behavior of an actuator.

```
#include "vrfPluginExtension.h"
#include "myActuator.h"

#include "cgf.h"
#include "factoryMgr.h"
#include "compTypes.h"

extern "C" {

  DT_VRF_DLL_PLUGIN bool DtInitializeVrfPlugin(DtCgf* cgf,
    DtVrfPluginInformation& info)
  {
    cgf->factoryManager()->componentFactory()->addCreatorFcn(
      DtAutomotiveActuatorType, MyActuatorComponent::creator);

    info.pluginName = "Add Actuator";
    info.pluginVersion = "1.00";
    info.pluginCreator = "MAK Technologies";
    info.pluginCreatorEmail = "sales@mak.com";
    info.pluginContactWebPage = "www.mak.com";
    info.pluginContactMailingAddress = "68 Moulton Street - Cambridge,
      MA 02138";
    info.pluginContactPhone = "(617) 876 8085";

    return true;
  }
}
```

This example:

- Registers a create function for a derived kind of automotive actuator
- Fills out information associated with the plug-in (to be displayed in the console when the plug-in is loaded).

The *DtCgf* class is passed in to the plug-in through the DtInitializeVrfPlugin entry point. In this function, you can register the creation functions for the classes you want to override. You can also register callback functions on important events, such as opening and closing scenarios. For details, please see the *cgf.h*.

2.3.1. Loading Plug-ins

Plug-in libraries are placed in the *./bin* directory. By default, *vrfSim* does not load any plug-ins. You can specify which plug-ins to load on the command line or in an xml file in the *./plugins* directory. The preferred way to configure loading of plug-ins is through the Plugins dialog box in the front-end. For details, please see Section 5.10, "Managing Plug-ins", in *VR-Forces Users Guide*.

Many of the examples included with VR-Forces are plug-ins. Please see the VR-Forces Class Documentation for the complete list of examples.

2.4. Customizing or Extending the Simulation Engine

"Using the DtCgf Class," on page 2-3, described how to use the simulation engine in an application, through *DtCgf*'s API. This section explains how *DtCgf* is implemented, and describes how to extend or override various elements of its implementation.

DtCgf relies on a variety of managers and other classes to perform its job. We have already seen that it relies on *DtExerciseConn* for DIS and HLA connectivity. Some other classes that *DtCgf* relies on for various functions are:

- *DtObjectManager* (defined in *vrfObjMgr.h*) is responsible for simulating local objects, and maintaining a database of remote objects. (For details, please see Chapter 3, *Objects*.)
- *DtSimManager* (*simMgr.h*) manages the exercise clock, schedules timed events, and provides other simulation management functions. (For details, please see Chapter 15, *Miscellaneous Classes and Utilities*.)
- *DtVrfObjectPlanAdministrator* (*vrfObjPlanAdministrator.h*) and *DtVrfObjectPlanManager* (defined in *vrfObjPlanMgr.h*) execute the plans for locally simulated entities. (For details, please see Chapter 11, *Plans*.)
- *DtTerrainInterface* (*terrainDb.h*) provides functions for querying a terrain database about its geometry and features. (For details, please see Chapter 13, *The Terrain API*.)
- *DtVrfStatusPublisher* (*vrfStatusPub.h*), communicates the existence and status of a VR-Forces application to other VR-Forces simulation engines, and to VR-Forces control applications like *vrfGui*. (For details, please see Chapter 15, *Miscellaneous Classes and Utilities*.)

These classes, in turn, rely on other lower-level classes to implement their functionality. For example, *DtObjectManager* uses a *DtVrfObject* to represent each object that exists in the simulation. *DtVrfObject* relies on *DtSimComponent*, its subclasses *DtActuatorComponent*, and *DtControllerComponent*, and all of their numerous subclasses to help it implement a sensor-actuator-controller model of simulation. *DtVrfObjectPlanManager* (and some of the *DtSimComponents*) use *DtPlan*, *DtSimStatement*, *DtSimTask*, *DtSimCondExpr,* and their subclasses to represent plans that can be assigned to *DtVrfObjects*, as well as the tasks, statements, and conditional expressions that are used to build them. Figure 2-1 illustrates this architecture.

If you customize the VR-Forces simulation engine, you will almost always derive a subclass from one or more of these classes to implement the desired functionality. Then you will register your subclasses with VR-Forces. To register custom subclasses with VR-Forces, you must understand the VR-Forces factory mechanism, and the *DtVrfCreator* class. They are described in the following sections.

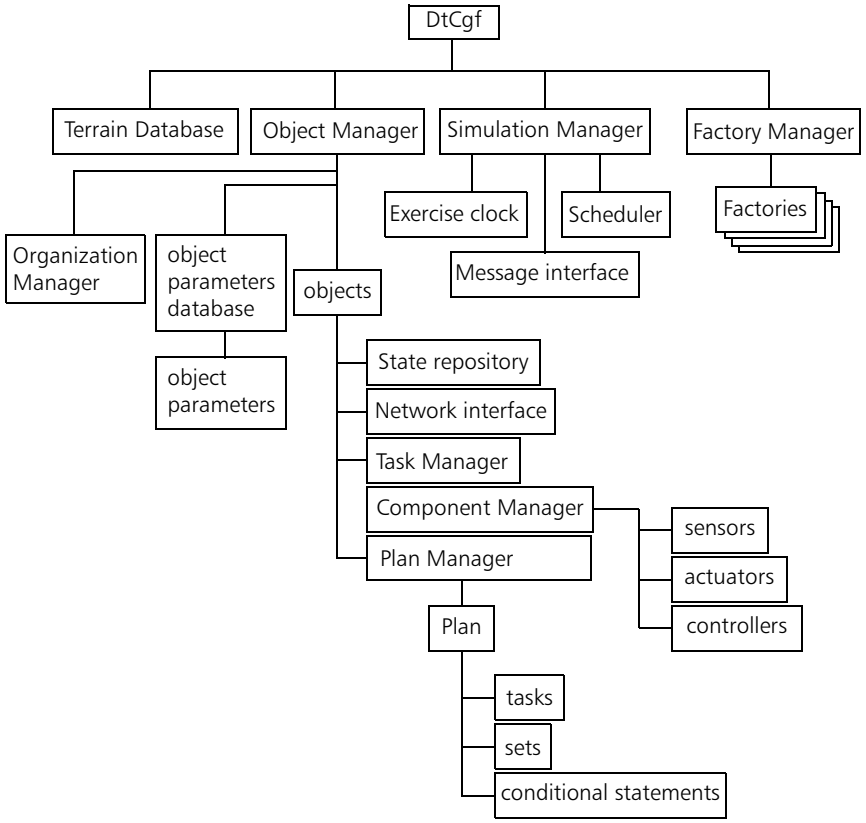


Figure 2-1. VR-Forces architecture

2.4.1. VR-Forces Factories

VR-Forces uses factories to pick the components it needs to create an object. There are many places in the VR-Forces architecture where the simulation engine needs to choose among many subclasses of a particular base class to properly configure some object. For example, when the simulation engine initializes a new entity, it needs to create all of the components necessary to simulate it – the appropriate kind of actuator for vehicle dynamics, controllers to implement tasks, and so on. Since, for example, each kind of actuator is represented by its own subclass of the base *DtActuatorComponent*, VR-Forces needs a way to choose the right subclass to instantiate. It uses factories.

A typical factory maintains a list of associations between class identifiers (such as strings or enumerations identifying the various kinds of objects among which we are choosing), and creator functions that can be used to instantiate the various subclasses. (A creator function is a static function of a class that returns a new'ed instance of that class.) To tell VR-Forces about a new subclass that you have written, you must register the creator function for that class with the appropriate factory.

You can get a pointer to the appropriate factory through *DtCgf*'s Factory Manager, which is a container for all of *DtCgf*'s factories. *DtCgf* has an accessor function called factoryManager() that returns a pointer to its *DtFactoryManager*. *DtFactoryManager* (defined in *factoryMgr.h*), in turn, provides accessors for all of the factories – taskFactory(), componentFactory(), and so on. Registering a creator function for a new subclass of *DtActuatorComponent*, for example, would look like this:

```
cgf.factoryManager()->componentFactory()->addCreatorFcn(
   DtAutomotiveActuatorType, MyActuatorComponent::create);
```

In this case, you are telling VR-Forces that an instance of your custom actuator component class (MyActuatorComponent) should be used when the simulation engine needs an "automotive actuator". Chapter 5, *Component Concepts*, has more information about creating custom components, and registering them with the component factory. Details of deriving from other classes, such as *DtSimTask* and *DtSimMsg*, are in the chapters devoted to those classes.

You should register your subclasses with the appropriate factories immediately after *DtCgf* is initialized, to insure that the factory is configured correctly before it is used by the simulation engine to create anything.

2.4.2. The VR-Forces Creator

The previous section describes how VR-Forces uses factory classes during execution to create instances of various classes when it needs them. *DtVrfCreator* is a kind of factory class that *DtCgf* uses to create its top-level managers and other important classes during initialization (within init()). For example, when the *DtCgf* creates a *DtVrfObjectManager*, it calls *DtVrfCreator*'s createVrfObjectManager() function to do so, rather than just calling `new DtVrfObjectManager`. This is to allow you to override the way VR-Forces objects are constructed. If you subclass *DtVrfObjectManager*, you can instruct *DtCgf* to use your subclass by configuring a *DtVrfCreator* so that createVrfObjectManager() returns a new'ed instance of your subclass. This is usually done by registering a creator function for your new subclass with a *DtVrfCreator*, before passing that creator to *DtCgf*'s init() function, for example:

```
// Your subclass of DtVrfObjectManager
{
public
  // You would probably override some virtual functions here to
  // modify behavior, and so on.
  ...

  // The creator function to be registered with DtVrfCreator
  static DtVrfObjectManager* create(DtSimManager* simMgr)
  {
    return new MyObjManager(simMgr);
  }
};
```

Now, when you create your *DtCgf*:

```
int main()
{
  // Instantiate a default creator
  DtDefaultVrfCreator creator;

  // Register the creator for your MyObjManager class
  creator.setObjectManagerCreator(MyObjManager::create);

  // Instantiate the DtCgf
  DtCgf cgf(DtSimulationAddress(1, 3001));
  DtExerciseConn exConn("VR-Link", "Minimal");

  // Pass a pointer to your creator to DtCgfinit()
  cgf.init(&exConn, &creator);
}
```

Notice that the creator that we instantiated is a *DtDefaultVrfCreator*, and not the base *DtVrfCreator*. This is because we do not want to have to manually register creator functions for all of the default managers, factories, and so on. The base *DtVrfCreator* starts empty – with no creator functions registered. But when you instantiate a *DtDefaultVrfCreator*, it already has self-registered creator functions for default managers, and so on. You only need to register creators for those classes for which you would like to provide alternate implementations.

Another way of configuring a *DtVrfCreator* so that it returns an instance of your subclass is to subclass *DtDefaultVrfCreator*, and override functions like createVrfObjectManager(). But the creator registration method described in this section is usually preferable because it is a bit simpler and allows more modularity. (Several modules can register creators with *DtVrfCreator* without coordinating with each other.)

2.5. Customizing or Extending the vrfSim Application

The *vrfSim* application is a VR-Forces application that responds to control messages sent by a tool such as the VR-Forces GUI (*vrfGui*), by making appropriate calls to the VR-Forces Simulation API (that is, *DtCgf*.) *vrfSim* is fairly simple in the sense that it does not add a lot of code to what is provided by the VR-Forces API, but it is very powerful because it incorporates the full functionality of the VR-Forces toolkit. It represents a self-contained CGF simulation engine, ready to be controlled by a remote control application, such as *vrfGui*. It accepts command-line options for setting modes and parameters, and it is configurable, because its capabilities and behavior are driven by the object parameter database and other files.

Many VR-Forces developers need an application that works just like *vrfSim* (keeps its command-line options, and ability to be driven by a remote VR-Forces GUI), but that replaces some default dynamics model with custom code, adds the capability to perform additional tasks, supports some new kind of sensor, and so on. Therefore, we provide source code and project files or makefiles for *vrfSim* in *./appsrc/vrfSimUser*. You can edit *vrfSimUser*'s *main.cxx* file, use the VR-Forces Simulation API to modify its behavior, and rebuild *vrfSim*.

2.5.1. The DtVrfApp Class

In *main.cxx*, all of the functionality of the *vrfSim* application is encapsulated in the *DtVrfApp* class, defined in *vrfApp.h*. The main() function just instantiates *DtVrfApp*, calls init() on it, and then executes mainLoop(). The application's functionality was put into a class, as opposed to directly in main(), so that other applications (like the VR-Forces examples) could inherit *vrfSim*'s functionality without duplicating code. *DtVrfApp* is not really a part of the VR-Forces simulation engine. It is a class that uses the VR-Forces simulation engine through the VR-Forces Simulation API.

When you modify or extend the *vrfSim* application, we recommend that you avoid making modifications to *vrfApp.cxx* or *vrfApp.h*, so that you do not have to merge your changes into future versions of these files when you upgrade. In most cases, you should be able to just obtain a pointer to the *DtVrfApp*'s *DtCgf* object (using its cgf() member function from within main()), and configure it appropriately using *DtCgf*'s member functions, managers, or factories.

Most of the VR-Forces API examples take this approach. For example, addActuator, which shows how to add a custom dynamics model for ground vehicles, adds just the following line to *main.cxx* to register its new subclass of *DtActuatorComponent* with the VR-Forces simulation engine:

```
app->cgf()->factoryManager()->componentFactory()->addCreatorFcn(
   DtAutomotiveActuatorType, MyActuatorComponent::creator);
```

Overriding DtVrfCreator

When your application needs to configure *DtVrfCreator*, as described in "The VR-Forces Creator," on page 2-11, (for example, when you need to register a custom subclass of a class like *DtVrfObjectManager*), you can pass your *DtVrfCreator* pointer to DtVrfApp::init(), which will pass it on to DtCgf::init(), as follows:

```
int main()
{
   DtVrfApp app(argc, argv);

   // Instantiate a default creator
   DtDefaultVrfCreator creator;

   // Register the creator for your MyObjManager class
   creator.setObjectManagerCreator(MyObjManager::create);
   app.init(&creator);
...
}
```

Subclassing DtVrfApp

When you need to change the behavior of *DtVrfApp* in a custom application (for example to add or remove command-line options), we recommend subclassing *DtVrfApp* and overriding the relevant functions.

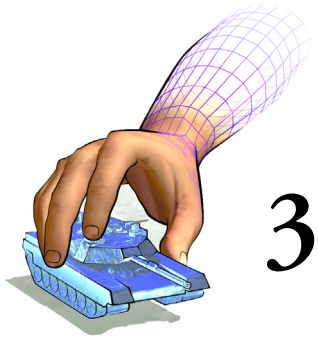
If you are embedding VR-Forces functionality in a larger application, you usually will not want to use *DtVrfApp*, since it represents an application that does nothing but execute a VR-Forces simulation engine, and therefore implements a simple main loop for you. Instead, instantiate the classes that you need explicitly within your application, for example, *DtCgf*, *DtCgfDispatcher*, *DtBatchManager*, and so on.

2.6. Checking for VR-Forces Licenses at Runtime

VR-Forces can check the availability of VR-Forces licenses at runtime. This lets you write code to handle a situation in which an expected license is unavailable for some reason. For example, you might want to use the opportunity to pop up a warning to alert the user. You can check for the availability of back-end, remote control API, or GUI licenses. For details, please see *vrfCheckLicense.h*.

The following example shows how to embed a VR-Forces back-end in a manned simulator. Before trying to create an instance of the top-level back-end class (*DtCgf*), it checks to see if a back-end license is available. If not, it prints a warning message.

```
#include <vrfCheckLicense.h>

...

if (DtHaveVrfSimLicense() == true)
{
  DtCgf* cgf = new DtCgf(addr);
  cgf->init(exerciseConn);

  ...
}
else
{
  DtWarn << "Error checking out a VR-Forces back-end license."
    << std::endl;
  ...
}
```



3

Objects

This chapter discusses objects in general, including how to create new objects, and in particular, describes control objects.

3.1. The Object Manager and Simulated Objects

All VR-Forces applications have an instance of a *DtVrfObjectManager* (*vrfObjMgr.h*). It is one of the top-level managers maintained by the *DtCgf* class. The Object Manager creates and maintains local objects. It also instantiates local instances of remotely simulated objects. Generally speaking, a simulated object is an individual entity, an aggregate entity, or a control object. The Object Manager maintains lists of each object type. It provides a callback mechanism to notify applications when objects are added or deleted. If you need to know about an object, you go to the Object Manager.

The Object Manager reads the object parameter database and uses this information when objects are created. It is also responsible for reading and writing the order of battle file. The Object Manager registers with the Simulation Manager's message executive to receive all *DtDeliverRadioMessage* messages, and directs them to the appropriate radio channel.

The Object Manager monitors the network for state updates from remote objects in an exercise and for VR-Forces-specific state information. When it detects the presence of a new remote object, it creates a new remote *DtVrfObject* to represent it and adds it to list of objects to be managed.

The Object Manager is responsible for saving and restoring the state of locally simulated entities as part of a scenario or checkpoint. It reads and writes the order of battle file. It uses the *DtReaderWriter* mechanism to save/restore itself to file. (For details, please see Chapter 14, *Reading and Writing Files*.) Almost all of the data members in the state repositories can read and write their data, and therefore are saved as part of an object's entry in the order of battle file.

3.1.1. Creating the Object Manager

The Object Manager is one of the manager objects created by the *DtCgf* object. The *DtCgf* object takes responsibility for deleting the Object Manager at the end of the simulation. You can create your own derived Object Manager. For a description of the process, please see "The VR-Forces Creator," on page 2-11.

3.1.2. Simulated Objects

All simulated objects are instances of *DtVrfObject* (*vrfObject.h*). The different types of simulated objects (entities and control objects) are not distinguished by the classes used to instantiate them. For example, there is no control object class or ground vehicle class. Instead, object types are distinguished by the subcomponents that they have, in particular, their state repository.

In this manual, when we use the term object, unless the context clearly indicates otherwise, we are referring to simulated objects (*DtVrfObjects*).

All objects have:

- An object type – an 8-digit superset of the DIS/RPR FOM entity type enumerations. For example, the object type for an M1A2 tank is 1 (1 1 225 1 1 3 -1).
- Parameters – values that a user can configure, and which are stored in an object parameter database.

DtVrfObjects contain instances of the following items through a has-a relationship:

- Identifiers:
  - Object name (*DtString*) – a unique string used to identify the object within the exercise, and which can persist from session to session
  - Object identifier (*DtObjectIdentifier*)
  - Global object designator (*DtGlobalObjectDesignator*).
- Object Label (*DtString*)- a text string that can persist beyond a particular run of an exercise and does not have to be unique. (For details, please see "The Object Label," on page 3-11.)
- A configurable set of the following subcomponents:
  - State repository (*DtVrfObjectStateRepository*)
  - Network interface (*DtSimObjectNetInterface*)
  - Organization Manager (*DtOrganizationManager*)
  - Task Manager (*DtSimTaskManager*)
  - Plan Manager (*DtVrfObjectPlanManager*)
  - Component Manager (*DtSimComponentManager*).

All objects have a state repository and network interface. Control objects have only these two subcomponents. Only entities (platform entities and aggregates) use the other subcomponents. State repositories and network interfaces are described in this chapter. The other subcomponents are described in Chapter 4, *Entities*.

3.1.3. Local and Remote Objects

From the point of view of a particular VR-Forces application, a local object is any object that it simulates. A remote object is an object that is received over the network from another application. A VR-Forces application participating in an exercise may know about both local and remote objects.

There are two categories of remote objects – objects simulated by VR-Forces applications, and objects simulated by applications other than VR-Forces, such as the VR-Link F18. When VR-Forces learns about a non-VR-Forces remote object, it knows only what is transmitted in the standard DIS or HLA message. By contrast, VR-Forces objects share additional state information over the network, such as echelon ID and label.

Figure 3-1 illustrates how objects simulated by different VR-Forces applications and a non-VR-Forces application appear to VR-Forces applications. In this figure, application 1:3001 simulates a waypoint and a fixed-wing entity. Application 1:3002 simulates a tracked entity. A non-VR-Forces application is also running on the network.

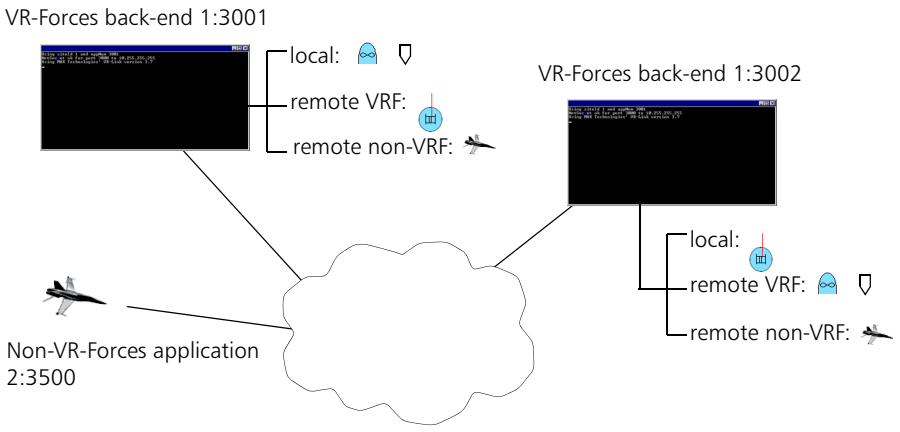


Figure 3-1. Local and remote objects as seen by VR-Forces

3.1.4. Spatial Organization of Objects

The *DtVrfObjectManager* class contains a *DtSpatialVrfObjectManager*, which it uses to spatially organize specified *DtVrfObjects*. A filter function predicate determines which *DtVrfObjects* are spatially sorted, to let you easily change the sorting in a derived class. Additionally, the creation, initialization, and use of the *DtSpatialVrfObjectManager* class is all performed in the appropriate functions, allowing for user modification.

A simple algorithm is used to determine the spatial organization configuration (number of subdivisions in each dimension). Also, all *DtVrfObjects* are stored in the Spatial Manager, including routes, waypoints, and so on.

To access the spatially sorted vrfObjects, the getVrfObjects() functions are provided to get all vrfObjects that may intersect the specified primitive (currently only *DtChord* or *DtExtent*). The resulting set contains all objects that may intersect the chord or extent. Not all are guaranteed to do so, but the set will not be missing any that actually do. If the exact intersection is required, as for line-of-sight testing in the vrfObjectStateRepository, then further refinement is required.

3.2. How the Object Manager Creates an Object

The Object Manager needs to create an instance of *DtVrfObject* for each simulated object. It does so as follows:

1. The Object Manager calls DtVrfObject::createObject() to create an instance of a new *DtVrfObject*. The *DtObjectType* and a Boolean flag, indicating whether the object is local or remote, are cached by the object for further construction in its virtual create() function.
2. The Object Manager calls DtVrfObject::create(). Using the *DtObjectType* as the key, it looks up the object parameter database entry for the object. The appropriate subcomponent list is chosen based on the local/remote flag cached by the object in its constructor. (For details, please see "How the Object Manager Chooses an Object's Subcomponents," on page 3-7.) Each of the subcomponents is created through a factory method, given the corresponding type string specified in the subcomponent list.
3. The Object Manager calls DtVrfObject::init(). This invokes the init() functions on each of its subcomponents. Upon completion of this call, the object is fully initialized and ready to be ticked.
4. The Object Manager adds the new object to its list of objects to be managed.

The Object Manager ticks all of its objects (local and remote) in its tick() function.

For a description of how you can add object types, please see "Creating a New Local Object," on page 3-28 and "Creating an Entity," on page 4-8.

3.2.1. How the Object Manager Chooses an Object's Subcomponents

This section expands on step 2 in the previous section. When the Object Manager creates an instance of a *DtVrfObject*, it needs to configure it with different subcomponents, depending on whether or not the object is local or remote. The Object Manager knows which subcomponents to use because the object parameter database entry for each object specifies two lists of subcomponents, one for local objects; the other for remote objects. The strings in the subcomponent list correspond to the C++ objects that make up a *DtVrfObject* (through a has-a relationship).

When the Object Manager creates a new object, it looks up the entry in the object parameter database for the object type to be created. The Object Manager passes the object's local or remote status into the *DtVrfObject* constructor. When the new object gets initialized by the Object Manager, it:

1. Looks up its entry in the object parameter database.
2. Determines if it is a local or remote object.
3. Instantiates the set of subcomponents from the corresponding subcomponent list (local-objects or remote-objects). Each item in the subcomponent list is created from a factory.

For example, if the object is a local object and the state-repository component lists the string **ground-entity-state-repository**, the Object Manager knows that it has to create a *DtGroundEntityStateRepository* for this object.

Figure 3-2 illustrates this process using the subcomponent list for a ground vehicle. The following list matches the subcomponent strings to their base class. To find the subclasses for the various subcomponents, please see the class documentation for the base class and navigate to the subclass that you are interested in. Each subclass has a type() member function that returns the type string for the component. The type is registered with the appropriate factory and is what gets specified in the object parameter database.

- state-repository: *DtVrfObjectStateRepository* (*./doc/classdoc/classref/class_dt_vrf_object_state_repository.html*)
- net-interface: *DtSimObjectNetInterface* (*./doc/classdoc/classref/class_dt_sim_object_net_interface.html*)
- task-manager: *DtSimTaskManager* (*./doc/classdoc/classref/class_dt_sim_task_manager.html*)
- plan-manager: *DtVrfObjectPlanManager* (*./doc/classdoc/classref/class_dt_vrf_object_plan_manager.html*)
- component-manager: *DtSimComponentManager* (*./doc/classdoc/classref/class_dt_sim_component_manager.html*).

At any given time, the Object Manager may maintain a mix of local and remote objects.

The Component Manager, Task Manager, and Plan Manager are created only for VR-Forces objects.

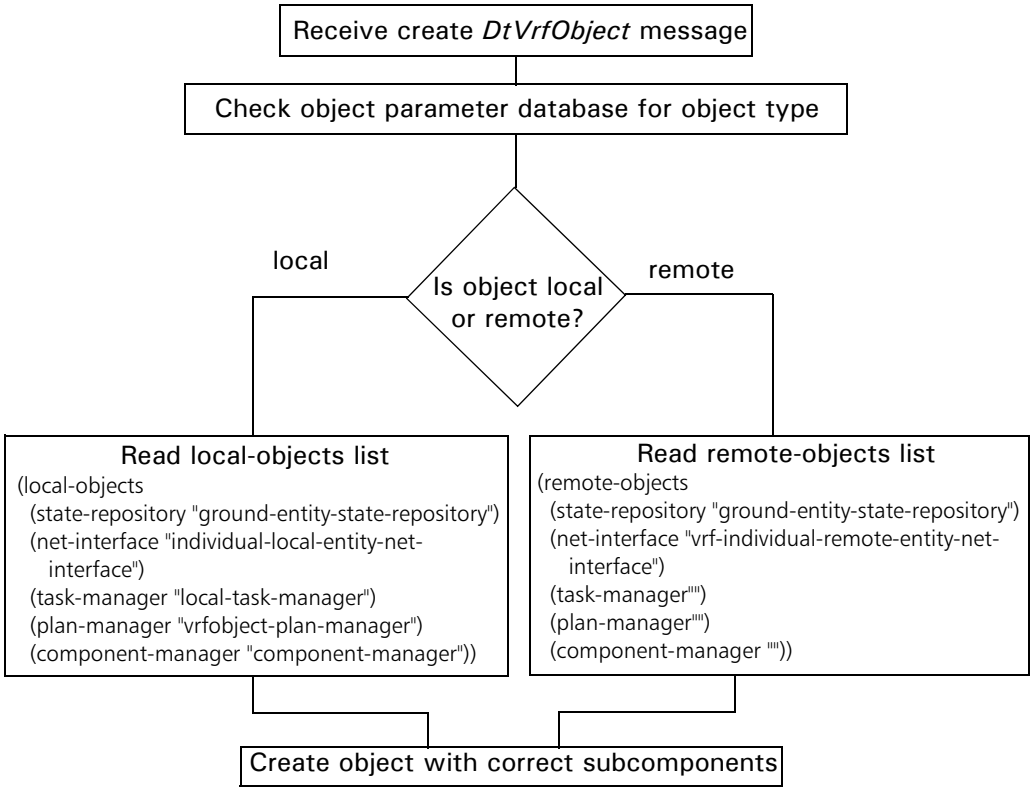


Figure 3-2. Creating an object

3.2.2. Identifying Objects

A *DtVrfObject* can be uniquely identified through one of the following:

- Object Name: A unique string given to an object when it is created. Users can change the name through the GUI or programmatically. For details, please see "The Object Name," on page 3-11.
- Echelon ID: A unique string that represents an entity's place in the organizational hierarchy. It changes if an entity is reorganized into or out of an aggregate. Echelon IDs are managed by the Organization Manager, *DtOrganizationManager*.
- Object Identifier: The site:application:object number triplet used in DIS and the HLA/RPR FOM to identify an object in inter-application communication. It is automatically assigned by the *DtVrfObjectManager*, and persists for the duration of the simulation. Object identifiers are implemented using the *DtEntityIdentifier* class. In DIS, this identifier is equivalent to the global object designator.
- Global Object Designator: A *DtGlobalObjectDesignator* (or global ID) can be used regardless of protocol to identify an entity in inter-application communication within PDUs, interactions, and attribute updates. In HLA, *DtGlobalObjectDesignator* is typedef'd to *DtString* – a VR-Link wrapper around char* that can be used to store object names. In DIS, *DtGlobalObjectDesignator* is typedef'd to *DtEntityIdentifier*, since in DIS that same type of ID is used to identify an object both within an application, and between applications.

3.2.3. Looking Up Objects

You can look up objects in the Object Manager by object ID (site:application_number:object_number), echelon ID, or name. Object names must be unique. The name of an object simulated by an application other than VR-Forces is its marking text.

3.2.4. The Object Type

The object type (*DtObjectType*, in *objType.h*) is an extension of the VR-Link *DtEntityType* class. It adds an enumerated field to the DIS/RPR FOM entity type enumeration. The new field indicates the general type of object, for example, individual or aggregate[1]. The remaining fields (kind, domain, country, category, subcategory, specific, and extra) are unchanged. For example, the entity type enumeration for an M1A2 tank is 1 1 225 1 1 3 -1. The VR-Forces object type is 1 (1 1 225 1 1 3 -1).

The object type is used to:

- Find the appropriate entry for a newly created object in the object parameter database
- Determine (along with force) the icon that should be displayed for an entity
- Create the right kind of HLA object (done by VR-Link)
- Look up information such as target priority or the ammunition to select for a given target.

In the object parameter database, objects can have wildcard values in their object type, for example, 1(1 1 225 -1 -1 -1 -1). However, when you include an object type in code, you must use zeros for unkown or generic values. The equivalent object type in code for this example would be (1 1 1 225 0 0 0 0).

Predefined Object Types

The *DtObjectType* class has static predefined object types. These types can provide a convenient short-hand for specifying various kinds of objects. The predefined static types are:

- Null (0's)
- Any (wildcards)
- Individual objects
- Aggregates
- Pseudo-aggregates
- Individual lifeforms
- Individual platforms
- Cultural features
- Munitions.

The VR-Forces extensions to the standard DIS/RPR FOM entity type enumerations are specified in *objTypeEnums.h*.

1. The multi-resolution aggregates in VR-Forces have a general object type of pseudo-aggregate.

3.2.5. The Object Name

All simulated objects have an object name, a unique string used to identify the object within the exercise. Object names are implemented using the *DtString* (*rwString.h*) class. An object name must be unique across all VR-Forces applications participating in an exercise. An object's name persists not only for the lifetime of the simulation, but from session to session. (It is saved as part of the object's data in the order of battle file.) All references to control objects or entities in entity plans are made using object names.

You may have the *DtVrfObjectManager* suggest a unique name to assign to the object, or you may choose one yourself. Call DtVrfObjectManager::nextName() to have VR-Forces generate a name for a *DtObjectType* and *DtForceType*. It is up to the user to ensure that the name is unique.

Objects can have arbitrary names; however, the names have restrictions on their length, because they are transmitted over the network in the marking text field. Table 3-1 lists the limits on object name length.

Table 3-1: Object name length restrictions

Protocol	Individual Entity	Aggregate	Control Object
DIS	11	32	255
HLA RPR FOM	11	10	255

3.2.6. The Echelon ID

Entities that belong to a hierarchical organization have an echelon ID, a unique string that reflects their position in the hierarchy (for example, `1 M1A2, 2 Plt, 1 Force`). Please see "The Organization Manager," on page 4-11, for more information about echelon IDs.

3.2.7. The Object Label

A label is a text string that can persist beyond a particular run of an exercise (unlike an object ID, which can change between sessions if the site:application_number changes.) It does not have to be unique. It would typically be something that makes sense to a human exercise participant, such as labeling a waypoint "Waypoint Alpha".

3.3. State Repositories

All *DtVrfObjects* have a state repository data member that contains all the state information for the object. State repositories are essentially container classes for state information. If the object is locally simulated, this is the information that is being generated by the simulation. If an object is a remote object, this is information that has been copied (and converted, as necessary) from the object's remote network interface. The repository holds, among other things, an object's position, velocity, acceleration, and orientation in local (database) coordinates, and an indication of whether or not the object is destroyed.

Do not confuse VR-Forces state repositories with the state repositories provided for VR-Link *DtObjectPublishers* and *DtReflectedObjects*. In the context of VR-Forces, the VR-Link state repositories are referred to as network state repositories. (For details, please see "The Network Interface," on page 3-24.)

The state repository (*DtVrfObjectStateRepository*, in *vrfObjSR.h*) contains most of the information found in the network interface, but in a form more suitable for use within a simulation. For example, locations and velocities are kept in local (database) coordinates (as contrasted to geocentric coordinates, which are used in network messages.)

All state repositories have at least the following member data:

- Object type (*DtObjectType*). (For details, please see "The Object Type," on page 3-10)
- Force type
- Object Parameters (*DtVrfObjectParameters*)
  - A pointer to an object parameter database entry
  - A local parameter entry (modifiable)
- Object geometry (*DtVrfObjectGeometry*)
- Kinematic state (*DtVrfKinematicState*)
- An Attachment Manager (*DtAttachmentManager*)
- A Subordinate Manager (*DtVrfSubordinateManager*)
- A Process State Repository Manager (*DtVrfProcessStateRepositoryManager*).

There are several derived state repository classes that contain state information specific to a particular category of objects. For example, the fixed-wing state repository class, *DtFixedWingStateRepository*, maintains information specific to fixed-wing entities, like maximum angle of attack and minimum altitude.

The state repository is a readable, writeable member of a *DtVrfObject* and, therefore, is saved and restored as part of an object's data in the order of battle file.

3.3.1. Checkpointing an Entity's State

Whenever you save or checkpoint a scenario, VR-Forces saves the state of all locally simulated entities to the order of battle file. This state includes items such as kinematic state (position, velocity, orientation, and so on), damage status, and resource levels. VR-Forces also saves information about the processes the entity was carrying out at the time of the save, such as the name of a waypoint it was heading toward, or the identity of the current target in a weapon system.

3.3.2. Object Parameters

Each Object Manager maintains an object parameter database. The object parameter database is a set of text files that store parameter and component data for each type of object that VR-Forces can simulate. When you load a scenario, the Object Manager reads and stores the object parameter database. When an object is created, the Object Manager looks up the object type in the object parameter database and reads the components and parameter values specified for it. It creates the object using those components and parameters.

The contents of the object parameter database file and the meaning of each parameter are discussed in *VR-Forces Configuration Guide*. VR-Forces has an API that you can use to create object parameter databases in file formats of your own choosing. For details, please see "The Object Parameter Database API," on page 3-42.

All objects are associated with a parameters object, a *DtVrfObjectParameters* (*vrfObjParams.h*) or derived class. Object parameters hold all of the information needed to construct a *DtVrfObject* with the proper type of subcomponents. The object parameter database is consulted when an entity is created. The parameters are looked up, then a copy of those parameters is kept with the object.

DtVrfObjects have two object parameters members. One is a pointer to the object parameter database entry the object is created with, and one is its own copy that it creates. The local copy is initialized with values from the object parameter database entry.

You should not modify the contents of the object parameter database entry (returned by parametersEntry()). The local object parameters member (returned by the parameters() accessor) is provided for you to modify as desired for a particular object. If you change the parameters entry for an object, it could affect the construction of future objects.

To access an object's object parameter database entry, do the following:

```
DtVrfObject* vrfObject;
DtVrfObjectStateRepository* stateRep = vrfObject->vrfState();
DtVrfObjectParameters* paramEntry = stateRep->parameterEntry();
```

To access an object's local parameter copy, do the following:

```
DtVrfObjectParameters* params = stateRep->parameters();
```

For more information about parameters, please see "Entity Parameters," on page 4-17.

The object parameter database lists the component strings associated with each object type. When the Object Manager creates an object, it reads the entry for that object in the object parameter database. (For details about the Object Manager, please see "The Object Manager and Simulated Objects," on page 3-3.) It uses the appropriate string in the creator function and the object factory does the rest.

For example, the following code is the beginning of the actuators entry for a generic ground vehicle in the object parameter database. It tells the Object Manager that the component type for the powertrain is an automotive-actuator. The Object Manager looks up the string automotive-actuator in the factory and determines that it has to create a *DtAutomotiveActuator* for the ground vehicle.

```
(actuators
  (powertrain
    (component-descriptor-type "automotive-component-descriptor")
    (component-type  "automotive-actuator")
  )
```

For more information about factories, please see "VR-Forces Factories," on page 2-10. Also, please see the addActuator example in Chapter 6, *Creating Actuators and Controllers*.

3.3.3. Object Geometry

A state repository has an object geometry member, *DtVrfObjectGeometry* (*vrfObjGeom.h*) (derived from *DtSimObjectGeometry*), which contains any bounding geometry and actual model geometry, or both, available for a simulation object. The object geometry represents the extent in space of a given simulated object. It maintains a rectangular bounding volume and an ordered list of vertices describing the physical extent of the object. Control objects may have one or more vertices representing the object. Individual entities are represented by a single vertex and a rectangular bounding volume. The file *kinTools.h* contains utility functions for determining such things as the elevation angle to a given point, distance between points, and whether or not a given point lies inside the bounding volume.

Objects such as platform entities usually do not provide any model data, but are described by a rectangular bounding volume. For an area control object or a route, the vertices of the object are provided. (For details, please see *DtBoundingGeometry*.) The list of vertices may either be closed (as in an area), or open (as in a route). The vertices may also be projected (as in a phase line). If the vertices are projected, then in topocentric coordinates, any computations performed on the vertices ignore the Z component. If you know that your object has projected coordinates, then setting the object geometry projected flag (DtVrfObjectGeometry::setProjection(true)) enables some of the geometry functions to be optimized for 2-D.

Object geometry provides an interface for working in either local (database) coordinates or world coordinates. It is initialized for an object from its object parameter database entry. For example, to iterate over the vertices in an object, do the following:

```
DtVrfObject* vrfObject;
DtVrfObjectStateRepository* stateRep = vrfObject->vrfState();
DtVrfObjectGeometry = stateRep-> vrfObjectGeometry();
// Get an iterator for traversing the object's list of local vertices
DtLocalVectorIterator iterator =
  vrfObjectGeometry()->localVertexIterator();
DtLocalVector* vertex = NULL;
for (vertex = iterator.first();vertex; vertex = iterator.next())
{
  DtInfo("Vertex %s\n", vertex.string().string());
}
```

VR-Forces includes convenience functions associated with object geometry that can be useful when performing modeling, such as collision avoidance and targeting. Please see *kinTools.h*.

Object geometry and kinematic state are closely related. If you change an object's position through its kinematic state, the vertices in its object geometry get updated, and vice versa.

3.3.4. Kinematic State

DtVrfKinematicState (*vrfKinState.h*) encapsulates the kinematic state of an object with respect to the local (database) and network (geocentric) coordinate systems, as well as with respect to its own parent object, for attached objects such as launchers or turrets. Through an object's kinematic state you can query or update an object's position, velocity, acceleration, and orientation. Conversions between the various coordinate systems are handled transparently within the kinematic state object. You can work with whatever coordinate system is most convenient, without needing to write any code to handle conversion.

For example, you can ask for an object's position in the following ways:

```
DtVrfObject* object;
DtVrfObjectStateRepository* stateRep = vrfObject->vrfState();
DtVrfKinematicState* kinematicState = stateRep->vrfKinematicState();
DtVector localPos = kinematicState->localPosition();
DtVector parentPos = kinematicState->parentPosition();
DtVector worldPos = kinematicState->worldPosition();
DtInfo("Object: %s local position : %s parent position :
  %s world position :%s\n",
  object->objectName().string(),
  localPos.string().string(),
  parentPos.string().string(),
  worldPos.string().string());
```

In the case of parentLocation(), if the object is not attached to another object, the object is considered "attached" to the database. (For details about attachment relationships, please see "The Attachment Manager," on page 3-17.) Its parent location is equivalent to its database location. In the case of a turret attached to a tank, however, the parent location for the turret would be given in the body coordinate system of the tank.

Object geometry and kinematic state are closely related. If you change an object's position through its kinematic state, the vertices in its object geometry get updated, and vice versa. (For details, please see "Object Geometry," on page 3-15.)

3.3.5. The Attachment Manager

The Attachment Manager (*DtVrfAttachmentManager*, in *vrfAttachMgr.h*) maintains a tree of physically attached sub-parts for the state repository class. The sub-parts may represent items such as a gun or turret attached to a tank, or they may represent entire objects, such as a tank mounted on an LCAC. These parts are represented as other *DtVrfObjectStateRepositories*. Parts that need to maintain geometry with respect to some parent geometry use this manager. Since some parts may be *DtVrfObjects* in their own right, a reference state repository class (*refSR.h*) is used to represent these parts. Creation of the part state repositories is done outside the Attachment Manager; parts are attached and detached as required. Once a part has been attached with the attachPart() call, the *DtVrfAttachmentManager* assumes responsibility for deleting the memory associated with the part. Creation of articulated and attached parts normally occurs when an object's state repository is being initialized with data from its parameter entry. The parameter entry contains the list of articulated part types, attached part types (if any exist), or both.

The Attachment Manager provides iterators to traverse either the top level state repositories it maintains or the entire forest it represents.

The *DtObjectType* (*objType.h*) object (which is an extension of the DIS enumerated type) is used to describe the part type. This entity type has a Kind of *DtSimArtPartKind*, with a category that matches the *DtArtPartType* used to represent the part on the network. Parts not described by reference state repositories are expected to have unique part types (nothing will break if they do not, findPartByType() finds the first part that matches). For articulated parts, the DIS standard provides different articulated part type values for parts of the same logical type (for example, two turrets could have part types *DtPrimaryTurret1* and *DtPrimaryTurret2*) so this unique part type restriction should be easy to maintain.

For example, to look up the main gun on a tank, do the following:

```
DtVrfObject* tank;
DtObjectType turretType(DtObjectTypeOther, DtSimArtPartKind, 0, 0,
  myTurretToControl, 0, 0, 0);
DtVrfObjectStateRepository * turret =
  (DtVrfObjectStateRepository *) tank->vrfState()->attachments()->
  findPartByType(turretType);
```

For attached objects, it may often be the case that the entity types are not unique. For this reason, findPartByType() ignores reference repositories when searching the elements contained in the Attachment Manager.

Attaching Articulated and Attached Parts

To attach and detach parts (state repositories) use the state repository's attachToParent() and detachFromParent() member functions. These functions invoke the Attachment Manager's attachPart() and detachPart() member functions, respectively.

3.3.6. The Subordinate Manager

Objects such as aggregate entities, may contain a collection of other objects. The *DtVrfSubordinateManager* (*vrfSubordMgr.h*) defines an interface for maintaining a collection of these subordinate objects. Subordinates differ from attachments in that they imply an organizational relationship, as opposed to a physical relationship. The Subordinate Manager provides an interface for adding, removing, and iterating through a list of subordinate objects. For example, to examine the list of subordinates belonging to an object, do the following:

```
DtVrfObject* overlay;
DtVrfObjectStateRepository* stateRep = overlay->vrfState();
DtKeyedVrfObjectIterator iterator = stateRep->vrfSubordinates()->
  vrfIterator();
DtVrfObject* subordinate;
for (subordinate = (DtVrfObject*)iterator.first(); entity;
  subordinate = (DtVrfObject*)iterator.next())
{
  DtInfo("Subordinate name : %s object identifier : %s\n"<
    subordinate->objectName().string(), subordinate->
    objectIdentifier().string());
}
```

Entities that exist as part of a military hierarchy have controller components that manipulate the Subordinate Manager directly. In general, you should not have to write any code to manipulate the Subordinate Manager.

For more information, please see "The Organization Manager," on page 4-11.

3.3.7. The State Repository Hierarchy

As noted in "Simulated Objects," on page 3-4, the differences between simulated objects are expressed in the state repository class hierarchy. The differences between a tank, a helicopter, or a route are largely a function of the kind of state repository each has. Control objects use a *DtVrfObjectStateRepository*. Entities use *DtMovingObject-StateRepository* or one of its derived classes. Figure 3-3 illustrates the hierarchy of derived state repositories for simulating different kinds of objects.

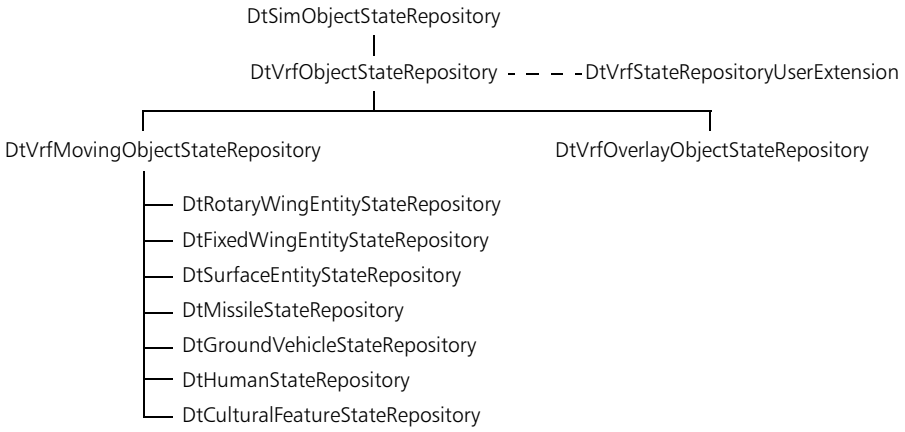


Figure 3-3. State repository hierarchy

DtVrfObjectStateRepository

The *DtVrfObjectStateRepository* (*vrfObjSR.h*) holds information that is common to all simulated entities within VR-Forces. It is an intermediate state repository that has a role similar to an abstract base class for entity state repositories. It is not created for any entities in VR-Forces. However, all other state repositories in VR-Forces inherit this class.

The DtVrfObjectStateRepository maintains the independently tasked and request task needed state for an entity. These members are typically accessed through the *DtLocal-TaskManager* class (users should not normally need to access them directly). They exist as readable and writeable members of the DtVrfObjectStateRepository so they get saved and restored as part of an entity's state in the order of battle file. Similarly, the DtVrfObjectStateRepository maintains a readable and writeable list of current tasks an entity is working on. This data is manipulated by the *DtTaskControllerComponents* configured on an entity (users should not typically have to access it directly). This data is also saved as part of an entity's state in the order of battle file. This enables entities to remember what task they were working on at the time the scenario was saved (whether assigned from a plan or from the GUI as an independent task).

DtMovingObjectStateRepository

DtVrfMovingObjectStateRepository (*vrfMvObjSR.h*) is derived from *DtVrfObjectStateRepository*. It holds information that is common to all VR-Forces objects that can move. It is the base class for all of the individual entity state repositories. It is also the state repository used for aggregate entities. It provides storage, accessors, and mutators for the following data:

- Ordered speed (the speed the entity is supposed to use when moving)
- Maximum speed
- Maximum reverse speed
- Turning radius
- Maximum slope
- Inverse stopping deceleration
- Maximum acceleration
- Maximum stopping deceleration
- Fuel efficiency
- Direction of movement (forward() function).

3.3.8. Extending State Repositories at the Base Class Level

You can extend the base state repository class *DtVrfObjectStateRepository*, as an alternative to, or in addition to extending leaf-level classes in the *DtVrfObjectStateRepository* hierarchy. The base state repository class, *DtVrfObjectStateRepository* has-a *DtVrfStateRepositoryUserExtension* (*vrfSRUserExt.h*) that you can extend for all types of entities and objects (Figure 3-3).

For example, suppose you want to extend all types of entities and objects in VR-Forces (fixed-wing, rotary-wing, ground, and so on) with a new kind of identifier. Rather than extending each type of state repository to include the new identifier, you just need to create a single derived kind of *DtVrfStateRepositoryUserExtension* class. This allows you to write the extension once, and configure it as part of a variety of different kinds of entities, control objects, or overlay objects. The alternative to this approach would be to subclass each kind of state repository separately, extending each class in exactly the same way.

Like the *DtVrfObjectStateRepository* classes, classes derived from *DtVrfStateRepositoryUserExtension* are readable and writeable. Any *DtReaderWriter* data members you add to your derived *DtVrfStateRepositoryUserExtension* will be saved to an order of battle file along with the rest of an object's *DtVrfObjectStateRepository* data.

The example project extendStateRepository demonstrates how to extend state repositories at the base class level. The following procedure contains code excerpts from this example.

To extend state repositories at the base level:

1. Create your derived kind of *DtVrfStateRepositoryUserExtension* class. In your derived class, you must provide a copy constructor and a clone() member function.

```
#include <vrfSRUserExt.h>
// New state repository extension, with a single new
// data member, myTag (a new kind of identifier).

class MyDerivedSRExtension : public DtVrfStateRepositoryUserExtension
{
  . . .
protected:

  DtRwString myTag;
}
```

2. Register any readable/writeable data member with the parent registry in the constructor.

```
MyDerivedSRExtension:: MyDerivedSRExtension (
  DtVrfObjectStateRepository& stateRepository,
  const DtString& rwName,
  DtReaderWriterRegistry* parentRegistry) :
  DtVrfStateRepositoryUserExtension(stateRepository, rwName,
    parentRegistry),
  myTag("my-tag", &myRegistry)
{
  myTag = "some-user-info";
}
```

3. Provide a type() member that returns a unique string. It is used to register the class with the factory.

```
DtString MyDerivedSRExtension::type() const
{
  return “my-state-repository-user-extension”;
}
```

4. Provide a static create function. This will be registered with the factory for this type of class.

```
DtVrfStateRepositoryUserExtension* MyDerivedSRExtension::create(
  DtVrfObjectStateRepository& stateRepository,
  const DtString& rwName,
  DtReaderWriterRegistry*  parentRegistry)
{
  return new MyDerivedStateRepositoryExtension(stateRepository, rwName,
    parentRegistry);
}
```

5. In main, register the creator with the factory that creates this type of object. When VR-Forces encounters this type in a object parameter database entry, it calls your class's create function to create an instance of that type.

```
int main(int argc, char **argv)
{
  DtVrfApp app(__argc, __argv);
  DtVrfApp app(argc, argv);

  app.init();

  // Registers create function for new kind of user extension
  // with the factory. This is the string returned by your
  // class's type() function.

  app.cgf()->
    factoryManager()->
    stateRepositoryUserExtensionFactory()->addCreatorFcn(
    MyDerivedSRExtensionType, MyDerivedSRExtension::create);

  app.mainLoop();
```

6. Configure the appropriate object parameter database files with the new type of extension. Set the value of the state-repository-extension-type variable to the string returned by your new class's type() function.

```
;; Configure M1A2 parameter entry with new state repository
;; extension (file M1A2_1_1_1_225_1_1_3_-1.1.23.ope).
;;
(M1A2
  (parameter-type "ground-vehicle-param")
  (object-type  1 (1 1 225 1 1 3 -1))
  . . .
  (state-repository-extension-type
    "my-state-repository-user-extension")
```

7. Access the extension as needed in your code. You can obtain a pointer to the variable through the object's state repository. Use dynamic_cast to determine the correct type.

```
#include <vrfObject.h>
#include <vrfObjectSR.h>
#include <vrfSRUserExtension.h>

DtVrfObject* vrfObject;
. . .
MyDerivedSRExtension* userSRExtension =
  dynamic_cast<MyDerivedSRExtension*>(vrfObject->
  vrfState()->userExtension());

if(userSRExtension)
{
  userSRExtension->setTag("some-useful-info");
```

8. Save and restore new data to the order of battle file. All *DtReaderWriter* member data in your user extension is saved and restored as part of the entity state in the order of battle file as part of a scenario. For example, the user extension containing the single data member named 'tag' would look like the following for an M1A2 object saved in a scenario:

```
;; Entry for an M1A2 object saved as part of a scenario in an
;; order of battle (.oob) file.
(local-vrf-object
  (vrf-entity  "vrf-entity")
  (object-type  1 (1 1 225 1 1 3 0))
  (object-name  "M1A2 1")
  . . .
  (user-extension
    (my-tag  "some-user-info")
)
```

Initializing State Repository Extensions in the Object Parameter Database

You can configure initial values for the readable/writeable data members of a derived *DtVrfStateRepositoryUserExtension* by adding a user-extension parameter entry to an entity's OPE file. For example, if you create a derived *DtVrfStateRepositoryUserExtension* of type **my-state-repository-user-extension**, that has a readable/writeable string data member with the *DtReaderWriter* name **my-tag**, you could configure its initial value in the OPE file as follows:

```
(state-repository-extension-type  "my-state-repository-user-extension")
(user-extension
   (my-tag "some-initial-value")
)
```

3.4. The Network Interface

All networked objects have an instance of a *DtSimObjectNetInterface* (*netIface.h*). *DtSimObjectNetInterface* is an abstract base class from which local and remote network interfaces are derived. The network interface represents the interface between an object's VR-Forces state repository and its network (VR-Link) representation. The simulated object may be local or remote. If the object is local, the network interface is responsible for updating a VR-Link publisher with data from its VR-Forces state repository each tick. If the object is remote, the network interface updates a VR-Forces state repository with data from its VR-Link reflected object. Figure 3-4 illustrates local and remote network interfaces within a VR-Forces application.

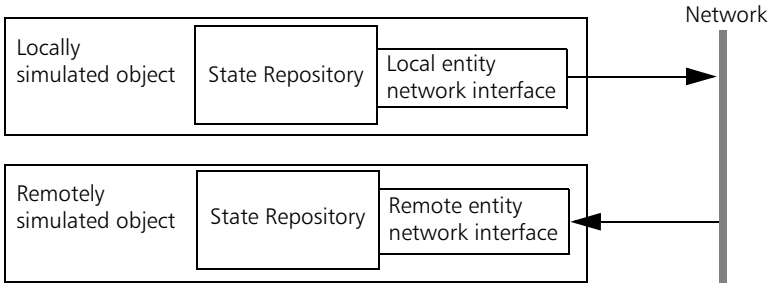


Figure 3-4. Network interfaces

Local network interfaces send data over the network through VR-Link publishers. Depending on the kind of object being simulated, it may use a *DtEntityPublisher*, *DtAggregatePublisher*, or *DtEnvironmentalProcessPublisher*. Remote network interfaces maintain one of the following VR-Link reflected objects: a *DtReflectedEntity*, *DtReflectedAggregate*, or a *DtReflectedEnviromentProcess*.

Figure 3-5 illustrates use of local and remote network interfaces in two VR-Forces applications. If back-end 1:3002 is simulating a tank, the *DtVrfObject* representing the tank has a local network interface. In back-end 2:3001, a *DtVrfObject* with a remote network interface is created to represent the tank. In both applications, the *DtVrfObject* representing the tank has a state repository. On back-end 1:3002, the local network interface for the *DtVrfObject* sends state information over the network. On back-end 2:3001, the remote network interface for the *DtVrfObject* receives state information from the network.

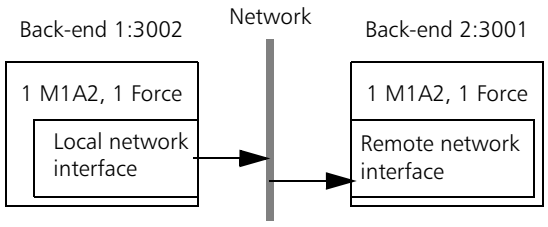


Figure 3-5. Local and remote network interfaces

3.4.1. Local Network Interfaces

A local network interface contains a VR-Link publisher and an associated VR-Link network state repository (not to be confused with the simulation state repository). Each tick, an object's local network interface takes state information from the object's state repository and copies it to its network state repository. The VR-Link publisher is responsible for sending updates, as necessary, in the form of DIS PDUs or RPR FOM object attribute updates. It also sends VR-Forces-specific interface messages (*DtVrfObjectData*) to supplement the standard DIS/RPR FOM data with application-specific information. This additional data includes the object name and object label.

3.4.2. Remote Network Interfaces

A remote network interface contains a VR-Link reflected object and its associated VR-Link network state repository (not to be confused with the simulation state repository, owned by a *DtVrfObject*). Each tick, the network interface takes state information maintained in the VR-Link network state repository and updates the object's simulation state repository. Receipt of standard DIS/RPR FOM data is handled at the VR-Link level in the network interface's reflected object member data. Remote network interfaces also register for *DtVrfObjectData* interface messages, and update the object's simulation state with VR-Forces application-specific data.

3.4.3. Configuring an Object with a Network Interface

You configure the network interface for a type of object by editing its entry in the object parameter database. When you configure an object's network interface, the string you enter determines the class that gets created when the object is instantiated, as listed in Table 3-2. Network interface type strings are defined in *vrfNetIfTypes.h*.

Table 3-2: Network interface classes

String	Class
Individual platform-level entity	
individual-local-entity-net-interface	*DtSimIndividualLocalEntityNetInterface*
vrf-individual-remote-entity-net-interface	*DtVrfIndividualRemoteEntityNetInterface*
Pseudo-aggregate entity	
pseudo-aggregate-local-entity-net-interface	*DtSimPseudoAggregateLocalEntityNetInterface*
vrf-aggregate-remote-entity-net-interface	*DtVrfAggregateRemoteEntityNetInterface*
Control object	
local-environmental-net-interface	*DtSimLocalEnvironmentalNetInterface*
vrf-remote-environmental-net-interface	*DtVrfRemoteEnvironmentalNetInterface*

Locally simulated aggregates use a network interface of type pseudo-aggregate-local-entity-net-interface.

3.4.4. Tuning the Network Interface

You can tune the local and remote network interface subcomponents. Specify the **net-interface-min-tick-period** and **net-interface-min-tick-period-variance** parameters in the local and remote subcomponents blocks. If you reduce the rate at which the network interfaces are ticked, you can limit how often VR-Forces needs to do coordinate system conversion from network (geocentric) to database coordinates. The object parameter database has examples of setting these parameters.

3.5. Creating and Managing Objects

The Object Manager creates and maintains local objects. It also instantiates local instances of remotely simulated objects. It monitors the network for state updates from remote objects in an exercise and for VR-Forces-specific state information. When it detects the presence of a new remote object, it creates a new remote *DtVrfObject* to represent it and adds it to list of objects to be managed.

3.5.1. Object Factories

When VR-Forces reads object specifications from a file, or when it receives them in an interface message, it needs to instantiate those objects in a simulation. To allow you to add new types of objects without having to modify the file-reading code, VR-Forces uses object factories. A factory is a list of creator functions indexed by a key (which can be a string, an integer, or both) that indicates the kind of object to create. The following types of objects have factories that use this basic scheme:

- Parameters
- Tasks
- Components
- Component descriptors
- Set data requests
- Conditional expressions
- Plan statements
- State repositories
- Process state repositories.

The factories are also used to create entities and control objects that are specified in interface messages from the GUI.

The factories are created by the *DtVrfCreator*. For more information about how to add new types of objects to VR-Forces using object factories, please see "Customizing or Extending the Simulation Engine," on page 2-8.

3.5.2. Creating a New Local Object

Locally simulated *DtVrfObjects* in VR-Forces should be created through the *DtVrfObjectManager*. They are created by calling one of the Object Manager's createAndInitVrfObject() functions. These functions create, initialize, and add a new instance of a *DtVrfObject* to the Object Manager.

There are two overloaded versions of the createAndInitVrfObject() function, for use with different kinds of object geometry. The first form takes a list of vertices as an argument. This is the set of vertices in local coordinates that make up the geometry of the object. This form is suitable for creating control objects, such as routes and waypoints. For example, to create a route named Route 1, consisting of three vertices, do the following:

```
DtVrfObject * route;

DtList vertices;

// Add some vertices to our route. Expressed in geocentic coordinates.
vertices.add(new DtVector(-2812729.7730, -4332999.1841, 3728433.175));
vertices.add(new DtVector(-2812375.6097, -4332915.4285, 3728795.2170));
vertices.add(new DtVector(-2812101.6319, -4332990.6562, 3728913.6333));

// Route object type.
// Extensions to entity type enumerations are in objTypeEnums.h
DtObjectType routeType(1, 17, 0, 0, 2, 0, 0, 0);

route = objectManager->createAndInitVrfObject(
  routeType,
  true,
  "Route 1",
  DtString::nullString(),
  DtForceOther,
  vertices);
```

The second form of the createAndInitVrfObject() call takes a single position as an argument. This form is suitable for creating entities or other objects with simple geometry such as waypoints. The object's geometry gets initialized with a single vertex given by the position. For example, to create a waypoint named Waypoint Alpha, do the following:

```
DtVrfObject * waypoint;

// Waypoint object type.
// Extensions to entity type enumerations are in objTypeEnums.h
DtObjectType waypointType(1, 16, 0, 0, 1, 0, 0, 0);

// Initial position of waypoint, in geocentric coordinates.
DtVector initialPosition(-2812375.6097, -4332915.4285, 3728795.2170);
waypoint = objectManager->createAndInitVrfObject(
  waypointType,
  true,
  "Route 1",
  DtString::nullString(),
  DtForceOther,
  initialPosition);
```

If the new local object has been configured with a local network interface, then the object is published over the network. Any remote VR-Forces application's Object Manager will detect the existence of the new *DtVrfObject* and automatically create a remote instance of it.

! Do not call the *DtVrfObject* constructor directly. Do all of the creation of *DtVrfObjects* through the Object Manager (through its createAndInitVrfObject()) functions. The Object Manager not only instantiates the object, but initializes and adds the object to itself for management.

3.5.3. Removing a Locally Simulated Object from the Simulation

To destroy a simulated object, call one of the DtVrfObjectManager::removeAndDelete() functions. One form of the this function takes a pointer to a *DtVrfObject* directly; one form takes a *DtEntityIdentifier* key. In this function, the Object Manager calls queueObjectForRemoval() to place the *DtVrfObject* on a list of objects to be removed. At the end of the Object Manager's tick, it iterates through the list of objects to be removed, removes the object from its list of *DtVrfObjects*, and deletes it. If the *DtVrfObject* is locally simulated, then it also sends out a *DtVrfObjectDeleted* interface message to notify remote VR-Forces applications that the object should be deleted. Remote VR-Forces applications receive the interface message and delete the object in their Object Managers.

! Do not delete a *DtVrfObject* directly. Use DtVrfObjectManager::removeAndDelete() to ensure that the *DtVrfObject* is removed at a safe point during the simulation frame.

3.5.4. Getting Notified When Objects are Added or Removed

You can register callbacks with the Object Manager to receive notification for the following events:

- Object about to be added
- Object added
- Object about to be removed.

The callbacks are registered with the *DtVrfObjectManager* using the following functions:

- addVrfObjectAboutToBeAddedCallback()
- addVrfObjectAddedCallback()
- addVrfObjectAboutToBeRemovedCallback().

"Object about to be added" callbacks are invoked immediately after a new *DtVrfObject* has been created and initialized, but just before it gets added to the Object Manager's list. "Object added" callbacks get invoked immediately after a newly created and initialized *DtVrfObject* is added to the Object Manager's list. "Object about to be removed" callbacks get invoked just prior to removal of an object from the Object Manager's list.

Callback functions must have a signature of the following form:

```
typedef void (*DtVrfObjectListActivityCallbackFcn)
  (DtVrfObject * vrfObject, void * usr);
```

The following example demonstrates how to register the callbacks described in this section. The application (for example, a simulation back-end) displays diagnostic print statements as objects are added to, and removed from, the Object Manager.

```
static void addingObjectCb(DtVrfObject * vrfObject, void * usr)
{
   DtInfo("About to add new object % \n", vrfObject->objectName());
}

static void addedObjectCb(DtVrfObject * vrfObject, void * usr)
{
   DtInfo("Just added new object %s\n", vrfObject->objectName());
}

static void removingObjectCb(DtVrfObject * vrfObject, void * usr)
{
   DtInfo("About to delete object %s \n", vrfObject->objectName());
}

main()
{
. . .
   objManager->addVrfObjectAboutToBeAddedCallback(addingObjectCb);
   objManager->addVrfObjectAddedCallback(addedObjectCb);
   objManager->addVrfObjectAboutToBeRemovedCallback(removingObjectCb);
. . .
}
```

Remember to unregister your callbacks when you no longer need them. Each of the above add*Callback() functions has a corresponding remove*Callback() function you can call to remove it from the Object Manager.

3.5.5. Looking Up Individual Objects

You can look up an individual object in the Object Manager using one of the following keys:

- Object name
- Echelon ID
- Object identifier
- Global object designator.

Lookup functions return a pointer to the object if it is found, NULL if it is not found. For example, to look up an object named Waypoint Alpha, do the following:

```
DtVrfObjectManager* objectManager;
DtVrfObject * object;
. . .
object = objectManager->lookupVrfObjectByName("Waypoint Alpha");
```

3.5.6. Iterating Through Simulation Objects

The Object Manager provides two different member functions for iterating through the objects in a simulation. An iterator of type *DtVrfObjectFilterIterator* is returned by the Object Manager's vrfIterator() member function. The following example shows how to iterate over all the objects in the simulation:

```
DtVrfObject* pVrfObject = NULL;

DtVrfObjectFilterIterator iterator = pObjectManager->vrfIterator();

for (pVrfObject = iterator.first(); pVrfObject; pVrfObject =
  iterator.next())
{
  . . .
}
```

Including and Excluding Objects

You may want to include or exclude certain types of objects. For example, Example 3.1 causes all air vehicles to be excluded.

Example 3.1

```
// Note: -1s are wild cards for DtObjectType::matchPattern().
DtObjectType airObjectFilter(DtObjectTypeIndividual, DtPlatform,
  DtPlatformDomainAir,  -1, -1, -1, -1, -1);

DtVrfObject* pVrfObject = NULL;

DtVrfObjectFilterIterator iterator = pObjectManager->vrfIterator();

for (pVrfObject = iterator.first(); pVrfObject; pVrfObject =
  iterator.next())
{
  if (pVrfObject->objectType().matchPattern(airObjectFilter))
  {
    continue;
  . . .
}
```

Using an Object Predicate Filter

You can include and exclude objects by providing a filter when you request the iterator from the Object Manager. The filter is a *DtVrfObjectPredicate*. *DtVrfObjectPredicates* is discussed in detail in "Object Predicates," on page 3-35, but the following example shows a *DtObjectTypeEqualPredicate*, which is a kind of *DtVrfObjectPredicate* being used to accomplish the same thing as Example 3.1. In this case, the air vehicles are skipped by the iterator itself. Note the second argument to the constructor of the predicate. It is a negate flag, indicating that the sense of the predicate should be negated, or inverted.

```
DtObjectTypeEqualPredicate airObjectFilter(
  DtObjectType(DtObjectTypeIndividual, DtPlatform, DtPlatformDomainAir,
  -1, -1, -1, -1, -1)), true);

DtVrfObject* pVrfObject = NULL;

DtVrfObjectFilterIterator iterator = pObjectManager->
  vrfIterator(airObjectFilter);

for (pVrfObject = iterator.first(); pVrfObject; pVrfObject =
  iterator.next())
{
  . . .
}
```

VR-Forces provides a few commonly used predicates, including *DtObjectTypeEqualPredicate*, *DtDestroyedPredicate*, and *DtFilterFunctionPredicate*, which allow you to specify an arbitrary filter function. You can, of course, create your own derived predicates. Please see "Object Predicates," on page 3-35 for more information.

Early versions of VR-Forces allowed an object type and negate flag to be passed as arguments to the vrfIterator() member function instead of a *DtVrfObjectPredicate*. This form is still supported for backward compatibility, but use of the *DtVrfObjectPredicate* is the preferred method.

Using Managed Object Lists

While iterators are fairly easy to use, in some cases they may be quite inefficient. For example, if multiple entities had components that needed to iterate through all the non-air domain entities, then each would be comparing every entity's object type to the filter every simulation frame, even though entity object types never change, and even though each component ends up with the same set of entities. It would be somewhat more efficient if each component needing all non-air domain entities registered callbacks for entity additions and removals, and then built and maintained its own list of non-air entities. When a new entity entered the simulation, its object type could be checked once, and it would then be placed on the list or not, based on the filter criteria. While this is an improvement over iterating through all the entities every frame, the use of memory is still inefficient, because every component that needs such a list has to keep (and manage) what is, in effect, the same list.

The Object Manager provides the getList() member function to request such a list. The list is automatically built and maintained if it does not already exist at the time of the request. These lists are called Managed Object Lists, and once requested, can be iterated using a *DtManagedObjectList::const_iterator*. You can use the same predicates used for the filtered iterators discussed in "Using an Object Predicate Filter," on page 3-32, to specify a Managed Object List. When a Managed Object List is requested, the Object Manager checks to see if there is already a list with the same predicate being maintained. If there is, a shared pointer to this list is returned; otherwise, the list is created and a shared pointer to the new list is returned. Managed Object Lists are typically requested during initialization of a component and cached for use later in the component's tick. The memory associated with a given Managed Object List is automatically cleaned up when the last reference to the list is destroyed.

The following example shows a component requesting a list of objects that excludes entities in the air domain, maintaining a reference to that list, and iterating over it in its tick() function. It keeps the *DtManagedObjectListPtr* returned by getList() in a member variable called **myList**.

```
// access (and create, if needed, the air vehicle-free list)
DtExampleComponent::init()
{
  DtObjectTypeEqualPredicate airObjectFilter(
    DtObjectType(DtObjectTypeIndividual, DtPlatform,
    DtPlatformDomainAir, -1, -1, -1, -1, -1)), true);
  myList = mySimManager->vrfObjectManager()-> getList(airObjFilter,
    true);
}

// use the air vehicle-free list
DtExampleComponent::tick()
{
  DtVrfObject* pVrfObject;
  DtManagedObjectList::const_iterator iter;
  for (iter = myList.begin(); iter != myList.end(); ++iter)
  {
    pVrfObject = *iter;
    . . .
  }
}
```

You can create filtered lists in which the object attributes being tested by the predicate change, such as building and maintaining a list of all the destroyed objects. The Object Manager's getList() member function has two optional arguments – the first is a flag that you can use to tell the Object Manager that the list should be re-evaluated periodically. By default, this is done every simulation frame. The third optional argument specifies a period, in seconds, for the re-evaluation.

The following example requests a list of all destroyed entities, re-evaluated once a second:

```
// The negate constructor argument could be set to true to get a list of
// healthy objects
DtDestroyedPredicate destroyedPredicate();
myList = mySimManager->vrfObjectManager()->getList(destroyedPredicate,
  true, 1.0);
. . .
```

3.5.7. Object Predicates

Object predicates are used to determine if an object meets some condition or set of conditions. The *DtVrfObjectPredicate* is an abstract class that defines an interface for object predicates. It consists primarily of an eval() function that takes a *DtVrfObject** argument and returns a `bool`. Constructors normally take any data needed (such as a value to compare an object attribute to) and a `bool` negate flag which inverts the sense of the test (and defaults to `false`). VR-Forces provides object predicates for:

- Checking if an object is destroyed (*DtDestroyedPredicate*, in *destroyedPred.h*).
- Checking if an object has an object type that matches a specified pattern (*DtObjectTypeEqualPredicate*, in *objTypeEqPred.h*).
- Checking if an object has a local vertex list that differs from the vertex list contained in the predicate (*DtLclVerticesNotEqualPredicate*, in *lclVerticesNotEqPred.h*).
- Evaluating a specified filter function (*DtFilterFunctionPredicate*, in *fltrFcnPred.h*). *DtFilterFunctionPredicate* uses a *DtFilterFcn*, which is defined in *fltrFcnInf.h* to be:

  ```
  typedef bool (*DtFilterFcn)(const DtVrfObject * object, void * usr);
  ```

  The filter function and optional user data must be supplied with the constructor.
- Combining predicates (*DtCompositePredicate*, in *compositePred.h*). Objects pass evaluation by a composite predicate only if they also pass evaluation by all sub-predicates.
- Checking whether an object's echelon ID has changed since the last time it was evaluated (*DtEchelonIdChangedPredicate*, in *echelonIdChangedPred.h*). This predicate compares an object's echelon ID to the echelon ID in the predicate.
- Tests an object's *DtForceType* for inclusion in a set of *DtForceTypes* (*DtForceTypePredicate*, in *forceTypePred.h*).

To create your own object predicate, you need to override the virtual method:

```
bool eval(DtVrfObject* object)const
```

which performs the desired test, and:

```
int predicateEqual(const DtVrfObjectPredicate& orig const)
```

The predicateEqual() member function is used to determine if an object predicate is the same as one already specified for an existing list. Given a predicate of type **MyPredicate** that contained member data **myData**, an example implementation of predicateEqual() might look like the following:

```
int MyPredicate::predicateEqual(const DtVrfObjectPredicate& orig) const
{
  // Use dynamic_cast to first determine if it is the right kind of
  // predicate
  const MyPredicate * castOrig = dynamic_cast<const MyPredicate *>
    (&orig);

  if (!castOrig)
  {
    return 0;
  }

  // Compare any predicate-specific data
  if (myData != castOrig->myData)
  {
    return 0;
  }

  // Return the result of the base class comparison, since everything
  // else matches
  return DtVrfObjectPredicate::predicateEqual(orig);
}
```

3.5.8. Notifying an Application When a Simulation Object Changes

You can use *DtVrfObjectPredicate* with *DtVrfObject* to register a callback function to be called when the predicate becomes true. For example, a UAV component might register a callback to be notified if an object destroyed predicate becomes true for the entity representing the UAV's controller station. The callback to be registered is defined in *vrfObjPredCbInfo.h* and has the form:

```
typedef void (*DtVrfObjectPredicateCallbackFunction)( DtVrfObject* obj,
   DtVrfObjectPredicateEvaluator* predicateEval, void* userData);
```

For example, in the following code snippet, a callback named objectIsDestroyedCallback, to be called if an object pointed to by pVrfObject is destroyed, is registered with a *DtVrfObject*.

```
DtDestroyedPredicate destroyedPredicate;
pVrfObject->addVrfObjectPredicateCallback(objectIsDestroyedCallback,
  destroyedPredicate, 1.5, this);
```

The third argument, 1.5, indicates that every 1.5 seconds, the object should re-evaluate if it is destroyed. The final argument is a `void* userData` argument that will be supplied to the callback function when it is invoked.

The *DtVrfObjectPredicateEvaluator* is created by the *DtVrfObject* to manage the predicate evaluation. A pointer to it is provided to make it easy to change that predicate, if desired, when the callback is invoked (using the DtVrfObjectPredicateEvaluator::predicate() and DtVrfObjectPredicateEvaluator::setPredicate() member functions), or modify the interval at which the predicate is evaluated (using VrfObjectPredicateEvaluator::setTestInterval()).

3.5.9. Important Coding Recommendations

Note the following recommendations:

- Iterators returned by the Object Manager are not robust iterators. Traversal may not work correctly if objects are inserted or removed during the lifetime of the iterator. Therefore, we recommend that you do not keep or reuse an iterator for more than one simulation frame.
- We recommend that you do not retain pointers to *DtVrfObjects* over multiple ticks. It is safer to retain an identifier of some sort (that is, the object name, object identifier, or global identifier), and use that identifier to look up the object in the Object Manager when you need it.

3.6. Object Operations and Blocking Terrain Calls

There are several circumstances under which terrain data must be available to complete an operation, such as creating objects, completing move to location tasks, and set location data requests. If you are simulating on paged terrain and required terrain has not paged in yet, the simulation may be blocked pending the availability of the required terrain.

Simulation models cannot make blocking terrain calls that page in terrain patches (or any other operation which takes a long time) without having serious consequences on the simulation. Even a single long tick can have very negative effects on a movement model that is integrating over time. In most cases the *DtMovementBasedTerrainPreloadController* should take care of paging in the required terrain before an entity reaches it, but in case that does not happen it is important to design simulation models to deal with the consequences.

To prevent terrain intersection calls from blocking while paging in terrain, all simulation models must make their terrain calls using the **dataAvailable** flag. When specified, this optional parameter prevents the terrain call from blocking while the data is paged in. In addition to specifying this flag on calls into *DtPhysicalWorld* and *DtTerrainInterface*, it must also be specified on any call to any of the place(), placePoints(), and clampToGround() functions on the various state repository classes. In the case of a terrain intersection function, the **dataAvailable** flag will be set to false and no terrain intersections will be returned. In the case of a place() or clampToGround() function, the **dataAvailable** flag will be set to false and the points will be placed as specified without regard to the terrain. Since the clampToGround() and place() functions are actually doing multiple terrain intersections (one for each support point), they also have a **requireAllData** flag. This flag causes the call to abort and to reset the state of the entity to the way it was before the call if any of the support points do not have data. This makes it easier to poll on a place() or clampToGround() call until all the data is ready.

To find blocking calls at runtime, the **assertOnBlockingTerrainCalls** parameter in *./appData/settings/vrfSim/vrfSim.mtl* should be set to 1. This causes any terrain calls that do not specify the **dataAvailable** flag to assert. This only affects back-ends running in debug mode.

Calling the blocking form of any terrain intersection function causes an assertion. It is not necessary for the call to actually block and even non-paging implementations will assert. This is done to make it easier to find potential problems regardless of the particular terrain or location being used. In certain cases, it is safe to turn off the assert by setting the **overrideBlockingAssert** parameter on the terrain intersection call to true. This could include cases where the call is being made outside the main simulation thread, for instance from the path planning thread.

3.6.1. Queueing Object Creation

When VR-Forces creates an object, the terrain at the requested creation location must be available. If the terrain has not yet been paged in, in most cases the creation of the object must be queued so that the rest of the simulation can continue to run while the terrain is being paged in.

The *DtVrfObjectManager* queues DtIfCreateVrfObject messages until the terrain data for the requested creation location is available. Each time a creation request is processed, a terrain intersection is done and the **dataAvailable** flag is checked. If the terrain is available, the request is processed immediately. If not, the object manager adds the request to a queue which is checked each tick.

When creating entities using *DtCgf*, you can specify whether or not the creation should allow the call to block on terrain paging using the **allowBlocking** parameter. In most cases if the simulation has not yet started it is preferable to allow this parameter to default to true as you will get a *DtVrfObject* in return which you can immediately begin to work with. If the simulation is running, **allowBlocking** should always be set to false to prevent terrain paging from negatively affecting the simulation. In this case a NULL pointer is returned because the creation will be queued.

3.6.2. Queueing Set Location Requests

DtLocalEntitySetController and its subclasses queue Set Location requests until the terrain data is available for the requested location. Only one Set Location request is ever queued at a time. If another request comes in while terrain is still paging in from the first, the original request is cancelled.

3.7. Control Objects

All control objects (also called tactical graphics in the VR-Forces front-end) are *DtVrfObjects*. Control objects allow you to identify locations on the terrain map that entities need to interact with in some way. You might want to create control objects to represent things such as minefields, tank ditches, buildings, bridges, and so on. VR-Forces provides the following types of control objects:

- Waypoints
- Phase lines
- Routes
- Areas
- Obstacles.

Control objects and their uses are described fully in *VR-Forces Users Guide*.

3.7.1. Creating Control Objects

Control objects are created using the standard factory system, so you can add your own types. Like entities, they can have parameters, and have entries in the object parameter database. Control objects are readable and writable, and are saved to the order of battle file and object map file when you save a scenario.

All control objects have a:

- State repository of type *DtVrfObjectStateRepository* (in *vrfObjSR.h*)
- Network interface of type *DtSimLocalEnvironmentalNetInterface* (in *lclEnvNetIf.h*).

3.7.2. Control Object Geometry

A control object's geometry is maintained in its state repository's object geometry member, *DtVrfObjectGeometry*. The object geometry maintains the control object's list of vertices and a rectangular bounding volume. For control objects such as a waypoint, this may be a single vertex, while a route or area may contain many vertices.

3.7.3. Control Object Parameters

Each type of *DtVrfObject* has an entry in the object parameter database. Like all other objects, a control object has an object type. Since a control object must be represented in the back-end that is simulating it and communicated to remote VR-Forces applications, each control object has a local and remote state repository, and a local and remote network interface.

A typical entry is as follows:

```
(phase-line-parameters
  (parameter-type "vrf-object-param")
  (object-type  1 (17 -1 -1 1 -1 -1 -1))
  (bounding-volume
    (local-bvol  0.000000 0.000000 0.000000)
    (offset  0.000000 0.000000 0.00000)
   )
  (local-objects
    (state-repository "vrf-overlay-object-state-repository")
    (net-interface "local-environmental-net-interface")
  )
  (remote-objects
    (state-repository "vrf-overlay-object-state-repository")
    (net-interface "vrf-remote-environmental-net-interface")
  )
)
```

A control object has the object super-type 1, which means it is an individual object. The kind value is one of the following:

Table 3-3: Kind values for control objects

Object type	Class	Kind
Point	*DtPointObjectKind*	16
Line	*DtLinearObjectKind*	17
Area	*DtArealObjectKind*	18
Circle	*DtCircleObjectKind*	19
Text	*DtTextObjectKind*	20

Enumerations for object type fields are defined in *objTypeEnums.h*.

3.7.4. Identifying Control Objects

Control objects have an object name, a string that uniquely identifies the object. The choice of name is arbitrary, but it must be unique within the simulation exercise. The object name is the identifier saved with the object in the order of battle and object map files, and is the means by which control objects are identified in entity plans.

Control objects are expressed on the network as environmental processes. At the VR-Link level, they are expressed as data PDUs (the term PDU is used loosely to represent DIS PDUs or HLA messages.) At the VR-Forces level, control objects are configured to instantiate network interfaces that implement the sending and receiving of environment process PDUs over the network.

Control objects also have a force type, stored in its state-repository (*DtVrfObjectBaseStateRepository*). By default, the force type is *DtForceTypeOther*.

3.8. The Object Parameter Database API

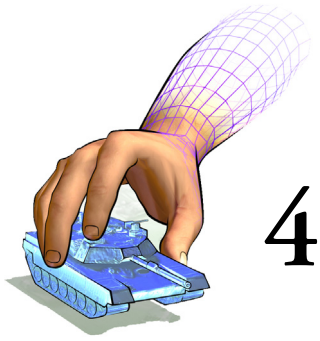
DtVrfParameterDb (*vrfParamDb.h*) is the base object parameter database class. It holds a collection of *DtVrfObjectParameters* (*vrfObjParams.h*) entries, keyed on *DtObjectType*. It has member functions for adding, removing, and looking up *DtVrfObjectParameters* entries. It also includes an interface for load() and save(), but with an empty implementation. You can create derived types of object parameter database classes that know how to load and save themselves from file in whatever format you choose.

VR-Forces has one kind of derived *DtVrfParameterDb* type, *DtMtlParameterDb* (*mtlParamDb.h*). *DtMtlParameterDb* extends the base class by adding the ability to load from and save to *DtReaderWriter* (MTL) format. The *DtMtlParameterDb* class knows how to read and write object parameter database files in this format and in the (pre-VR-Forces 3.7) single file format.

The *DtCgf* class is responsible for creating the object parameter database object. By default, it creates an instance of type *DtMtlParameterDb*. You can override this behavior and have VR-Forces create some other kind of derived *DtVrfObjectParametersDb*. To do this, register a static creator function for the new kind of *DtVrfObjectParametersDb* with the *DtDefaultVrfCreator* (*defVrfCreator.h*) class before initializing the *DtCgf*, for example:

```
DtCgf cgf;
. . .
DtDefaultVrfCreator creator;

// Specify the base parameter database type, DtVrfParameterDb. This
// is the simplest kind of parameter database. It does not serialize
// itself to file in MTL format.
creator.setParameterDatabaseCreatorFcn(DtVrfParameterDb::create);

// Initialize the CGF, passing in our altered creator.
cgf->init(exerciseConn, &creator);
```

The example createParamDbDynamically demonstrates how to use a different type of object parameter database with VR-Forces.



4

Entities

This chapter describes how to create and manipulate entities. It also describes entity parameters and resources. It assumes you are familiar with the concepts and classes described in the preceding chapters. Entity components and ports are described in Chapter 5, *Component Concepts*.

4.1. Entities

VR-Forces simulates individual entities and aggregate entities. Entities are simulated objects (*DtVrfObjects*) that correspond approximately to HLA RPR FOM and DIS platforms and lifeforms. In VR-Forces, aggregate entities are modelled as a single entity that represents an organizational structure (an aggregated entity) or as individual entities that represent an organizational structure (a disaggregated entity). The members of a disaggregated entity can be individual entities or smaller aggregated entities.

VR-Forces uses the following mechanisms to simulate entities. These mechanisms are discussed in detail in this chapter, and in some cases, in later chapters in this manual.

- Highly parameterized models, which allow you to change the characteristics of entities.

  VR-Forces Configuration Guide describes how to set parameter values in the object parameter database. The important point to understand as a developer, is that when you create new components, you need to consider whether or not you want the components to have parameters that users can set in the object parameter database, and if so, design the components accordingly. For more information, please see "State Repositories for Entities," on page 4-17.

- A component architecture and Component Manager.

  To simulate entity behaviors and carry out tasks, VR-Forces uses a component architecture similar to that used in many robotics applications. The Component Manager creates, connects, and maintains the lists of components. Please see "Components and The Component Manager," on page 4-5.

- The Organization Manager. The Organization Manager maintains an entity's place in the organizational hierarchy. Please see "The Organization Manager," on page 4-11.
- Process state repositories and the Process State Repository Manager. Process state repositories maintain the state of an entity. When you save or checkpoint a scenario, the information in the process state repository is written to the order of battle file. Please see "State Repositories for Entities," on page 4-17.
- The Task Manager. The Task Manager does the processing and message handling required to execute a task. It does not execute specific tasks. Please see "The Task Manager," on page 4-28.
- Resources and the Resource Manager. The Resource Manager tracks entity resources, such as ammunition, fuel, and so on. Please see "The Resource Manager," on page 4-30
- The Plan Manager. Each entity has a Plan Manager to manage its plan. For details, please see Chapter 11, *Plans*.
- Radios. VR-Forces simulates a radio-like communication system for sending tasks to entities and receiving reports from entities. Each entity has a radio. For more information about radios, please see "Entity Communication," on page 4-35.

4.1.1. The Basic Structure of an Entity

As stated previously, all entities are *DtVrfObjects*. VR-Forces does not use derived classes to distinguish different types of entities, such as ground vehicles, missiles, and so on. They are distinguished from each other and from other objects by the state repository and other subcomponents that get created when they are instantiated. An entity is expected to have the following subcomponents:

- State repository (*DtVrfObjectStateRepository*)
- Network interface (*DtSimObjectNetInterface*)
- Task Manager (*DtSimTaskManager*) (local entities only)
- Plan Manager (*DtVrfObjectPlanManager*) (local entities only)
- Component Manager (*DtSimComponentManager*) (local entities only).

The particular subcomponents that get created for an entity are determined by the subcomponents section of the entity's parameter block in the object parameter database.

4.2. Components and The Component Manager

VR-Forces supports three basic types of components – sensors, controllers, and actuators. The component architecture (Figure 4-1) supports any number of sensors, controllers, and actuators in a simulated entity. Components communicate with each other through ports.

All components are derived from the class *DtSimComponent* (in *simCmpnt.h*), or a subclass of *DtSimComponent*. Components are specified in the object parameter database file by using *DtComponentDescriptors* (*compDesc.h*). Component descriptors can also provide parameter information to a component. Each local entity has a Component Manager, *DtSimComponentManager* (in *compMgr.h*).

The Component Manager creates, connects, and maintains the lists of *DtSimComponents* (*simCmpnt.h*) (sensors, controllers, and actuators) for an entity. Entities that are configured with a Component Manager have an instance of a *DtSimComponentManager* (*compMgr.h*). Only local entities should be configured with a Component Manager.

Groups of related sensors, controllers, and actuators can be assembled into *DtComponentSystems*. A *DtComponentSystem* is a set of *DtSimComponents* that are managed as a group. They typically represent some logical subsystem of an entity, such as a weapon or movement system.

This section describes the purpose and function of each type of component and briefly describes ports and port groups. For detailed information about components, please see Chapter 5, *Component Concepts*.

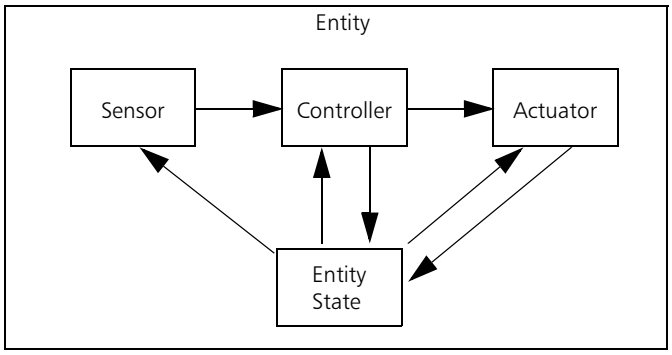


Figure 4-1. Entity component architecture

4.2.1. Sensors

Sensor components provide models of the simulated environment, which are used by controller components to make decisions and perform tasks. The simplest sensor might provide (simulated) ground truth, while more sophisticated ones could use complex models for sensing electromagnetic emissions, or a cognitive model to simulate a soldier or crew member's perception of the simulated world. Sensor components can get information from:

- The virtual network (through a VR-Link exercise connection and reflected entity list)
- From a terrain database
- By monitoring the state of the entity model
- Other sources.

Sensors are derived from *DtSimComponent*. The use of sensors is optional. If a controller only needs to work with ground truth, it can get that information directly, although this may make reuse of the controller component more limited.

4.2.2. Controllers

Controller components typically provide control inputs to actuator components (although this is not required). A subset of controller types register interest in task messages and perform the specified tasks directly, or provide control inputs to actuators to accomplish these tasks. Controllers can use information provided by sensors to perform their functions, but this is not a requirement. For example, the controller that implements the various forms of the Wait task neither requires sensor information, nor provides control inputs to any actuators.

Controllers are created as *DtControllerComponents* (in *cntrlCmpnt.h*) or a subclass. Controllers that receive and implement tasks should derive from *DtTaskControllerComponent* (*taskCtlrCmpnt.h*), because this class provides some basic task management that all of these types of controllers must perform. Each *DtControllerComponent* has a *DtSimBaseRadio* (*baseRadio.h*) for registering interest in task messages.

4.2.3. Actuators

Actuator components:

- Provide the physical model of the entity being simulated
- Change the entity's state
- Interact with other simulated objects (such as by sending messages).

Actuators can use control inputs provided by controller components as parameters to its model each simulation frame. Actuators may:

- Send messages on the radio system
- Generate events on the virtual network, such as detonations
- Modify the state of the entity simulated: position, velocity, damage, and so on.

Actuators can also use data directly from a sensor, when modeled or filtered data is preferred over simulated ground truth for a model that does not require a controller.

Actuators are created as *DtActuatorComponents*. Each *DtActuatorComponent* (*actCmpnt.h*) has a *DtReceiverTransmitter* (*rcvrTrans.h*) for sending and receiving messages.

While the component architecture suggests a model for implementing an entity model, these are guidelines, not rules. For example, you could build an entire entity model as a single component and put it on the sensor list. How you use the component architecture is up to you.

4.2.4. Inter-Component Communication

Components exchange data with each other through ports. There are two kinds of ports: *DtInputPort* (*inputPort.h*) and *DtOutputPort* (*outputPort.h*). Components that produce data (such as a sensor) typically have one or more output ports. Components that accept data as input typically have one or more input ports. Ports may provide data as simple as a single number (such as a throttle value), or more complex data such as a list of entities that have been detected.

Port groups are collections of ports. There are two kinds of port groups: *DtInputPort-Group* (*inputPortGroup.h*) and *DtOutputPortGroup* (*outputPortGroup.h*). Each is a collection of the ports of the corresponding type (*DtInputPort* or *DtOutputPort*). A component can have-a *DtInputPortGroup* or *DtOutputPortGroup* member that it uses to manage collections of ports.

Port groups are useful for creating logical groupings of ports. They indicate that the data handled by the ports in the group is related in some way. For example, the automotive port group contains the throttle, steering, and brake ports. The automotive actuator that makes use of this group expects to receive all three inputs in order to work properly.

Ports and port groups also provide an interface that makes it possible to multiplex and prioritize multiple data sources to a single data sink. For example, the ground vehicle automotive actuator can accept inputs from a collision-avoidance conroller, move-to-point controller, and follow-route controller, but it does not want inputs from all of those controllers at once.

4.3. Creating an Entity

The Object Manager creates entities with the DtVrfObjectManager::createAndInit-VrfObject() member function. An Object Manager may create an entity under the following circumstances:

- It receives a createEntity() call from the remote control API
- It receives a createEntity() call directly from within an application
- It discovers a remote entity on the network and creates a remote entity to represent it
- The back-end reads the order of battle file (readOrderOfBattleFile()) and creates the specified entities.

- If a back-end reads the order of battle file, it does not read the object map file and creates all entities as local entities.
- VR-Forces will not create entities unless a simulation has loaded a terrain and a coordinate system exists.

4.3.1. Creating a Local Entity from Within an Application

If you want to create a local entity from within an application, call DtVrfObjectManager::createAndInitVrfObject(). The createAndInitVrfObject() member function creates and initializes the new *DtVrfObject*, and adds it to the Object Manager. Call the form of createAndInitVrfObject() that takes a single position as an argument. This form is suitable for creating entities that have simple geometry defined by a single vertex. For example, to create a new M1A2 tank, do the following:

```
DtVrfObject * tank;

// M1A2 object type.
// Extensions to entity type enumerations are in objTypeEnums.h
DtObjectType tankType(1, 1, 1, 225, 1, 1, 3, 0);

// Initial position of entity, in local (database) coordinates.
DtVector initialPosition(2000.0, 4000.0, 0.0);

// Ask the Object Manager to generate the next available unique name.
// It will generate a name similar to "M1A2 1".
DtString name = objectManager->nextName(tankType, DtForceFriendly);

tank = objectManager->createAndInitVrfObject(
  tankType,
  true,
  name,
  DtString::nullString(),
  DtForceFriendly,
  initialPosition);
```

The echelon ID is not immediately available when a new organized entity is created. It may take one or more simulation ticks before the echelon ID gets assigned to the new entity. To get notified when the echelon ID is assigned or changed, you can register a callback function with the object. For example:

```
#include "echelonIdChangedPred.h"
DtEchelonIdChangedPredicate pred(tank->echelonIdString());
tank->addVrfObjectPredicateCallback(echelonIdChangedCb, pred);
```

In the callback function echelonChangedCb(), you can examine the echelon ID, for example:

```
static void echelonIdChangedCb(DtVrfObject * obj,
  DtVrfObjectPredicateEvaluator* predicateEval, void * usr)
{
  DtInfo("Echelon Id: %s);
}
```


Do not call the *DtVrfObject* constructor directly. Do all of the creation of *DtVrfObjects* through the Object Manager (through its createAndInitVrfObject()) functions. The Object Manager not only instantiates the object, but initializes and adds the object to itself for management.

4.4. Managing Local and Remote Entities

As described in "How the Object Manager Chooses an Object's Subcomponents," on page 3-7, each VR-Forces application has its own Object Manager. When the Object Manager creates a new entity (through the createAndInitVrfObject() function), it looks up the entry in the object parameter database for the object type to be created, and uses this data to direct the construction of the object.

For example, the following is an excerpt of the M1A2 entry from the object parameter database. It shows the local and remote objects lists.

```
(M1A2
  (parameter-type "ground-vehicle-param")
  ...
  (local-objects
    (state-repository  "ground-entity-state-repository")
    (state-repository-min-tick-period  -1.000000)
    (state-repository-min-tick-period-variance  -1.000000)
    (net-interface  "individual-local-entity-net-interface")
    (net-interface-min-tick-period  -1.000000)
    (net-interface-min-tick-period-variance  -1.000000)
    (task-manager  "local-task-manager")
    (component-manager  "component-manager")
    (plan-manager  "vrfobject-plan-manager")
  )
  (remote-objects
    (state-repository  "vrf-moving-object-state-repository")
    (state-repository-min-tick-period  -1.000000)
    (state-repository-min-tick-period-variance  -1.000000)
    (net-interface  "vrf-individual-remote-entity-net-interface")
    (net-interface-min-tick-period  -1.000000)
    (net-interface-min-tick-period-variance  -1.000000)
    (task-manager  "")
    (component-manager  "")
    (plan-manager  "")
  )
)
```

For remote objects simulated by non-VR-Forces applications, only the state repository and network interface subcomponents are created.

4.5. The Organization Manager

The Organization Manager (*DtOrganizationManager*, in *orgManager.h*) is a subcomponent of the Object Manager. It maintains the organizational hierarchy for all organized entities. Any entity whose object parameter database entry has **is-organized** set to True has an entry in the Organization Manager. An entity's place in the hierarchy is expressed through its echelon ID. (For details, please see Section 4.5.2, "Echelon IDs" and Section 3.5, "Entities", in *VR-Forces Users Guide*.)

The Organization Manager has the following responsibilities:

- It provides an up-to-date hierarchy of all local and remote VR-Forces organized entities
- It provides functions for modifying the hierarchy.

In simulations that use multiple VR-Forces simulation engines, each simulation engine has its own Organization Manager. Each Organization Manager maintains a complete picture of the hierarchy, including all remote VR-Forces entities. Calls to modify the hierarchy can be made to any Organization Manager. The Organization Manager that receives the call, sends the appropriate messages to the other Organization Managers to ensure that the hierarchy is changed correctly.

4.5.1. The Entity Hierarchy

Organized entities in VR-Forces exist in an organization tree beneath a force-level superior. Each force has a force-level aggregate that is locally simulated on each VR-Forces simulation engine and not published to the network. By default, a newly created entity is placed immediately below the force level superior corresponding to the force of the entity. For example, when a new M1A2 is created, it is added as a subordinate of 1 Force. If this M1A2 is the first subordinate of 1 Force to be created, the 1 Force aggregate is created at the same time. You can move entities in the hierarchy with calls to the Organization Manager API, for example:

```
// platoon1 is a DtVrfObject pointer to a platoon aggregate.
// tankA is a DtVrfObject pointer to a tank entity.

// To make tankA a subordinate of platoon1:
mySimManager->vrfObjectManager()->organizationManager()->setSuperior(
  tankA->objectIdentifier(), platoon1->objectIdentifier());
```

The Remote Control API also has function calls to change the entity hierarchy.

- You do not have to create the force-level aggregate. Force-level aggregates are created by the Object Manager the first time that an entity of a given force type is created.
- Remote entities that are simulated by non-VR-Forces applications can participate in the organization by publishing their subordinates. However, VR-Forces cannot control their positions in the hierarchy.

4.5.2. Echelon IDs

Every VR-Forces entity that has an organization controller is assigned an echelon ID that indicates its position in the entity hierarchy.

Chapter 3, *VR-Forces Simulation Concepts*, in *VR-Forces Users Guide* discusses organized entities in detail. To summarize, an echelon ID specifies an entity's position in an aggregate unit and identifies the hierarchical units to which it belongs. Echelon IDs are formatted as follows:

```
designator, [category], [echelon level], superior designator,
[superior category], [superior echelon level], . . .,
top-level designator, [top-level category], [top-level echelon level]
```

For example, the third M1A2 tank in 1st platoon, Force 1, would have the echelon ID string:

```
"3 M1A2, 1 Plt, 1 Force"
```

You can get an organized entity's echelon ID through its echelonId() accessor. The elements that make up the echelon ID (designator, category, echelon-level, and superior) can also be retrieved individually from the entity's echelon ID.

The echelon ID for an entity is not available immediately upon creation. The assignment takes place after the entity is attached to its superior within the Organization Manager. Please see "Creating a Local Entity from Within an Application," on page 4-9 for more information.

How Echelon IDs Are Set

Each organized entity has an organization controller component attached to it (*DtOrganizationController*, in *orgCtlr.h*). Aggregates have a *DtPseudoAggregateOrganizationController* (in *pseudoAggOrg.h*), which is a derived version of *DtOrganizationController*. Additionally, the force level aggregates have their own derived version of the organization controller, *DtTopLevelOrganizationController* (in *topLevOrgCtrl.h*).

DtOrganizationController maintains the echelon ID of the entity it is attached to. It registers for DtSetDesignator messages, which are used to determine the designator that will be used in the echelon ID, and it monitors its superior's echelon ID, using it to reset its own if a change occurs.

DtPseudoAggregateOrganizationController sets its own echelon ID (just like individual entities do) and sends DtSetDesignator messages to its immediate subordinates. This component registers for subordinatesChanged() callbacks from the Organization Manager, and is responsible for making sure that all of its subordinates use different designators, as well as performing reorganizations if necessary.

DtTopLevelOrganizationController is a special controller. It is used only for the force-level aggregates. Force level aggregates are simulated locally on each back-end and are not published. *DtTopLevelOrganizationController* handles these differences.

To illustrate how an echelon ID is built by the organization controllers, consider the echelon ID `1 M1A2, 1 Plt, 1 Force`. From left to right, the echelon ID is created as follows:

- 1 (entity designator) – the entity's *DtOrganizationController* receives a DtSetDesignator message from the platoon's *DtPseudoAggregateOrganizationController*
- M1A2 (category) – specified in the object parameter database **category** parameter
- 1 (aggregate designator) – the aggregate's *DtPseudoAggregateOrganizationController* receives a DtSetDesignator message from the force level *DtTopLevelOrganization-Controller*
- Plt (category) – specified in the object parameter database **category** parameter
- 1 (force-level designator) – managed by the *DtTopLevelOrganizationController*
- Force – specified in the object parameter database **echelon** parameter.

Figure 4-2 illustrates changes to an entity hierarchy and changes to the echelon IDs to show each entity's place in the hierarchy. In the initial order of battle, the echelon IDs show that each entity is subordinate to the force level. In the second order of battle, four of the entities are organized into a platoon and their echelon IDs have changed to show that the platoon is their superior. In the third order of battle, one of the entities is removed from the platoon. It is superior is the force level again. Notice that it does not have the same echelon ID it had when it was originally under the force level. Also, note that the echelon IDs for the remaining subordinates of `1 Plt` did not change. When the membership of an aggregate changes, the echelon IDs do not change unless you enable auto-reorganization or manually reorganize the aggregate. For more information about reorganization, please see "Reorganizing Aggregate Entities," on page 4-16.

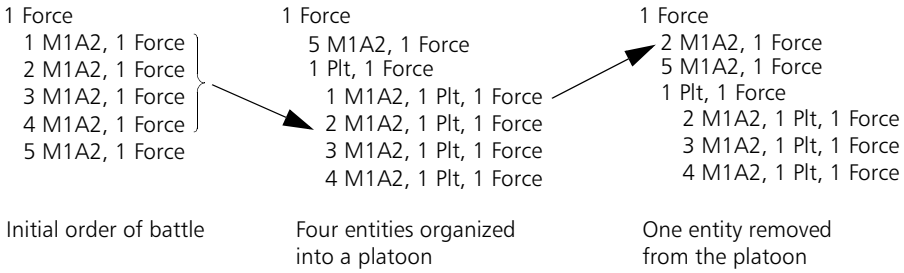


Figure 4-2. Changes to the entity hierarchy

4.5.3. How the Organization Manager Works

The Organization Manager has two parts – the *DtOrganizationView* (in *orgView.h*) and the *DtOrganizationManager*.

The *DtOrganizationView* listens to the network and maintains a set of interconnected nodes (*DtOrganizationManagerNode*, in *orgMgrNode.h*) that correspond to the current organization state of all entities in the exercise. These nodes are in the form of a set of trees, with the root node of each tree being a force level aggregate. A node does not have to be part of a tree at all times. It needs to be independent when an entity is in transition from one location in the hierarchy to another, or when an entity is first created.

The organization information is sent over the network as part of aggregate state PDUs. Each aggregate publishes a list of subordinates. Additionally, VR-Forces objects publish whether they are a subordinate of the force level aggregate, because the force level aggregates are unpublished objects. From this information, the *DtOrganizationView* keeps its representation of the organization synchronized with the other Organization Managers in the exercise (if any).

The *DtOrganizationManager* class manipulates the entity hierarchy. It uses the setSuperior() and setSuperiorToForce() member functions, to assign entities as subordinates of published aggregates or the force-level aggregate. These function calls result in modification of the subordinate lists maintained by each aggregate's subordinate manager. The modifications to the hierarchy are not observed by *DtOrganizationView* until the aggregates publish their updated subordinate lists.

You can make an API call to modify the hierarchy from any back-end's Organization Manager. An Organization Manager can only change the state of local aggregates (modify their subordinate lists). Therefore when you make a request that requires action by multiple Organization Managers, the Organization Manager that receives the request sends messages to remote Organization Managers to enlist their help in carrying out the request. Each Organization Manager makes the changes required to the entity hierarchy on its back-end.

4.5.4. Querying the Organization Hierarchy

The *DtOrganizationView* maintains the entity hierarchy for each Organization Manager. You can use the following member functions to determine an entity's superior and its subordinates:

- isSubordinateOfTopLevel()
- superior()
- superiorVrfObject()
- isSubordinateOf().

DtOrganizationView::lookupNode() returns a pointer to the *DtOrganizationManagerNode* for an entity in the organization hierarchy. You can use this node to find more detailed organization information about a particular entity, such as a list of subordinate identifiers.

4.5.5. Getting Notification of Changes to the Hierarchy

If you need to know when an entity's subordinate list changes, you can register for callbacks with the Organization Manager. To register a callback, call DtOrganizationManagerNode::addSubordinatesChangedCallback(). The callback is invoked any time a subordinate is added or removed from a node in the Organization Manager. You can register this callback on both local and remote VR-Forces objects.

4.5.6. Modifying the Entity Hierarchy

You can change the entity hierarchy using the Simulation API and the Remote Control API. (For details, please see "Changing the Entity Hierarchy," on page 12-11.)

Changing the Entity Hierarchy Using the Simulation API

The *DtOrganizationManager* class has the following member functions for changing the entity hierarchy:

- setSuperior()
- setSuperiorToForce()
- queueSetSuperior()
- queueSetSuperiorToForce()
- detachFromSuperior().

The setSuperior() and setSuperiorToForce() member functions identify entities by their echelon IDs. The entities and superiors must exist.

The queueSetSuperior() and queueSetSuperiorToForce() member functions identify entities by name. The entities do not have to exist. These functions are used when VR-Forces loads a scenario.

The detachFromSuperior() member function removes a subordinate from its superior's subordinate list. The entity is then left without any parent in the organization hierarchy. This call is made implicitly by the setSuperior() and setSuperiorToForce() calls. Normally, you should not use it.

4.5.7. Aggregate Organization

The organization of entities within an aggregate unit is expressed by the numeric designator assigned to each member entity in its echelon ID. The lower the designator, the higher the entity is in the chain of command. For example, in the following list of subordinates of a hypothetical platoon, the lead subordinate is `1 M1A2, 1 Plt, 1 Force`. It has the lowest designator, the "`1`" in `1 M1A2`.

```
1 M1A2, 1 Plt, 1 Force
2 M1A2, 1 Plt, 1 Force
3 M1A2, 1 Plt, 1 Force
4 M1A2, 1 Plt, 1 Force.
```

Reorganizing Aggregate Entities

Aggregate reorganization does not apply to aggregates in the aggregated state. For details about the differences between agregated and disaggregated aggregate entities, please see Section 3.6, "Aggregate Entities", in *VR-Forces Users Guide*.

If entities get removed from an aggregate or get destroyed, you can reorganize the aggregate. When you reorganize an aggregate, the echelon IDs get reassigned so that the lowest numerical designators are assigned to functioning (non-destroyed) entities. The benefit of reorganization is that if plans are associated with the echelon IDs with the lowest numerical designators, entities will continue to carry out the plans.

The reorganization controller has two modes – automatic and manual. In auto-reorganization mode, the reorganization controller continually monitors the health and status of its member entities, and reorganizes the unit the instant it determines that it is necessary. In manual mode, a reorganization of the unit only occurs when the controller receives a DtSetReorganizeRequestType interface message from the front end.

To manually reorganize an aggregate entity, you need to send a DtSetReorganizeRequestType interface message.

Reorganization can be a drastic change, in that the echelon ID names of the member entities in the organization are changed. The changes are permanent (at least until the next time a reorganization occurs). This means that if a leader is destroyed, and the unit gets reorganized, the original leader's identity is altered. Restoring a destroyed leader after a reorganization has taken place with a Set → Restore action in the GUI, restores the entity to full health and stores, but does not give back its original identity as leader of the aggregate unit. If all the members of an aggregate are the same type of entity, such as tanks, it probably does not matter which one is the leader, but if they are different types of entities, such as a mix of trucks and tracked vehicles, it might make a difference which one is the leader.

4.6. State Repositories for Entities

The role of state repositories and the state repository hierarchy has been discussed in previous chapters. All entities have either a *DtVrfMovingObjectStateRepository* (*vrfMvObjSR.h*) or a state repository derived from *DtVrfMovingObjectStateRepository*. For more information, please see the class documentation for *DtVrfObjectStateRepository* (*./doc/classdoc/simclassref/class_dt_vrf_object_state_repository.html*). Individual entities need derived state repositories. The state repositories allow you to examine platform-specific state information.

Because the *DtVrfMovingObjectStateRepository* and its children are descended from *DtVrfObjectStateRepository*, they all have a Resource Manager. (For details, please see "The Resource Manager," on page 4-30.)

4.6.1. Entity Parameters

This section assumes you are familiar with the information in "Object Parameters," on page 3-13 and Chapter 7, *The Object Parameter Database* in *VR-Forces Configuration Guide*.

Each platform-specific state repository overrides createParameters() to create an instance of an appropriate parameters class. The entity parameter class hierarchy (Figure 4-3) looks just like the state repository hierarchy. Each state repository instantiates a parameter class to hold the parameters that are specific to that state repository. Therefore, as with state repositories, all entities have a parameter type of *DtMovingObjectParameters* (*mvObjParams.h*) or a parameter type derived from *DtMovingObjectParameters*. Aggregate entities can be modeled with *DtMovingObjectParameters*. Individual entities need derived state repositories. When you create or configure entities, you must be sure that the state repository and parameter classes that you specify match each other.

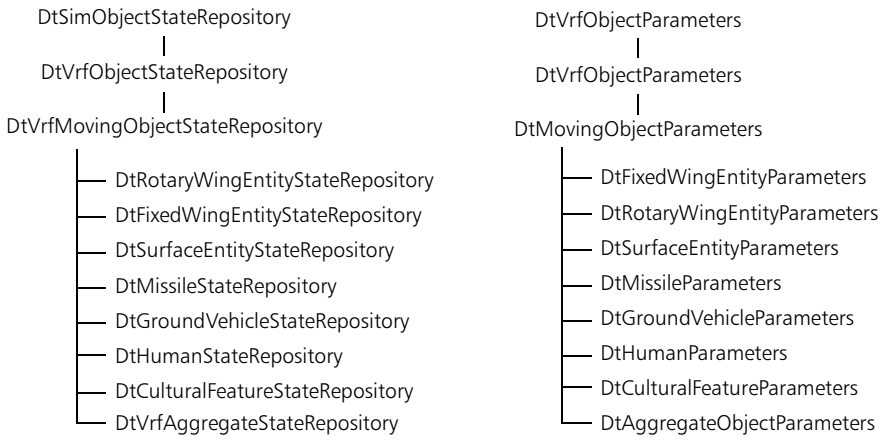


Figure 4-3. Object parameter class hierarchy

In Figure 4-3, *DtSimObjectStateRepository* and *DtVrfObjectParameters* are listed to show their place in the class hierarchy. However, you should never create an instance of these classes. Specifying a parameter type *DtVrfObjectParameters*may result in a run-time error.

DtVrfObjectParameters (*vrfObjParams.h*) derives from *DtReaderWriter*, allowing instances of a parameter class to be read or written. The parameter classes also use the *DtReaderWriter* capability to indicate whether or not a value was specified for a given member.

DtVrfObjectParameters provides the necessary infrastructure for working with the parameter factory. As with most classes that use factories, the type() member function returns the class type string, as found in *paramTypes.h*.

Each parameter class provides accessors and mutators for the parameters it supports, as well as supplying a default value for each parameter in the constructor.

4.6.2. Parameter Type Strings

Parameters have a parameter type string. This string is used by the Object Manager to create the proper parameters class when it creates an entity. The parameter type should match the state repository type specified for the entity. For example, an entity with the state repository type **ground-entity-state-repository** needs certain parameters that are specific to ground vehicles, and therefore expects parameters that are of parameter type **ground-vehicle-param** or a type derived from **ground-vehicle-param**. To see the type string for a state repository class, please see the class documentation for the type() member frunction for that class. The documentation for DtVrfObjectStateRepository is at *./doc/classdoc/simclassref/class_dt_vrf_object_state_repository.html*.

As discussed in "Object Parameters," on page 3-13, a database of all the parameter information is maintained within the Simulation Manager. The data is stored in ASCII format, so you can edit it.

Parameters use the factory system, so it is easy to create new parameter types for your models to use. You can create entity-wide parameters, or parameters for specific components. Parameters for components are specified within the component descriptors, which are then used to indicate which components to include with an entity.

Certain components in the system need so much data that it is worth the effort it takes to store the data in a separate file to organize it better. The *DtReaderWriterFile* (*rwFile.h*) class, makes this easy to do for any parameters you may need to represent. *DtReaderWriterFile* lets you give a *DtReaderWriter* (*rdrWritr.h*) a derived object and a file name, and it handles opening, reading, writing, and closing the file automatically.

For more information about parameters, please see "Object Parameters," on page 3-13 and "Entity Parameters," on page 4-17.

4.6.3. Parameter Inheritance

Member variables for which no value has been specified can inherit values from other instances of parameter classes that are logical ancestors within the parameter class hierarchy. This is done using two virtual member functions, inheritValues() and inheritValuesFromThisType(). If you derive a new parameters class from an existing one, you need to override these two member functions.

The inheritValues() member function takes a pointer to another parameter object that it may inherit from. The type passed is a *DtVrfObjectParameters**, although the actual parameter object may be a derived form. Using the parameter argument's type() member function and its own type() member function, inheritValues() determines if the argument passed in has the same exact type as the type of the object doing the inheriting. If it does, it calls inheritValuesFromThisType(). For the DtSimObjectParameters::inheritValues() member function, if the type does not match, the function returns. For all others, though, if the type does not match, then inheritValues() calls the inheritValues() member function of the type that it derives from, thus passing the parameter argument up the class derivation tree.

All versions of inheritValues() (except for DtSimObjectParameters::inheritValues()) look the same except for the invocation of the base class's inheritValues() member function.

The following example shows the implementation of inheritValues() for the *DtGroundVehicleParameters* class, which is derived from *DtMovingObjectParameters*.

```
bool DtGroundVehicleParameters::inheritValues(
  DtVrfObjectParameters * benefactor)
{
   if (!benefactor)
   {
      return false;
   }

   if (benefactor->type() == type())
   {
      inheritValuesFromThisType((DtGroundVehicleParameters *)benefactor);
      return true;
   }
   else
   {
      return DtMovingObjectParameters::inheritValues(benefactor);
   }
}
```

The inheritValuesFromThisType() member function is passed the same parameter argument received by inheritValues(). It checks each member to determine if a value has already been specified for it. If no value has been specified for the member, and a value has been specified for the corresponding member in the parameter argument, the value from the parameter argument's member is copied to the local member. After this has been done for each member, inheritValuesFromThisType() calls the inheritValuesFromThisType() member function of the class it derives from.

All the derived parameters classes function similarly. Each provides accessors and mutators for all the entity parameters specified for that type. They all override the type() and creator() member functions as required for all classes using the factory scheme. They also override inheritValues() and inheritValuesFromThisType(), as discussed in the previous paragraph.

DtVrfObjectParameters includes the lists of component descriptors specifying the components to be used for an entity type. The inheritValuesFromThisType() member function treats the list as a whole unit. If any component descriptors are found on a list, no inheritance takes place. If no component descriptors are specified on a list, then the parameter object can inherit an ancestor's entire list.

DtGroundVehicleParameters includes a list of soil factor (*DtSoilFactor*) entries. Unlike the component descriptors in *DtVrfObjectParameters*, these individual entries may be inherited from an ancestor, even if other entries already exist. For example, a parameter entry might only specify a soil factor entry for one soil type that it wishes to change. Entries for other soil types from an ancestral entry would be inherited in inheritValuesFromThisType().

4.7. Process State Repositories

To enable entities to save and restore their state as part of a scenario (or checkpoint), each VR-Forces object has a *DtVrfProcessStateRepositoryManager* (in *procSRMgr.h*), which maintains the set of *DtVrfProcessStateRepositories* (*processSR.h*) used by an entity. Process state repositories are used to share state data among the components needed by a particular entity subsystem (such as a weapon system, or movement system).

The use of process state repositories is not intended to circumvent the use of ports and port groups for communication between components. Use ports and ports groups to communicate information such as the steering and throttle inputs between an automotive controller and an automotive actuator. (For details, please see "Ports and Port Groups," on page 5-20.) By contrast, the process state repository does not facilitate communication; it holds state information such as "how long have I been waiting?" or "what gear am I in?" which multiple components might want access to. The components should not have to connect to one another to exchange this data. By placing state information in a process state repository, we only have to maintain a single copy of the information, easily accessible to a number of components.

Process state repositories are derived from *DtReaderWriter*, which gives them the ability to read and write themselves to file, so state information can be saved to the order of battle file. The name inherited from *DtReaderWriter* is used as a key into the Process State Repository Manager, enabling lookup of process state repositories by name. An entity can have multiple process state repositories of the same type, but with different names. This makes it possible to configure an entity with multiple similar subsystems, such as multiple weapons.

When you use multiple subsystems, you must use a unique process state repository name for each subsystem. Failure to do so could result in a crash if multiple subsystems attempt to modify or destroy the same process state repository. For example, if you wanted to create an entity with multiple indirect fire systems, each *DtIndirectFireActuator* and *DtIndirectFireWeaponController* component pairs which make up a subsystem must use a process-state-repository-name which is unique to that pair

The *DtVrfProcessStateRepositoryManager* holds the list of *DtVrfProcessStateRepositories* for an entity, keyed on their reader/writer name. It does not create or delete any of the process state repository objects used by the object. Specific components are responsible for creating and deleting them.

All process state repositories are derived from *DtVrfProcessStateRepository* (*processSR.h*). The class documentation has a comprehensive list of derived process state repository classes on the *DtVrfProcessStateRepository* page.

Derived process state repository classes contain parameters and state member data appropriate to different categories of processes. For example, the *DtAutomotivePSR* holds all of the state and parameters needed for processes associated with ground vehicle movement, It contains data members such as the name of the waypoint the entity was heading toward, and the current gear the vehicle is in. All of the components used to simulate ground vehicle movement use the *DtAutomotivePSR* to either examine or set states associated with that process.

Components use the process state repository as a place to set or retrieve state as needed. For example, the weapon process state repository (*DtVrfWeaponPSR*) holds the **target-acquired** state for a given weapon system. When the *DtWeaponController* acquires a target in its acquireTarget() member function, it sets the **target-acquired** status in the process state repository, for example:

```
if (weaponCorrectlyOriented())
{
  myWeaponProcessState->setTargetAcquired(true);
  . . .
}
```

The *DtBallisticGunController* queries the weapon process state repository for the target acquired status in its tick, for example:

```
if (myProcessState->munitionLoader().isLoaded() &&
  myProcessState->targetAcquired() &&
  permissionToFire())
{
  fire();
}
```

Process state repositories are created through the standard factory mechanism used in VR-Forces. (For details, please see "VR-Forces Factories," on page 2-10.) The string constants used as keys to the process state repository factory are defined in *procSR-Types.h*.

4.7.1. Creating and Configuring Process State Repositories

A set of components working together to accomplish a task can be thought of as a subsystem. Typically the set of components making up a subsystem share a single process state repository. For example, the set of controllers and actuators that logically represent a single weapon share the same weapon process state repository. One of the components is responsible for creating and registering the process state repository with the Process State Repository Manager. The remaining components look up and cache a pointer to the process state repository. Typically, the actuator is the component that creates the process state repository, and the rest of the components (typically controllers) keep a pointer to it.

A component that creates a process state repository does so when it is initialized (by calling its createPSR() member function from its init() function, during the entity creation process). The component that creates the process state repository is also responsible for registering it with the entity's Process State Repository Manager. The component tells the Process State Repository Manager the name of the object, and providing a pointer to the object. This makes the process state repository available for lookup-by-name by any interested components. It also ensures that the contents of the process state repository are saved and restored as part of the entity's state in a scenario save or checkpoint. The component that creates the process state repository is also responsible for removing it from the Process State Repository Manager and deleting the memory associated with it. This typically happens in the component's destructor.

If a component is responsible for creating a process state repository, its component descriptor specifies the type and name of the process state repository to create.

For backwards compatibility, if a descriptor entry in the object parameter database does not specify a process state repository, then the component uses the appropriate default names and types defined in *procSRTypes.h*.

For example, the *DtAutomotiveActuatorComponent* creates the *DtAutomotivePSR* for ground vehicles. The entry in the object parameter database is as follows:

```
. . .
  (actuators
    (powertrain
      (component-descriptor-type "automotive-component-descriptor")
      (component-type  "automotive-actuator")
      (process-state-repository-name "vrf-automotive-process-state")
      (process-state-repository-type
        "vrf-automotive-process-state-repository")
    )
. . .
```

This tells the automotive actuator to create a process state repository of type *DtAutomotivePSR* with the name **vrf-automotive-process-state**.

Since one of the components in a subsystem has created the process state repository, the rest of the interested components must look up the process state repository by name in the Process State Repository Manager. Components that need to look up and cache a pointer to a process state repository override the postAddComponentsInit() member function (a member function that gets invoked by the Component Manager for each component, after all components have been created and initialized). The name of the process state repository to create is given in its component descriptor. For backwards compatibility, if no process state repository name is specified in the descriptor's entry in the object parameter database, then the appropriate default name is used (default process state repository names are defined in *procSRTypes.h*).

For example, for ground vehicles, all controllers derived from *DtGroundAutoController-Component* need to look up the *DtAutomotivePSR* (created by the *DtAutomotiveActuatorComponent*). A typical set of descriptor entries for these components would look like the following:

```
. . .
 (move-to
   (component-descriptor-type "ground-move-to-descriptor")
   (component-type  "ground-auto-move-to-controller")
   (process-state-repository-name "vrf-automotive-process-state")
 )

 (move-along
   (component-descriptor-type "ground-move-along-descriptor")
   (component-type  "ground-auto-move-along-controller")
   (process-state-repository-name "vrf-automotive-process-state")
 )
. . .
```

This example indicates that both the move-to and move-along controllers should look up the process state repository by the name **vrf-automotive-process-state** in the Process State Repository Manager.

4.7.2. Extending Process State Repositories

Because process state repositories are saved as part of an entity's state in the order of battle file, all of the state member data that needs to be preserved in a scenario or checkpoint should be kept in a process state repository. Components are not saved as part of a scenario, so any state member data in those classes is not preserved when a scenario gets saved. If you are extending the toolkit by adding or extending components, you may want to extend the process state repositories to include any new process-related state you introduce.

To extend a process state repository, do the following:

1. Derive a new *DtVrfProcessStateRepository* class, providing implementations for the standard constructor, type(), and creator functions. Add the new state member data (derived from *DtReaderWriter* to ensure they get saved in the entity's order of battle entry).
2. Register the new class's creator with the process state repository factory.
3. Edit the object parameter database, specifying that the new type of process state repository be created in the appropriate component descriptor.
4. Set/get the new state data in the process state repository in your components.

Deriving a New Process State Repository Class

For example, suppose you have added a new behavior to ground vehicles that takes driver fatigue into account (you have modified or extended existing automotive components to model and keep track of that information). The automotive-related components all share a *DtAutomotivePSR* object, so start by creating a derived class, *MyAutoPSR*:

```
class MyAutoPSR : public DtAutomotivePSR
{
  . . .

  virtual DtString DtMyAutoPSR::type() const;

  static DtVrfProcessStateRepository* creator();

  // Set/get fatigue level
  virtual void setDriverFatigue(double fatigue);
  virtual double driverFatigue() const;

protected:

  // Define a driver fatigue level between 0 (not fatigued) and 1
  // (completely exhausted)
  DtRwReal myDriverFatigue;
}
```

In your derived class, make sure to register your new data member with the *DtReader-Writer* parent registry (this ensures that the item will be included as part of the object's data that gets written to file).

```
MyAutoPSR:: MyAutoPSR (DtVrfObject* vrfObject, const DtString& rwName,
  DtReaderWriterRegistry*  parentRegistry) :
  DtAutomotiveProcessStateRepository(vrfObject, rwName, parentRegistry),
  MyDriverFatigue("driver-fatigue", 0.0, &myRegistry)
{
}
```

Implement the type() and creator() functions. They are needed by the standard factory mechanism to incorporate your new class into the application.

```
DtString DtMyAutoPSR::type() const
{
  // This is the string that gets registered with the process state
  // repository factory.
  return "my-automotive-process-type";
}

DtVrfProcessStateRepository* DtMyAutoPSR::creator(const DtString& rwName,
  DtVrfObject* vrfObject, DtReaderWriterRegistry* parentRegistry)
{
  return new DtMyAutoPSR (vrfObject, rwName, parentRegistry);
}
```

Registering the Creator for the New Process State Repository

In *main.cxx*, register the creator for your new process state repository class with the process state repository factory.

```
 app.init();

// Register the new process state repository. The string passed into this
// function is the same as the string returned by the MyAutoPSR's
// type() function.
app.cgf()->factoryManager()->processStateRepositoryFactory()->
  addCreatorFcn(
  "my-auto-process-state-repository", DtMyAutoPSR::creator);

app.mainLoop();
```

Editing the Object Parameter Database

Modify the object parameter database entry for the entities that will use the new process state repository. The component responsible for creating the automotive process state repository is the automotive actuator, so you must change the type of process state repository it specifies to create in its descriptor.

```
. . .
  (actuators
    (powertrain
      (component-descriptor-type "automotive-component-descriptor")
      (component-type  "automotive-actuator")
      (process-state-repository-name "vrf-automotive-process-state")
      (process-state-repository-type
        “my-auto-process-state-repository")
    )
. . .
```

This causes the new process state repository to get created instead of the base class version normally created by VR-Forces.

Getting State Data from the Components

Finally, get the data that is in your components. Each component derived from *DtAutomotiveController* has a pointer to the process state. In a derived controller, you can get a pointer to the new process state repository in the following manner:

```
MyAutoPSR* autoPSR = dynamic_cast<MyAutoPSR*>(myProcessState);

if(autoPSR)
{
  // Now we can safely retrieve and/or set driver fatigue
  DtInfo(“fatigue level %f\n”, autoPSR->driverFatigue());
  . . .
```

Similarly, if you have derived a new kind of automotive actuator, you can cast its **myProcessState** member to the new *MyAutoPSR* type to access the new driver fatigue state.

Whenever you save a scenario containing your modified ground vehicle (which takes driver fatigue into account), the process state repository entry in the order of battle file will contain your new state member data, for example:

```
. . .
  (vrf-automotive-process-state-repository-default
    (current-gear  0)
    (heading-to-turn-to  0.000000)
    (K-turn-reverse-goal-heading  0.000000)
    (K-turn-reverse-complete False)
    (control-object-name  "")
    (entity-to-follow  "")
    (following-offset-vector  0.000000 0.000000 0.000000)
    (route-name  "")
    (current-route-vertex  0)
    (first-waypoint  "")
    (second-waypoint  "")
    (collidee  "")
    (is-colliding False)
    (is-clearing False)
```

```
        (avoiding-left False)
        (driver-fatigue 0.5)
      )
. . .
```

When you restore a scenario that contains your new kind of entity, your new process state repository data is restored.

4.8. The Task Manager

Local entities that need to respond to and carry out tasks must have a Task Manager, *DtSimTaskManager* (*taskMgr.h*). (Please see Section 7.1.4, "Local and Remote Subcomponents" in *VR-Forces Configuration Guide* for information about configuring an object with a Task Manager.) The Task Manager works with an entity's Component Manager to handle all tasks for a specific entity, skip task requests, and to report task completion to superiors and the entity's Plan Manager. The Task Manager keeps a list of all components that have registered for various task types, and is responsible for notifying those task controllers when there is a task they need to execute. The Task Manager clears any previous tasks out of the controllers before starting a new task. Subtasks are the exception; they are allowed to run simultaneously with other tasks, but only one top level task is allowed at any one time. The Task Manager does not handle the execution of specific tasks, such as Follow Route or Move To. The execution of specific tasks is handled by task controller components. The Task Manager is also responsible for reporting what tasks an entity is capable of executing. Any task type for which a controller has registered is considered to be a task which can be executed.

This section is primarily for conceptual purposes. As long as an entity has a Task Manager, there is very little that you as a developer need to do to make sure it is doing its job.

To access an entity's Task Manager, do the following:

```
DtSimTaskManager* taskManager = vrfObject->taskManager();
```

The first time the Task Manager is ticked, it calls awaitingTask() on the *DtPlan* for the entity to get it to issue the first task from the plan.

The following sections explain several task management functions. You do not normally have to do anything special to implement them.

The Task Manager is also responsible for task interruption. It uses the task execution rules to know what types of tasks interrupt other types of tasks. When a new task is starting execution, the Task Manager checks the execution rules, and stops any top level tasks that are not allowed to run concurrently with the new task. Newer tasks always take precedence over older tasks. For details about configuring task execution rules, please see Section 8.8, "Configuring Task Execution Rules", in *VR-Forces Configuration Guide*.

4.8.1. Reporting that a Task is Complete

The Task Manager sends Task Complete messages when an entity completes execution of a task. (It does not send a Task Complete message when an entity completes a subtask.) Controller components call DtSimLocalTaskManager::taskComplete() to do this, passing in the original task. The Task Manager fills out the report and sends it to whoever originally issued the task. (The sender of the task may be a superior aggregate or a force-level aggregate.) It also calls awaitingTask() on the entity's *DtPlan* to get it to issue the next task in the plan.

4.8.2. Skipping a Task

The skipTask() function handles *DtIfSkipTask* messages sent to the entity. In skipTask(), the Task Manager calls the Component Manager's controller list clearTask(), and then calls awaitingTask(). This effectively causes the entity to stop executing any current task, and notifies a *DtPlan* (if one exists for the entity) that it is ready to receive its next task in the plan. Skip task messages are processed only if the entity is independently tasked or the message was sent from the entity's superior.

4.8.3. Responding to the Tasked by Superior Request

When you task an entity from the GUI, you essentially remove it from the control of its superior in the entity hierarchy. This puts the entity into an 'independently-tasked' state – it no longer participates in the execution of tasks coordinated by its immediate superior. You can return control of an entity back to its superior by sending it a *DtSetTaskedBySuperiorRequest* (*setTskBySupr.h*) message. The local Task Manager handles these messages. In the taskedBySuperiorRequestCallback(), it sets the independently tasked state to `false`, returning control of the entity back to its superior entity.

4.8.4. Handling Clear Task Messages

Aggregates can issue Clear Task requests to subordinates to manage how they execute a behavior. Each entity registers for these requests automatically through its Task Manager. To process the message, the local Task Manager looks up the task controller handling the type of task specified in the request. Then it invokes its clearTask() member function.

4.9. The Set Data Manager

All entities have a Set Data Manager, *DtSetDataManager* (*setDataManager.h*). This class listens for all set data requests that arrive for the entity over the radio, and notify any components which have registered an interest in the set data request.

All components that need to be notified when a set data request arrives for a specific entity need to register a callback with the Set Data Manager. This callback can be registered either as a processor callback or a notify callback using addSetDataProcessorCallback() or addSetDataNotifyCallback().

Any controllers that will actually change state data register as a processor callback, and any controllers that want to be notified of the request, but will not actually carry the request out, register as notify callbacks. Any number of processor and notify callbacks are allowed for any given set request type. The only functional difference between a notify and processor callback is that the Set Request Manager reports that the entity is capable of all sets that have at least one processor callback registered. This determines what set data types are enabled on the GUI when the Set Menu is opened for a particular entity. set data request types that have only notify callbacks registered are not enabled in the GUI.

4.10. The Resource Manager

Entities simulated by VR-Forces can track and manage arbitrary resources. Each VR-Forces entity has a Resource Manager, which is an instance of the class *DtObjectResourceManager* (*objectResourceManager.h*). It is accessible through the entity's *DtVrfObjectStateRepository*, for example:

```
entity->vrfState()->objectResourceManager()
```

The *DtObjectResourceManager* is the top-level resource manager for the entity. The actual resources are stored in *DtSimResourceManager* instances owned by the *DtComponentSystems* on the entity, but this manager has access to all these sub-managers, and is able to look up resources in any of them. The *DtSimResourceManager* associated with a given *DtComponentSystem* can be reached through the resourceManager() accessor of any of its *DtSimComponents*. The rest of this section describes the capabilities of *DtSimResourceManager*. *DtSimResourceManager* provides an interface for adding, removing, and modifying the resources associated with a *DtComponentSystem*.

When you use the *DtObjectResourceManager* to look up resources that are owned by a *DtSimResourceManager*, the name of the system must be prefixed to the resource name, separated by a period. For example, to get the fuel resource for a ground vehicle, the string would be `movement.fuel`.

The Resource Manager tracks all consumable resources, which are instances of *DtSimResources* (*simRsrc.h*). Resources include such items as fuel, ammunition, water, and whatever else is modeled. The resources are known by a *DtString* name, so it is easy to support almost any kind of resource you want. The Resource Manager keeps track of the quantity of each resource the entity has, as well as the total amount of each type of resource the entity can hold. During initialization, resource capacity and starting values specified in any *DtVrfObjectParameters* are copied to the *DtSimResourceManager*.

The Resource Manager supports subresources. Resources can be grouped to form hierarchical resource lists. For example, an entity could have a resource named **ammunition**, which has **main-gun-ammunition** and **machine-gun-ammunition** as subresources. The **main-gun-ammunition** resource might have **APDS** and **HEAT** as further subresources. In this scheme, a plan or any piece of code could test the amount (or percentage amount) of available **ammunition**, **main-gun-ammunition**, **APDS**, or **HEAT**. A user could also give 50 rounds of **main-gun-ammunition** to the entity, which would automatically be distributed between the **APDS** and **HEAT** resources.

The *DtSimResourceManager* maintains a hashlist of all resource names under its control. Given the resources described in the previous paragraph, the Resource Manager would maintain separate hashlist entries for **ammunition**, **main-gun-ammunition**, **machine-gun-ammunition**, **APDS**, and **HEAT**. Any of these resources could be looked up by name using the member function lookupResource(). Resources can be added to a *DtSimResourceManager* using addResource() and removeResource().

The Resource Manager interface allows a *DtSimResource* to be:

- Incremented
- Decremented
- Reset (set to 0)
- Filled (set to default maximum).

The Resource Manager can report the current amount as a real number or as a percentage. If a resource type that has subresources is filled, the Resource Manager distributes the total resource allotment among the subresources. The individual subresources can be filled, changed, incremented, or decremented, within the limits of the total amount of the parent resource allowed.

The initial resource levels and the organization of resources into subresources are specified as parameters of the entity. The data is saved and restored as part of an entity's state in the order of battle file. When you load a scenario, each entity's resource levels are restored to the state they were in when the scenario was saved.

For more information about the Resource Manager, please see "Modeling Resource Consumption," on page 5-18.

4.10.1. Managing Resources

All resources derive from the abstract class *DtSimResource*. Since resources can be read from parameter files, *DtSimResource* provides the standard factory mechanisms, such as:

- A pointer/accessor to a *DtSimResourceFactory* (*rsrcFactory.h*)
- An addResourceCreatorFcn() member function to register factory callbacks for creating new types of resources
- A type() member function that returns the string type of the resource that resource creation callbacks are indexed by
- A createResource(), which can allocate and initialize a resource based on the contents of an MTL stream.

DtSimResource also provides the basic framework that derived classes can use to create and manage hierarchical resources. These include:

- A *DtSimResourceList* (*rsrcList.h*), which can be used to maintain a list of *DtSimResource* subresources
- A pointer to the *DtSimResourceManager* that is managing this resource, if any
- The member functions addChildResource() and removeChildResource()
- A pointer to the resource's parent resource, which is null if the resource is not a subresource.

DtSimResource declares several pure virtual functions that derived classes must implement to provide the standard functionality. Whenever the amount of a resource is changed, that resource is responsible for updating its subresources, if any, its parent resource, or both. Call the distributeAmountToChildren() member function to update subresources, if any. If the resource's parent pointer is non-null, call the parent's updateAmountFromChildren().

All resources can have a capacity, or 'full-amount', specified. If this capacity is changed, a parent resource and any subresources should be updated using the member functions distributeFullAmountToChildren() and updateFullAmountFromChildren().

A resource can be set to have a percentage of its full capacity by calling setPercentageFull() with a value between 0.0 and 1.0. You can find the ratio of the current value over the capacity by calling percentageFull(). These two member functions must also be implemented by derived classes.

For an example of modeling resources, please see "Modeling Resource Consumption," on page 5-18.

4.11. Embarkation

Embarkation is the ability to place an entity on or in another entity. Embarkation includes the ability to attach objects, such as points or routes, to entities. Support for embarkation means that you cannot assume that an entity is attached to the terrain. This has implications for how you develop entity models. This section describes the major classes that implement embarkation. Please see the class documentation for details about the classes that handle embarkation tasks and the embarkation model.

The embarkation controller (*DtEmbarkationController*) (*embarkationController.h*) handles embarkation of an object onto a parent and the disembarkation from that parent. The *DtEmbarkationController* is a state machine that handles the embarkation of an object from its current location to an embarked location.

DtAggregateEmbarkationController (*aggregateEmbarkationController.h*) passes tasks and set data requests to currently taskable subordinates (sets are passed down to all subordinates). The controller also adds the checkEmbarkationState() member function, which is called every tick regardless of whether or not the aggregate is tasked. The pseudo aggregate is considered embarked if all available taskable subordinates are embarked. The aggregate is then embarked on the same object as the current leader of the aggregate. If any subordinate disembarks, the aggregate also disembarks.

DtOccupancyDirectorController allocates reserved embarkation slots and determines which (if any) objects can be embarked on an object. This controller responds to messages of type DtIfRequestEmbarkationInformation. When the request is received, the controller looks to see if there are any slots available, and, if so, reserves it (if requested) and then returns a DtIfEmbarkationInformation structure with the reservation information. This controller also provides disembarkation information and responds to disembarkation requests. If an entity cannot embark either because the object does not support the requesting object type or because the slots are full, an information return structure is sent with the DtEmbarkCannotEmbark result status.

4.11.1. How Embarkation Affects Entity Models

Because entities can be embarked on other entities and can be tasked to perform operations on those entities, if you model dynamics of an entity, you must consider how your models will take advantage of this behavior.

You cannot assume that an entity is solely attached to the terrain. An entity can be attached to any other model (except for environmental objects) in the simulation. When an entity is embarked, you need to know the following information about it:

- Terrain intersections
- Velocity relative to the host
- Acceleration relative to the host.

To support embarkation, your code must do the following:

- Terrain Intersections:
  In your models, when you need to query for information about the surface or structure your entity is currently positioned on, do so through the entity's attached terrain member (myEntity->terrainAttachedTo()). This returns a pointer to a *DtAttachedTerrain* object, which is an abstraction of the surface or structure an entity is currently attached to. The underlying structure represented by this object may be a terrain database or another entity. The *DtAttachedTerrain* object provides an interface for determining terrain height and chord intersections.
- Velocity relative to the host:
  To handle velocity relative to the host, use the member function calls, relativeVelocity() and setRelativeVelocity(). The return and set values are in local (database) coordinates, but if the entity is attached, the return value of relativeVelocity() is the velocity of the entity only. If you call localVelocity(), the return value is the sum of the velocity of the entity and any objects it might be attached to. You must also call setRelativeVelocity() to preserve the notion that this velocity is for the entity only. If the entity is not attached, these calls pass through to localKinematicState().velocity() and localKinematicState().setVelocity().
- Acceleration relative to the host is handled that same way as velocity relative to the host, except that the member functions are relativeAcceleration() and setRelativeAcceleration().

4.11.2. Attaching Environmental Objects to Entities

You can attach environmental objects (routes, waypoints, areas, and so on) to entities. For example, a route can be placed on an aircraft carrier for a DI to walk along or a plane to taxi along. To support this, the createOverlayObject() member function has an optional parameter (initially the empty string) that can contain the name of the object to attach to. The sendVrf-OverlayObjectModifyMsg() member function has the parameters **attachTo** and **keepExistingAttachment**. This is so the create function can assign the attachment and future modifications can keep any attachment that has been created (or, can change it).

4.12. Entity Communication

Entities communicate with each other and with the VR-Forces GUI by sending messages. All inter-entity messages are sent using one of the two message passing systems in VR-Forces – the radio message system, or the simulation internal message system.

4.12.1. The VR-Forces Radio Message System

Messages that are sent between entities to represent radio traffic are sent though the radio message system. Only other entities that have radios configured on them are able to receive radio messages. An entity can have more than one radio and messages can be sent and received on any of the radios on the entity. Radios are configured in conjunction with a communication model, which affects whether or not a given radio message can be delivered.

4.12.2. The VR-Forces Simulation Internal Message System

Messages that are used for internal simulation purposes and which do not simulate real world messages, are sent though the simulation internal message system. All *DtVrfObjects* can send and receive simulation internal messages. Simulation internal messages always reach their destination, and do not pass though any connectivity checks.

For more information about radios, please see Chapter 9, *Entity Communication*.

4.13. The Hostility Manager

The Hostility Manager (*DtHostilityManager*, in *hostilityManager.h*) is a single, centralized manager owned by the Simulation Manager. It keeps track of hostility relationships between different forces. Each force can be hostile to any of the other forces. However, the relationships are not reciprocal. For example, Force A could be hostile toward Force B without Force B being hostile to Force A.

Force hostility is used for entity detection and target selection. By default, sensors only detect targets toward which their parent entity is hostile. Similary, only hostile entities are selected as targets.

By default, Force Friendly and Force Opposing are mutually hostile to each other. You can change the default hostility relationships by editing the *forceHostilty.mtl* file for the simulation model set you are using (for example, *./data/simulationModelSets/default/forceHostilty.mtl*). You can change force hostility during runtime. The settings are saved as part of the scenario. The hostility loaded with a scenario overrides the hostility loaded by the simulation model set's hostility file.

You can access the Hostility Manager from the Simulation Manager using the hostilityManager() member function. The following example sets Force Friendly to be hostile to Force Neutral:

```
simManager()->hostilityManager()->setHostility(DtForceFriendly,
  DtForceNeutral, true);
```

4.14. DI-Guy Integration

VR-Forces has built-in support for DI-Guy[1] models. It publishes DI-Guy animations and appearances, which 3D visualization applications such as the VR-Forces front-end and VR-Vantage Stealth can use to control lifeform models. The DI-Guy representation of the entity is stored in its state repository in the container class *DtRwDIGuyCharacterInfo*. This holds DI-Character, the current appearance, and the animation of the entity.

The *DtDiGuyController* class handles changes to the DI-Guy representation of the character and keeps the DI-Guy animation the entity is currently performing synchronized with the actual simulation of the entity. It responds to the Set DI-Guy Appearance set data request and updates the DI-Guy appearance in the entity's state repository. It responds to any DI-Guy Animation tasks and updates the entity's DI-Guy animation.

1. DI-Guy is a product of Boston Dynamics, Inc.

It maintains the synchronization between the DI-Guy animation that is being performed visually and where the enity is in the simulation by reading the data file, *character_digest.xml*. This file is produced by the DI-Guy application. It lists:

- The appearances available to DI-Guy characters
- The animations available
- For each animation, it specifies:
  - The time it takes to perform one cycle of the animation
  - the distance it covers
  - Posture changes it makes
  - Any directional changes it makes.

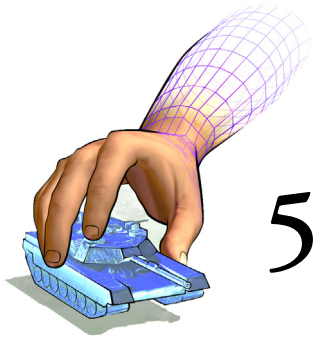
The *./appData/settings/vrfSim/character_animations_table.mtl* configuration file clarifies and sorts the information coming from *./appData/settings/vrfSim/character_digest.xml*. You can edit this file to set a user-friendly display name for the animation and determine which animations are taskable and which are intrinsic.

If you set an animation to intrinsic that is not intrinsic in the original *character_animations_table.mtl* file, you must extend the *DtDIGuyController* to handle that animation. This is because VR-Forces has to know how to synchronize the animation with the entity.

The *DtDiGuyCharacterDataMgr* class reads the configuration files when VR-Forces starts. It holds a list of DtDiGuyCharacterData containers that represent a list of all of the available DI-Guy characters in VR-Forces. Each DtDiGuyCharacterData container holds all the information about a particular DI-Guy character: the character type, all its possible appearances, all its available taskable animations and all its available intrisic animations.

When an entity is not doing a tasked animation, the *DtDIGuyController* uses the entity's current posture, speed, and weapon state to determine the best default intrinsic animation. For example if an entity is in a prone position and moving toward a waypoint, the controller queries the available intrinsic animations in the character data for an animation that is in a prone position and has a speed close to the speed of the entity. If the entity has no prone animations than it gives up and the entity finds the closest animation for its default posture.

By default, *character_animations.mtl* and *character_digest.xml* are in *./appData/settings/vrfSim*. You can specify a different location in *vrfSim.mtl* file.



5

Component Concepts

This chapter describes the high level working of components. Subsequent chapters explain how to create the individual component types. It assumes that you are familiar with the preceding chapters.

5.1. Introduction

As a group, the classes that model physics and behavior are called components. This chapter is a high-level description of entity components (sensors, actuators, and controllers). We will start by talking about the individual classes that make up the pieces of a model, the *DtSimComponents*. Next, we discuss how they fit into the component architecture, that is, how they are created, connected, and maintained by the *DtSimComponentManager*. The two chapters after this one explain how to create and modify components.

The class documentation contains additional details about components and component systems. The class documentation includes all of the examples referenced in this chapter.

5.2. Components

Entity behaviors in VR-Forces are implemented using components. Components are the building blocks that make up the sensor-controller-actuator architecture. VR-Forces supports three types of components:

- Sensors – create models of the external simulated environment
- Controllers – model behaviors that provide control inputs to actuators
- Actuators – implement the physical model.

The sensor-controller-actuator architecture is similar to that used in robotics. (For details, please see "Components and The Component Manager," on page 4-5.)

5.2.1. The DtSimComponent Class

All components in VR-Forces inherit *DtSimComponent* (*simCmpnt.h*). *DtSimComponent* provides the basic infrastructure of the component architecture. Sensors are derived from *DtSimComponent*, controllers are derived from *DtControllerComponent* (*cntrlCmpnt.h*), and actuators are derived from *DtActuatorComponent* (*actCmpnt.h*).

Since components are created by the factory system, *DtSimComponent* provides the usual factory-specific member functions as described in "VR-Forces Factories," on page 2-10. The constructor takes a `const DtString name, DtVrfObject* owner, DtSimManager* simManager, DtComponentDescriptor* desc`, and `DtReaderWriterRegistry* parentRegistry` as arguments. (Components are not currently read from or written to files -– the registry pointer is included for future use.) The name of the component must be unique within the entity that it is part of. Component descriptors are described in "Component Parameters and Component Descriptors," on page 5-4.

The type() member function of a component must be overridden if you create a new kind of component, reporting the *DtString* type of the component to be used by the component factory. The type string must be unique across all of the known types of components. The type strings used by VR-Forces components are defined in *compTypes.h*.

5.2.2. Component Parameters and Component Descriptors

Component parameters, including the name and type of component, are normally specified as part of the component descriptor that defines the component, a *DtComponentDescriptor* (*compDesc.h*). Component descriptors contain the readable-writable parameters used to create and initialize the component in the object parameter database. Component descriptors are described in "Component Descriptors," on page 5-19.

5.2.3. Inter-Component Communication

Components communicate with each other through ports. There are two types of ports: input ports (*DtInputPort*, in *inputPort.h*) and output ports (*DtOutputPort*, in *outputPort.h*). Ports can be used to transmit such data as a Boolean on/off switch, a list of target object IDs, or a range of floating point values. A component that produces data (such as an automotive controller that produces throttle values) will have one or more output ports. A component that consumes data (such as an automotive actuator that accepts throttle input) will have one or more input ports. For details, please see "Ports and Port Groups," on page 5-20.

5.2.4. Components and State Repositories

Components have a close relationship with an entity's state repository (*DtVrfObjectStateRepository*). Controllers often need to determine an entity's current state information. Actuators may also need to examine an entity's state and, typically, update it. When a component is created, the setEntity() member function provides it with a pointer to the entity that it is a part of. *DtSimComponent* caches that pointer, as well as a pointer to the entity's local state repository, for convenience. If a component is used for a particular type of derived entity class and needs access to a derived entity or derived entity's local state, these pointers can be cast to the appropriate type. Many of the VR-Forces derived classes override setEntity() to set a derived entity class pointer, so that this cast is not necessary every time the data is accessed.

5.2.5. The DtSimComponent::tick() Function

Components have a tick() function that gets called each frame, or optionally, at some interval specified by the user. The DtSimComponent::tick() function does not do anything. Derived components implement the tick() function in various ways, depending on their purpose. Controller classes can implement the tick() function to generate new control inputs each frame. Actuator components use the tick() function to update the entity's state. A component's tick() function is called by the *DtSimComponentManager*.

5.2.6. Sensors

Sensors are used to model the external simulated world. The *DtObjectSensor* (*objectSensor.h*) is the base class for all sensors. The *DtSignatureObjectSensor* (*signatureObjectSensor.h*) class is used to detect all objects that have a signature configured. The *DtRadarObjectSensor* (*radarObjectSensor.h*) is a special object sensor that also acts as an emitter. For details, please see "Sensors," on page 7-2.

5.2.7. Controller Components

Controller components typically provide control inputs to actuator components. They may accept sensor input and use this information to carry out their functions. Controller components must inherit *DtControllerComponent* (*cntrlCmpnt.h*) (which is derived from *DtSimComponent*). The *DtControllerComponent* overrides setEntity() so that it can cache a pointer to the entity's radio, just as the *DtActuatorComponent* does. Access to the radio is necessary for those controllers that need to respond to tasks and set data requests, send tasks to other entities (in the case of aggregates), or generate reports.

Some controller components implement tasks. The Task Manager calls the controller component list, which then calls the task functions.

5.2.8. Actuators

Actuators provide the physical model of an entity. All actuators in VR-Forces are derived from *DtActuatorComponent* (*actCmpnt.h*), which is derived from *DtSimComponent*. Since actuators may want to send radio messages (for example, a report), this class overrides setEntity() so that it can cache a pointer to the entity's radio. The *DtActuatorComponent* also provides an accessor to this radio pointer.

Many actuators receive control inputs from one or more controllers. They can use this control information, along with the entity's current state, to generate new state values for the entity. For example, in the DtAutomotiveActuatorComponent::tick() function, it checks the throttle, steering, and braking inputs, computes the new location, orientation, velocity, and acceleration values, and then sets them in the entity's state repository.

5.2.9. Components Provided with VR-Forces

VR-Forces provides many component classes for modeling entity behaviors. They provide you with a starting point when you want to extend the toolkit. Please see the class documentation for high level descriptions of components and component systems. The class documentation also contains an index of entities and the components they use. You may also want to use the OPD Editor or Entity Editor as a convenient way to explore the components and systems used on individual entity types.

5.3. Component Systems

A collection of related *DtSimComponents* can be organized into a *DtComponentSystem* (*compSystem.h*). A *DtComponentSystem* is a class that may contain zero or more sensors, controllers, and actuators, and systems. Within a *DtComponentSystem*, components and subsystems are ticked the same way as components that belong directly to the *DtComponentManager*. Each *DtComponentSystem* has a *DtSimResourceManager* that maintains any resources associated with the system. The *DtComponentManager* has a 'base' *DtComponentSystem* that holds all *DtSimComponents* and *DtComponentSystems* for the entity.

Systems are used to represent some logical subsystem of an entity, such as a weapon or its movement capabilities. To see which systems have been defined for use in VR-Forces, use the Entity Editor and OPD Editor, or go to the *./data/simulationModelSets/default/vrfSim/systems* directory to examine the system definition files (*.sysdef*) directly.

You can create an entity without any explicit systems defined by configuring the entity with sensors, controllers, and actuators directly. The 'base' *DtComponentSystem* will be created implicitly for the components when the entity is created. However, you can create larger, reusable 'building blocks' if you organize components into systems.

Only components contained in the same system can share process state repositories.

5.4. The Component Manager

The Component Manager, *DtSimComponentManager* (*compMgr.h*), is a top-level class owned by a *DtVrfObject* that is responsible for managing the object's *DtSimComponent* instances. It creates, connects, and maintains the lists of *DtSimComponents* for an entity. Entities that are configured with a Component Manager have an instance of a *DtSimComponentManager*. (For information about configuring an object with a Component Manager, please see Section 7.1.4, "Local and Remote Subcomponents" in *VR-Forces Configuration Guide*.) Only local entities have a Component Manager.

The *DtComponentManager* has-a *DtComponentSystem* data member that holds all *DtComponentSystem* and *DtSimComponent* instances for the entity. It is implicity created (so it will be present even if no systems are explicitly configured as part of an entity in its parameter database). This base system has-a *DtComponentNetwork* member that performs much of the actual management work on behalf of the *DtComponentManager*.

Immediately after creating and initializing the Component Manager in DtVrfObject::init(), the object invokes DtVrfComponentManager::addComponents(). This function:

- Creates the lists of sensors, controllers, and actuators from the entity's parameters member. The entity's parameters member contains the sensor, controller, and actuator descriptor lists that were specified in the object parameter database.
- Connects the components to one another through ports and port groups based on information in the connection descriptor list in the entity's parameter member. The connection descriptor list contains the names of ports and port groups to be connected.

The Component Manager ticks the system, sensor, controller, and actuator lists, which in turn iterate through their list of systems and components and call their tick functions. The component lists are ticked in the order: system, sensor, controller, actuator. The order in which individual components are ticked is the same order as the list of corresponding component descriptors in the entity's parameters member. The tick order of components and component lists can affect the priority of components. (For details, please see "Setting the Priority of Components," on page 5-16.)

5.4.1. Configuring a Component Manager

The *DtSimComponentManager* is owned by a *DtVrfObject*. The Component Manager is created as part of the *DtVrfObject* via the standard factory mechanism. (For details, please see "The DtSimComponent Class," on page 5-4.) If you want to simulate an entity using the component architecture, you must specify that a Component Manager be created as part of the entity.

To configure a *DtVrfObject* to be created with a Component Manager, specify a Component Manager in the object's entry in the object parameter database, for example:

```
(ground-vehicle-parameters
  (parameter-type "ground-vehicle-param")
  ...
  (local-objects
    (state-repository  "ground-entity-state-repository")
    (state-repository-min-tick-period  -1.000000)
    (state-repository-min-tick-period-variance  -1.000000)
    (net-interface  "individual-local-entity-net-interface")
    (net-interface-min-tick-period  -1.000000)
    (net-interface-min-tick-period-variance  -1.000000)
    (task-manager  "local-task-manager")
    (component-manager  "component-manager")
    (plan-manager  "vrfobject-plan-manager")
  )
  ...
)
```

Component Managers should only be created for locally simulated objects. Do not specify a **component-manager** in the remote-objects list.

5.4.2. Looking Up a Component

You can look up a component by name in the Component Manager. Since it returns a pointer to the base class *DtSimComponent*, the caller is responsible for knowing or determining the actual type of the component. Components have a type() member function that returns the type of a component (the same string used by the component factory to create it). For example, to look up an automotive actuator component in an M1A2 entity (named "powertrain" in the M1A2 object parameter database entry):

```
DtSimComponent* actuator = vrfObject->componentManager()->
  lookupComponent("powertrain");

// Make sure the component that was found is of the type we expect
if(actuator && actuator->type() == DtAutomotiveActuatorType)
{
   . . .
}
```

If the entity has been configured with systems, lookupComponent() will search recursively for the component. It searches first through the entity's immediate components and then inside its systems. It returns the first component that matches the given name.

You can also search for a *DtComponentSystem* by name, by calling lookupComponentSystem() in the *DtComponentManager*. The ability to look up a component system by name can be useful when there are multiple *DtSimComponents* with the same name, but which are in different *DtComponentSystems*. For example, consider a situation in which an entity has multiple weapon systems. To find the particular component you want, look up the weapon system you want first; then look up the *DtSimComponent* in that *DtComponentSystem* directly, by calling its lookupComponent() member.

5.4.3. Creating Components

An entity's *DtComponentSystems* and *DtSimComponents* are created in the DtSimComponentManager::init() function. The Component Manager creates the entity's systems and components from the entity's component descriptor lists. (For details, please see "Component Descriptors," on page 5-19.)

Sensor and actuator components for an entity are placed on *DtSimComponentLists*. Components are indexed on the lists by component name.

If an entity is not configured with any systems, the name of the component must be unique within the entity. If the entity has been configured with one or more systems, then the name of the component must be unique within its system. The name is normally specified as part of the component descriptor that defines the component. Component descriptors are described in "Component Descriptors," on page 5-19.

Controller components are placed on a *DtControllerComponentList* (*ctlrCompList.h*), which is derived from *DtSimComponentList*. The *DtControllerComponent* list adds member functions that call a corresponding member function for each controller component on the list. These include taskNotify() and skipTask(). These member functions are described in "Controller Components," on page 5-5.

They are typically invoked by the Task Manager, which manages the overall execution of tasks by an entity.

Components are created and added to the Component Manager in the order that they appear in the object parameter database. The order in which components are created and added to the lists is important, because it can affect the priority of one component over another when both provide inputs to the same target component. Please see "Setting the Priority of Components," on page 5-16 for more details.

5.4.4. Connecting Components

After the system and components are created, they are connected in the Component Manager's createConnectionsFromList() function. It takes the list of connections from the entity's local parameter entry and walks through the list, looking up the ports and port groups to be connected to one another. Each item in the connection list contains one of the following:

- Names of a <component or system>:output port pair and a <component or system>:input port pair to connect
- Names of a <component or system>:output port group pair and a <component or system>:input port group pair to connect.

The name of each component or system is the name in the descriptor. The names of the port and port groups are hard-coded in each component. (Please see the component's header file to find out the names of the port and port groups.)

! For each output port and input port that it connects, the Component Manager sets the priority of the connection to be the priority of the output port's component. Priority is represented as an integer value starting at 1. Increasing values represent decreasing priority. The priority of the connection affects how multiple inputs to the same port are arbitrated. For details, please see "Setting the Priority of Components," on page 5-16.

For example, in the following excerpt from the *ground-tracked.sysdef* system defintion (a component system used by tracked ground vehicles, such as the M1A2), the move-to controller's automotive-control output port group connects to the powertrain's automotive-control input port group.

```
(tracked-movement-system
  ...
  (controllers
    ...
    (move-to
      (component-descriptor-type "ground-move-to-descriptor")
      (component-type "ground-auto-move-to-controller")
    )
    ...
  )
  (actuators
    (powertrain
      (component-descriptor-type "automotive-component-descriptor")
      (component-type "automotive-actuator")
    )
    ...
  )
  (connections
    ...
    (connect move-to:automotive-control powertrain:automotive-control)
    ...
  ) );; end connections
```

You can connect systems to other components or other systems in a similar manner. Ideally, systems should be able to be treated as 'black boxes'. Connections between components defined within a system can be thought of as 'private' connections, while connections between the system and external components or systems can be thought of as 'public' connections. You create a 'public' connection endpoint for a system by configuring a mapping between the system and a given component:port or component:port-group. This enables you to ignore the complexity of the components and connections contained within the system and work with a typically much smaller set of of public connections. The mapping between a system connection endpoint and an actual component connection endpoint are resolved at runtime, when an entity's components have been instantiated and are being connected.

For example, consider one of the weapon systems defined in VR-Forces, the 30 mm Cannon System (*30mm-gun.sysdef*).

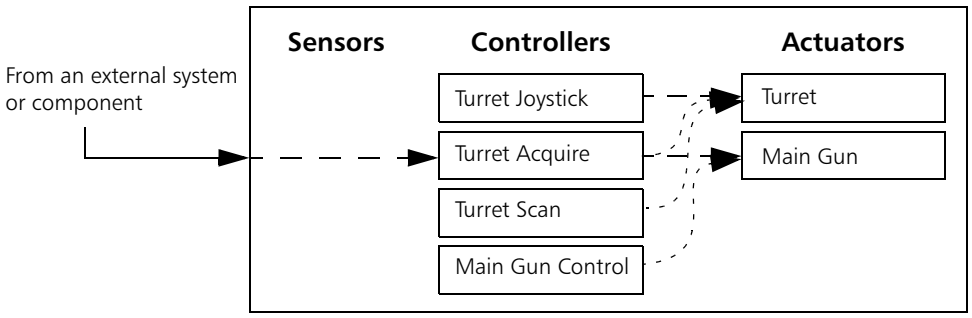


Figure 5-1. 30 mm cannon system diagram

In Figure 5-1, connections internal to the system are represented with dotted lines, while the sole external connection is represented as a solid line.

The connections for this system are defined as follows:

```
(connections
  (connect system:target-list turret-acquire:target-list)
  (connect main-gun-control-joy:gun-control main-gun-joy:gun-control)
  (connect turret-joystick:slew turret:slew)
  (connect turret-acquire:slew turret:slew)
  (connect turret-acquire:line-of-sight
    main-gun-control-joy:line-of-sight)
  (connect turret-scan:slew turret:slew)
)
```

The first line in the block of connections creates the mapping between an external system connection endpoint (system:target-list) and an internal component connection endpoint (turret-acquire:target-list). The name 'system' is always used to represent the system when creating this type of mapping.

When this system is used in the context of an entity, the only connection endpoint you need to be aware of is the 'public' endpoint system:target-list. For example, the BMP2 entity (*BMP2_1_1_1_222_2_2_1_-1.ope*) uses this particular weapon system. In the second line of the connection list, there is a single connection between the target-selection controller and the weapon system. The system endpoint is identified by the name of this system (in this case, weapon).

```
(systems
  …
  (weapon
    …
    (system-definition
      (filename  "$(system-dir)\weapons/30mm-gun.sysdef")
    )
  )
  …
)
…
(controllers
  …
  (target-selection
    …
)
…
(connections
  (connect target-selection:object-types-to-detect sensor:object
-types-to-detect)
  (connect target-selection:weapon weapon:target-list)
  (connect sensor:detected-objects contact-fusion:detected-objects)
  (connect other-2:detected-objects contact-fusion:detected-objects)
)
```

In the this example there is just one external system connection endpoint. However, there may be cases in which there are no external connection endpoints defined (the system is completely self-contained), or there may be multiple endpoints defined.

5.4.5. Ticking Components

By default, each component of every entity is ticked each simulation frame. VR-Forces allows you to specify a minimum tick period for a component, or all the components of a given entity type in the object parameter database. This minimum tick period can also be changed through the API. The minimum tick period is used to determine the next tick time that the component may be ticked. The time can be expressed as either real time (wall clock time) or simulation time, depending on the value of the **tick-period-uses-real-time** parameter.

The **tick-period-uses-real-time** parameter specifies whether or not the **tick-period** parameter should use wall-clock time or simulation time. If your simulation requires repeatability, set **tick-period-uses-real-time** to False for all components. For more information about repeatability, please see "Repeatability", in Chapter 3, *VR-Forces Simulation Concepts* in *VR-Forces Users Guide*.

You can also specify a minimum tick period variance, which specifies the range that the next tick time may vary by (this helps to distribute the individual component's ticks more evenly across different simulation frames).

Each simulation frame, the Component Manager is ticked. It in turn ticks the component systems list, the sensor list, the controller component list, and then the actuator component list. Each component list calls DtSimComponent::timedTick() for each component on that list, in the order that the components were specified in the object parameter database (or otherwise placed on the list). The timedTick() member function checks to see if the current time is greater than or equal to the next tick time for that component, which can be accessed via the nextTickTime() member function. If it is, then dT (delta time) is calculated for that component and the component's tick() member function is called.

Following the call to tick(), a new next tick time is calculated. To calculate the new next tick time, VR-Forces determines a random value between +/- **minTickPeriodVariance**/2, then adds it to the current time + the minimum tick period. The next tick time is never less that the current time.

You can set the next tick time for a component by calling DtSimComponent::setNextTickTime(). Use a next tick time equal to the current time to guarantee that the component will be ticked on the next simulation frame. Setting the minimum tick period and minimum tick period variance to 0.0 causes the component to be ticked every simulation frame (the default, unless modified by the object parameter database).

Because components are not necessarily ticked every simulation frame, do not use the value returned by DtExerciseClock::dT() in components, because this is not necessarily the time that has passed since the last call to a particular component. Use the DtSimComponent::dT() member function instead.

5.4.6. Setting the Priority of Components

When multiple components can provide data to the same receiving component, they connect to the component based on their priority.

When a component that wants to provide data (by sending data through an output port owned by the component) to another component (receiving data through an input port owned by the component) is ticked, it calls its output port's attemptToActivate() member function. The attempt to activate is made using the priority assigned to the component with the output port. The attempt to connect will succeed if one of the following conditions is true:

- There is no other output port already actively connected to the input port
- This particular output port is already actively connected to the input port
- There is another output port actively connected to the input port, but the output port attempting to connect has a higher priority than the output port currently connected.

The connection's priority was established when the components were connected. (For details, please see "Connecting Components," on page 5-12.) Each connection between an output port and an input port is assigned the same priority as the priority of the output port's component. Component priority is set in the context of a given *DtComponentSystem*. For components that are not explicitly configured as part of a system, priority is set in the context of the entity's implicit base *DtComponentSystem*.

If the call to the output port's attemptToActivate() returns `true`, the component typically computes its new data values (if necessary) and puts the resulting values on the output port. When the rest of the components get ticked, because they are ticked in priority order, highest to lowest, no other component will be able to establish an active connection to the input port this tick.

If the call to the output port's attemptToActivate() returns `false`, then there is not a valid connection to the input port (another component with a higher priority is probably already connected.) Most components do not bother to run their algorithms if they cannot establish an active connection between their output port and the input port they need to furnish the results to (although this is up to the implementer of the component in question).

The receiving component gets ticked. In the receiving component's tick() member function, it checks for a valid connection to the relevant input port by calling the input port's activeConnection() member. If that returns a valid pointer (indicating an output port is currently connected to the input port), it retrieves the value on the input port and uses the data in computations for this tick.

When a component providing data through an output port wants to release an active connection to another component's input port, it calls the output port's deactivate() member. This allows any other component to successfully connect to the input port. This call is often made by task controllers when they have completed their tasks. For example, once the move-to controller has reached its goal, it no longer needs to provide data to the automotive actuator input ports. At that point it deactivates its output ports.

A component providing data may also choose to remain connected, but temporarily reduce itself to the lowest priority (which allows any other interested component to connect). This is done by calling the output port's allowDeactivation() member function. The connection to the input port remains in place, but the output port's connection is assigned a special priority of -1 (lower than any component's assigned priority). In this way a component can continue to provide data inputs to another component until some third party component determines it wants to take over the connection and start providing inputs.

5.5. The Aggregate Multi-resolution Model

The aggregate model in VR-Forces is a multi-resolution model. Aggregates are configured with multiple sets of components and systems that have overlapping functionality. One set models the behavior of the aggregate while in a disaggregated state, and the other set models the behavior in the aggregated state. There is a special component, *DtAggregateModelResolutionController* (*aggregateModelResolutionController.h*) that manages the transition between aggregated states. It is responsible for registering for and handling *DtSetAggregateState* set data requests, which trigger the transition between models.

When the *DtAggregateModelResolutionController* receives a set data request indicating a transition to the aggregated state, is performs the following actions:

1. It disables all of the components associated with the disaggregated state.
2. It gathers and stores data about its current subordinates and their resources, and deletes the subordinates from the simulation. The execution of this activity is done through the *DtAggregator* (*aggregator.h*) helper class.
3. It enables all of the components associated with the aggregated state.

When the *DtAggregateModelResolutionController* receives a set data request indicating a transition to the disaggregated state, is performs the following actions:

1. It disables all of the components associated with the aggregated state.
2. It examines its stored data about its current subordinates and their resources, and uses this information to instantiate and initialize new subordinate entities, aggregates, or both. The activity is carried out by the *DtDisaggregator* (*disaggregator.h*) helper class.
3. It enables all of the components associated with the disaggregated state.

5.6. Modeling Resource Consumption

Some models need to track consumption of resources, such as fuel or munitions. Each *DtSimComponentSystem* maintains its own set of resources in a *DtSimResourceManager*. Each component in a *DtComponentSystem* has a reference to this manager. A component that uses resources must look up the actual resource by name through its resourceManager() member. If no *DtComponentSystem* has been explicitly configured in an entity's parameters, the resourceManager() function returns a pointer to the implicitly created base *DtComponentSystem's* resource manager.

Once you have a pointer to the resource, you can determine its current level, and modify it as needed. For example, the *DtAutomotiveActuatorComponent* has a useFuel() member function, which is called in its tick() member function. It decrements the fuel in the entity by a given amount, as follows:

```
void DtAutomotiveActuatorComponent::useFuel(double fuelUsed)
{
  DtSimResource *resource = resourceManager().lookupResource("fuel");

  if (resource)
  {
    DtString resourceType = resource->type();
    if (resourceType == DtRealResourceType)
    {
      DtRealResource * realResource = (DtRealResource *)resource;

      if (realResource->amount() <= fuelUsed)
      {
        DtWarn("%s out of fuel.\n", entity()->objectName().string());
        realResource->setAmount(0.);
      }
      else
      {
        realResource->setAmount(realResource->amount() - fuelUsed);
      }
    }
    else if (resourceType == DtIntegerResourceType)
    {
      DtIntegerResource* intResource =
        dynamic_cast<DtIntegerResource*>(resource);
      assert(intResource);
      if (intResource->amount() <= (int)fuelUsed)
      {
        DtWarn("%s out of fuel.\n", entity()->objectName ().string());
        intResource->setAmount(0);
      }
      else
      {
        intResource->setAmount(
          static_cast<int> (intResource->amount() - fuelUsed));
      }
    }
  }
}
```

The actual types and amounts of each resource are configured in the object parameter database. (For details, please see "The Resource Manager," on page 4-30.)

5.7. Fire and Detonation Processing

Processing of damage and fire interactions is managed by the *DtDetonationManager* (*detonationManager.h*)and the *DtFireManager* (*fireManager.h*). They notify the appropriate entity or entities when an interaction involving them occurs.

The components work together to minimize processing of interactions. The *DtFireManager* registers for both fire and detonation interactions. The *DtDetonationManager* registers for detonation interactions. The *DtDamageAdjudicationActuator* (*damageAdjudicationActuator.h*) and *DtUnderFireDeterminationSensor* (*underFireDeterminationSensor.h*) register themselves, their entities, and their appropriate callback functions with the managers. When a detonation or fire interaction occurs, the fire and detonation managers determine which registered entities are affected, and notify only those entities of the interaction. Thus only the appropriate registered entities need to process the interaction.

5.8. Specifying Parameters for Components

It is useful to be able to specify parameter values for components in the object parameter database. VR-Forces makes this possible by using component descriptors.

5.8.1. Component Descriptors

Component descriptors describe components and their parameters in the object parameter database. When an entity is instantiated, a component is instantiated for each component descriptor in its object parameter database entry. Each component has a const pointer to the *DtComponentDescriptor* from which it was created in the myComponentDescriptor member of *DtSimComponent*, the base class for all derived components. If a derived component needs to access its corresponding descriptor, it must cast the myComponentDescriptor pointer to the correct type.

Since component descriptors are read from the object parameter database, they conform to the standard factory mechanism described in "VR-Forces Factories," on page 2-10. The type names of the component descriptors are in *compDescTypes.h*.

For more information, please see "Creating a Component Descriptor," on page 6-10.

5.9. Ports and Port Groups

Components exchange data with one another through ports. There are two types of ports: *DtOutputPorts* (*outputPort.h*) and *DtInputPorts* (*inputPort.h*). Components that provide data have one or more *DtOutputPorts* (*outputPort.h*) as data members. Components that accept data as input have one more or more *DtInputPort* (*inputPort.h*) data members. Components' output ports are connected to input ports by the Component Manager at runtime, when an entity is created. Each tick, data can flow from one component to another, through those ports that are connected.

For example, the *DtGroundAutoControllerComponent* (and its subclasses) has a *DtAnalogOutputPort* throttle port member, on which it sets the throttle value it computes for ground vehicle each tick. The *DtAutomotiveActuatorComponent* has a *DtAnalogInputPort* throttle port data member, from which it gets throttle input values each tick. The Component Manager connects the ground auto controller's throttle output port with the automotive actuator's throttle input port, enabling data to effectively flow from the controller to the actuator.

5.9.1. Port Groups

Port groups are collections of ports. They are useful for creating logical groupings of related ports. Port groups can be used to simplify the code in components when you want to treat the entire set of ports in the same way. Port groups are divided into types: *DtInputPortGroup* (*inputPortGroup.h*) and *DtOutputPortGroup* (*outputPortGroup.h*). For example, the *DtAutomotiveActuatorComponent* has a *DtInputGroup* member, myAutomotiveControlPortGroup. This group contains the throttle, steering, and brake input ports. The automotive actuator expects to receive all three inputs each tick in order to work properly. It would not make sense to provide steering inputs without throttle. The fact that these ports are placed inside of a port group suggests that some relationship exists between the ports.

Port groups provide a convenient way to set up connections in the object parameter database file. They also provide a convenient way to test or set the activation state of a collection of ports. You can query and set the activation state for all ports in a group, and you can query a group to determine if any of the ports in a group are active.

Use of port groups is optional. A component that has port groups still has to create the individual ports in the group, and you can still directly manipulate them.

5.9.2. Connecting Components

The Component Manager connects components to one another through their ports, port groups, or both. It uses information in the component descriptor that specifies how the components should be connected (which output port should be connected to which input port).

The base entity parameter class has-a new connection list member. The Component Manager uses this list of connections to construct the new connections between ports and port groups. Connections are created in the createConnections() call in the Component Manager. It creates connections for each entry in the parameter connection list and sets up the connection endpoints specified in the connection list entry. It uses the "component name:port name" combination to look up the actual port endpoints, and configures the connection with this data. (It does a similar lookup for "component name:port group name".)

An Example of Port and Port Group Connections

Figure 5-2 illustrates the role of ports and port groups. *DtGroundCollisionAvoidanceController* and *DtGroundMoveToControllerComponent* are both derived from *DtGroundAutoControllerComponent*. Therefore, they both have throttle, steering, brake, and parking brake ports, and an automotive control output port group. The output port group is a container that holds pointers to the individual output ports and provides an interface to manage them collectively, rather than individually.

The *DtAutomotiveActuatorComponent* uses throttle, steering, brake, and parking brake values to model the behavior of ground vehicles. It gets these inputs from any of the *DtGroundAutoControllerComponent* subclasses. Rather than managing them individually, it creates an input port group (*DtInputPortGroup*, in (*inputPortGroup.h*)) to manage them as a group. (Conceptually, this is similar to plugging several power cords into a power strip and plugging the power strip into a single electrical outlet, rather than plugging each cord into an individual outlet. The power strip is the equivalent of the port group.)

Each simulation frame, those components whose next tick time is less than or equal to the current (clock) time are ticked in priority order, highest to lowest. If the *DtGroundCollisionAvoidanceController* is ticked and it detects a possible collision that must be avoided, it attempts to connect its output ports to the automotive actuator component's input ports. The priority of the collision avoidance controller is the priority of the individual port connections.

If it can get connected (if the input ports are not already attached to another component with a higher priority), it connects and provides its control data to the actuator. It retains control of the input ports even after the actuator has processed the control data. This prevents any other component whose priority is lower than *DtGroundCollisionAvoidanceController* (such as *DtGroundMoveToController*) from connecting to the input ports until that connection is explicitly released. This prevents lower priority components from connecting to the input ports while the collision avoidance maneuver is in progress (possibly over several simulation frames), even if *DtGroundCollisionAvoidanceController* is not being ticked every simulation frame. (Please see "Ticking Components," on page 5-15, for information about ticking components at intervals other than every simulation frame.)

When the *DtGroundCollisionAvoidanceController* determines that the collision avoidance maneuver is completed, it calls its automotive control output port group's allowDeactivation() member function. This keeps its output ports connected to the actuator's input ports, but moves it to the lowest possible priority so that another component, such as the move to controller, can connect to the component. (Its last data values are latched until some other component successfully connects to the input ports.)

In Figure 5-2, the *DtGroundMoveToController* has control of the port group.

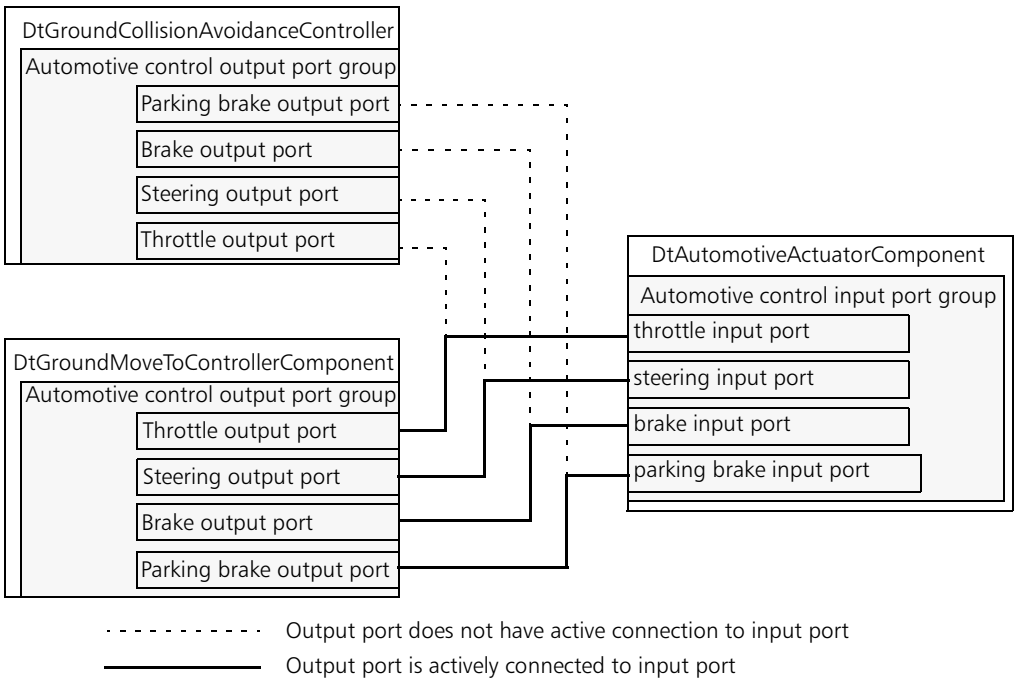


Figure 5-2. Port group

5.9.3. Types of Input and Output Ports Supported in VR-Forces

The *DtOutputPort* and *DtInputPort* are abstract base classes that provide interfaces for sending data, receiving data, and managing the connections between ports. They do not specify the kinds of data transmitted over the ports. Specific port types are represented by derived classes. There are input and output port classes for conveying integers, real values, entity identifiers, and so on. Please see the class documentation for information about the specific input and output port types available in VR-Forces.

DtSimComponents have-a list of *DtInputPorts* and *DtOutputPorts*. Derived component classes that have input port members, output port members, or both, create them in their createPorts() member function, and then add them to the appropriate list.

5.9.4. Sending Data Through an Output Port

Each tick, components that provide data through an output port typically perform some or all of the following steps:

1. Attempt to activate the connection

   The first step in sending data through an output port is to call its attemptToActivate() member function. This operation turns on any outgoing connections if the priority of those connections is sufficient. If attemptToActivate() returns false, this indicates that no *DtInputPorts* are connected to this port. In this situation no one is listening for data, so it is not necessary to calculate new values.

2. Place the data value on the port

   If the call to attemptToActivate() returns `true`, then the next step is to call setValue() to set the data on the output port. Optionally, a component can follow this with a call to the port's sendData() member function. This sends data to the receiver (the input port on the other end of the connection). Call the sendData() member function every time the data producer has provided new values to the port and wants those values to be sent.

Derived *DtOutputPort* classes typically call sendData() from within setValue(), so there is usually no need for you to call sendData() explicitly. By default, sendData() is automatically called the first time a value is placed on the port, and subsequently any time the value being set on the port changes. If you want to notify the receiver that the value has been set, even if it has not changed, then call sendData() after you have called setValue().

3. Deactivate the port (disconnect)

   When the user of an output port determines it no longer wants to produce output, it should call either its deactivate() or allowDeactivation() member functions.

   Calling deactivate() on an output port immediately severs the connection between the output port and the input port. This allows other components that have an output port connected to the same input port to potentially activate their connection.

   Calling allowDeactivation() on the output port keeps the connection active, but moves it to the lowest possible priority. When any other component attempts to provide data to the same input port, the connection from this output port is deactivated.

5.9.5. Retrieving Data from an Input Port

Each tick, components that receive data through an input port typically perform some or all of the following steps:

1. Check for an active connection:

   Components that retrieve values from ports check to see if there is an active connection to the port by calling the port's activeConnection() member function. This call returns a pointer to the currently active *DtConnection*, if one exists, NULL if not.

2. Check to see if new data is available:

   If the connection is active, the component checks to see if there is new data available on the port by calling its newData() member function.

3. Retrieve the data:

   If there is an active connection and new data available on the input port, the component retrieves the data by calling the port's value() member function.

! If the call to the port's activeConnection() member function returns false, then the data returned by the ports value() member function is undefined!

---

4. Reset the "new data" flag:

   Once data is retrieved from the port, components call the port's dataReceived() member function to clear the "new data" flag.

5.9.6. Sending Data through and Retrieving Data from Port Groups

DtOutputPortGroup has member functions that are similar to those in *DtOutputPort*: attemptToActivate(), sendData(), deactivate(), and allowDeactivation(). These calls operate on the entire set of *DtOutputPorts* in the *DtOutputPortGroup*.

DtInputPortGroup has member functions that are similar to those in *DtInputPort*: newData() and dataReceived(). *DtInputPortGroup* also provides anyPortsActive() and allPortsActive() member functions, enabling callers to check if any or all of its input ports are active.

5.9.7. Creating Ports and Port Groups in Components

If want to extend existing components and add new ports, port groups, or both to the component, you must override createPorts() or createPortGroups() as appropriate.

A component's ports are created in the createPorts() member function, and its port groups are created in createPorts().

In the createPorts() function, ports are created with a string name. This name is used in the entity's object parameter database entry to specify how an entity's components get connected (which ports get connected to which). This name must be unique within the component. Similarly, any port groups owned by a component are created in createPortGroups(). They also have a string name that must be unique within the component.

After a port is created, it must be added either to one of the component's port lists (it maintains a list of input ports and a list of output ports), or to a port group owned by the component. For example, in the *DtAutomotiveActuatorComponent*, the steering input port is created and added to its automotive control input group:

```
bool DtAutomotiveActuatorComponent::createPorts()
{
  mySteeringPort = new DtAnalogInputPort("steering");
  myAutomotiveControlPortGroup->addPort(mySteeringPort);
  . . .
}
```

After a port group is created, it must be added to the appropriate port group list owned by the component. (It maintains a list of input port groups and output port groups). For example, the *DtAutomotiveActuator* has an automotive control input port group, which it creates and adds to its list of input port groups:

```
bool DtAutomotiveActuatorComponent::createPortGroups()
{
  myAutomotiveControlPortGroup = new
    DtInputPortGroup("automotive-control");
  addInputPortGroup(myAutomotiveControlPortGroup);
  return true;
}
```

5.10. Remote Attachment

As described in Section 8.12, "Configuring VR-Forces Components on Remote Entities", in *VR-Forces Configuration Guide*, remote attachment is a mechanism that lets you attach sensors, controllers, and actuators to remote entities. Depending on your objectives, you may or may not need to write code to take advantage of this capability.

While remote entities can be configured with various types of components (sensors, controllers, actuators, and systems of components), keep in mind that there are cases in which remote attachment will not make sense. Some examples of limitations to the remote attachment mechanism:

- You would not attach a movement system to a remote entity, because that movement system would have no effect on the network state of the entity.
- Some components have built in expectations that they will be attached to a locally simulated entity. Sensors and weapon systems that depend on certain articulated parts in the host entity might not work properly in a remote environment, especially if those articulations are not being published by the entity.
- Some components may require a certain orientation of the entity to be effective, and will be unable to communicate desired orientation to the remote host entity.

5.10.1. Adding Existing Components to a Remote Entity

If you just want to equip a remote entity with an existing component or system, then you will not need to write any code. For details about how to configure remote attachment, please see Section 8.12, "Configuring VR-Forces Components on Remote Entities", in *VR-Forces Configuration Guide*.

5.10.2. Adding New or Modified Component(s) to a Remote Entity

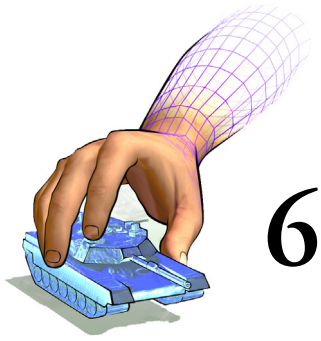
If you want to customize existing VR-Forces components for remote attachment, or you want to create new kinds of components to attach to remote entities, then you will need to write some code and configure the new components for attachment. Create or extend existing components through the toolkit as you would normally. There are numerous API examples included with VR-Forces that demonstrate how to create or extend components in VR-Forces. Once the new components are available though a back-end plug-in, configure the new components for remote attachment as described in Section 8.12, "Configuring VR-Forces Components on Remote Entities", in *VR-Forces Configuration Guide*.

5.10.3. Overriding the Remote Attachment Process

Typically, you will not need to override the remote attachment process itself. However there are a few hooks in the process which you may want to override and insert your own code, as follows:

- Upon new arrival of remote entity: DtRemoteComponentAttachmentManager::addRecentArrival()
- Determining whether or not remote attachment should take place: DtRemoteComponentAttachmentManager::addRemoteConfigurationToObject()
- Deferred initialization (actual attachment of components to remote object): DtRemoteComponentAttachmentManager::doDeferredInitialization().

To override any of the above functions, you will need to create a derived *DtRemoteComponentAttachmentManager* class. Create a derived *DtVrfObjectManager* class, and override its createRemoteComponentAttachmentManager() member function to have it create an instance of your derived class.



6

Creating Actuators and Controllers

This chapter describes how to create actuator and controller components. It assumes that you are familiar with the preceding chapters, particularly Chapter 5, *Component Concepts*.

6.1. Introduction

Component concepts are described in Chapter 5, *Component Concepts*. This chapter explains how to create actuators and controllers. It uses the following examples:

- Add Actuator – a simple example of how to replace the functionality of an existing actuator
- Stop to Shoot – a more complex example that shows how to add new controllers and actuators.

As you work your way through the examples, you will see that there is a great deal of similarity in the construction of the different components. This is because they all derive from *DtSimComponent*, and therefore, have similar basic functions that they must override. The real difference between the components is what they do in their tick() function, and this is entirely up to you. It is really a case of how you conceptualize the model you want your entities to follow and how you implement that model. For information about the conceptual model underlying components, please see "Components and The Component Manager," on page 4-5.

6.2. Adding a New Entity Behavior (Creating an Actuator)

If you want to create a completely new type of actuator, subclass *DtActuatorComponent* (*actCmpnt.h*). If you want to inherit functionality from one of the existing actuators, such as *DtAutomotiveActuatorComponent*, you can subclass that actuator.

In your actuator, you must do the following:

- Initialize the actuator
- Identify the type of component
- Create input ports to receive input from controllers
- Do what the actuator is intended to do.

The AddActuator example creates an actuator that simulates a vehicle that travels due north regardless of steering inputs, but at a speed regulated by the throttle input. It replaces the *DtAutomotiveActuatorComponent.* The source files for this example are in *./examples/addActuator*.

6.2.1. Initializing an Actuator

You can override init() to perform any initialization of your model. All components have an init() function that gets invoked whenever an entity that uses your actuator is instantiated.

In *myActuator.h*, we do the following:

```
virtual bool init();
```

In *myActuator.cxx*, we do the following:

```
bool MyActuatorComponent::init()
{
  // Here we just call down to the base init, but we can do any necessary
  // initialization of our model here.
  return DtAutomtiveActuatorComponent::init();
}
```

6.2.2. Identifying the Component Type

The type() function returns the string by which the actuator is identified in the object parameter database.

In *myActuator.h*, we do the following:

```
virtual DtString type() const;
```

In *myActuator.cxx*, we do the following:

```
DtString MyActuatorComponent::type() const
{
  return DtAutomotiveActuatorType;
}
```

Since we are replacing *DtAutomotiveActuatorComponent* with the new component, we return the type string normally returned by *DtAutomotiveActuatorComponent*. Every place in the object parameter database that the *DtAutomotiveActuator* is called for (with the string automotive-actuator), the new actuator will be used instead.

If you create a completely new actuator, you must give it a name, then modify the object parameter database to reference that name. For example, if *MyActuatorComponent* was a completely new actuator, it might do the following:

In the header file, define the string, as follows:

```
const char NewActuatorType[] = “new-actuator”;
```

In the source file, do the following:

```
DtString MyActuatorComponent::type() const
{
  return NewActuatorType;
}
```

In the OPE files for the affected entities, do the following:

```
(actuators
  (MyNewActuator
    (component-descriptor-type "automotive-component-descriptor")
    (component-type  "new-actuator")
  )
...
```

6.2.3. Ticking the Actuator

The most important function that you need to override in your derived actuator is tick(). This is where the real work of the actuator gets done. The job of tick() is to compute the state of the vehicle for the current frame, based on:

- The last frame's state
- Controller inputs (throttle position, steering wheel position, and so on)
- Frame length
- Perhaps, information about terrain.

The VR-Forces simulation engine calls tick() each frame, or optionally, at some interval specified by the user.

In the following code, note how the actuator gets information from the state repository and the port input group and how it then calculates and sets new values. (Some of the comments have been deleted. Please see *./examples/addActuator/myActuator.cxx* for the complete code.)

```
void MyActuatorComponent::tick()
{
  // The job of this function is to calculate the state of the object for
  // the current frame and set that state in the state repository of the
  // entity we belong to.
  if ((! mySimManager) || (! entity()->vrfState()) ||
    (! myProcessState) )
  {
    DtWarn("Ticked an unitialized component: %s\n", name().string());
    myAutomotiveControlPortGroup->dataReceived();
    return;
  }

  if (! mySimManager->exerciseClock())
  {
    DtWarn("Unitialized sim manager detected by component: %s\n",
      name().string());
    myAutomotiveControlPortGroup->dataReceived();
    return;
  }

  if (dT() == 0.)
  {
    // No simulation needs to happen, since no time has passed.
    myAutomotiveControlPortGroup->dataReceived();
    return;
  }
  // Grab a pointer to the state repository that we will use to get and
```

```
  // set current state.
  DtVrfObjectStateRepository* groundSR = entity()->vrfState();

  // You can use DtGroundVehicleStateRepository's member functions to
  // get the current state of the entity.
  DtVector lastPosition = groundSR->localPosition();
  DtVector lastVelocity = groundSR->localVelocity();

  // You can get control input values from the input port. The
  // throttle port will give you a throttle value between 0 and 1. In this
  // simple example, we're just interpreting throttle as indicating
  // desired speed as a fraction of max speed (here 10.0).
  double speed = 0;
  if (myThrottlePort)
  {
    speed = myThrottlePort->value() * 10.0;
  }

  // Let's do our simple simulation, moving north in a straight line at a
  // speed proportional to current throttle position.
  DtVector newVelocity(0, speed, 0);
  DtVector amountToMove;
  DtVecScale(newVelocity, dT(), amountToMove);
  DtVector newPosition;
  DtVecAdd(lastPosition, amountToMove, newPosition);

  // follows the height of the terrain.
  DtTerrainInterface* tdb = simManager()->terraininterface();
  if (tdb)
  {
    DtPoint point(newPosition);
    tdb->groundClamp(point);
    newPosition[2] = point[2];
  }

  // Now, update the entity's state repository with the new position and
  // velocity.
  groundSR->setLocalPosition(newPosition);
  groundSR->setLocalVelocity(newVelocity);

  // Remember to notify ports that we have received the data on them.
  myAutomotiveControlPortGroup->dataReceived();
}
```

6.2.4. Adding the Actuator Component to the Component Factory

To use the *addActuator* class, you must add it to the Component Factory. All factories are owned by the Factory Manager, which gets created by the *DtVrfCreator* when you initialize the VR-Forces application. To add a new component to the Component Factory, you make a call to the addCreatorFcn() member function after you initialize the VR-Forces application, but before you start the application's main loop. The following code fragment is from the *main.cxx* file for the addActuator example. It shows how to add *MyActuatorComponent* to the factory.

```
app->init();

  app->cgf()->factoryManager()->componentFactory()->addCreatorFcn(
    DtAutomotiveActuatorType, MyActuatorComponent::creator);

app->mainLoop();
```

6.2.5. Building and Running *myActuator*

When you build the *addActuator* project, it creates a new executable called *addActuator{Debug}.exe*, and places it in the *./bin* directory. The *addActuator* executable is a simulation back-end, extended to contain the new actuator class.

To run the *addActuator* executable, run the front-end and addActuator executable in separate mode.

6.3. Creating a New Controller

This section explains how to create a new controller and add it to an entity. It uses the stopToShoot example.

The VR-Forces models for turreted vehicles do not coordinate movement and firing. In the stopToShoot example we want to:

- Require a vehicle to come to a full stop before firing its ballistic gun
- Optionally, have the entity turn to face its target
- Make the turn-to-face-target behavior configurable in the object parameter database.

Our solution is to create two new controller classes. The *MySuppressMovingFire* controller suppresses firing of the ballistic gun while the entity is moving. The *MyStopToFireComponent* controller makes an entity come to a stop and, optionally, face the target. The component descriptor *MyStopToFireComponentDescriptor* makes *MyStopToFireComponent* configurable at run-time by adding a parameter to the object parameter database.

The source and project files for this example are in *./examples/stopToShoot*. A sample scenario demonstrating the new behavior is provided in *./examples/stopToShoot/scenario/stopToShoot.scn*. In the scenario, two T-72s are tasked to move along a route. At the end of the route is an M1A2. When the T-72s have advanced approximately 2/3 of the way along the route, they detect the M1A2, and then stop and shoot. Once the M1A2 is destroyed, they resume their course along the route.

6.3.1. Creating a Controller

The example creates two controllers. We describe how to create *MyStopToFireComponent*, because it also requires creation of a component descriptor. The process for creating the *MySuppressMovingFire* controller is essentially the same and you can view the comments in the header file for more details.

The general process for creating a controller is as follows:

1. Derive a new class from the appropriate parent class.
2. Implement the following member functions:
   - type(): returns a unique string identifier
   - creator(): a static function that returns a new instance of the class
   - setComponentDescriptor(): overrides the base class to check for a derived type of descriptor
   - tick(): provides a thread of execution to the component each frame.

As with all components, you must register a creator function for the component with the component factory.

After you create a new controller, edit the object parameter database to add it to the controller list for the entities that need to use it. For more information, please see "Adding New Controllers to the Object Parameter Database," on page 6-12.

Implementing the type() Member Function

The type() member function returns a unique string that identifies the component class.

The component types provided by VR-Forces are listed in *compTypes.h*. Be sure that your new component does not conflict with any of the existing types.

In *stopToFire.h*, we do the following:

```
const char MyStopToFireType[] = "stop-to-fire-controller";
```

In *stopToFire.cxx*, we do the following:

```
DtString MyStopToFireComponent::type() const
{
  return MyStopToFireType;
}
```

The creator() Member Function

The creator() member function returns a new instance of the derived component to be registered with the component factory.

In *stopToFire.h*, we do the following:

```
class MyStopToFireComponent : public DtGroundAutoControllerComponent
{
public:
  . . .
  // Creator function to be registered with DtSimCreator
  static DtSimComponent* creator(const DtString& name,
    DtVrfObject* owner, DtSimManager* simManager,
    DtComponentDescriptor* desc,
    DtReaderWriterRegistry* parentRegistry);
};
```

In *stopToFire.cxx*, we do the following:

```
DtSimComponent* MyStopToFireComponent::creator(const DtString& name,
  DtVrfObject* owner, DtSimManager* simManager,
  DtComponentDescriptor* desc, DtReaderWriterRegistry* parentRegistry)
{
  return new MyStopToFireComponent(name, owner, simManager, desc,
      parentRegistry);
}
```

The setComponentDescriptor() Member Function

The setComponentDescriptor() member function overrides the base class member function to check the type of descriptor being set for the component and cache a pointer to the new derived type.

In *stopToFire.cxx*, we do the following:

```
void MyStopToFireComponent::setComponentDescriptor(DtComponentDescriptor
  * descriptor)
{
  DtControllerComponent::setComponentDescriptor(descriptor);
  MyStopToFireComponentDescriptor =
    dynamic_cast<MyStopToFireComponentDescriptor *>(descriptor);

  if (! MyStopToFireComponentDescriptor)
  {
    DtWarn("Stop To Fire Controller specified with incorrect "
      "component descriptor. \n");
    DtWarn("It should use 'stop-to-fire-descriptor'\n");
    DtWarn("Check your object parameter database file.\n");
  }
}
```

The tick() Member Function

The tick() member function is where the controller does its work. If the entity has a target and permission to fire, we attempt to activate the automotive control port group. (For details about how controllers establish priority, please see "Setting the Priority of Components," on page 5-16 and "Adding New Controllers to the Object Parameter Database," on page 6-12.) The controller sets the brake, kills the throttle in the port group, and optionally, steers to face the target as shown in the following code:

```
if (haveTarget() && havePermissionToFire())
{
  if (myAutomotiveControlPortGroup->attemptToActivateAllPorts())
  {
    myBrakePort->setValue(1.0);
    myParkingBrakePort->setValue(false);
    myThrottlePort->setValue(0.0);

    myStoppingToFire = true;

    if (myStopToFireComponentDescriptor &&
      myStopToFireComponentDescriptor->turnToTarget())
    . . .
```

The attemptToActivate() member function checks whether or not the port group's connection is active. Attempts by other controllers to activate the port group during this tick will fail. As described in "Adding New Controllers to the Object Parameter Database," on page 6-12, by placing the MyStopToFire controller first on the controller list, you can be assured that the new stop-to-shoot behavior overrides the default movement behaviors.

6.3.2. Creating a Component Descriptor

A component descriptor specifies the kind of component to create and provides a way to set parameters for components. The component descriptor has member data that you can set as parameters in the object parameter database. In the Stop-to-Shoot example, *MyStopToFireComponentDescriptor* allows you to specify the boolean **turn-to-target** parameter to the *MyStopToFireController*.

The component descriptor needs to implement type() and creator() member functions similarly to components. Please see the comments in *stopFireDesc.h* for the functions that are common to all component descriptors. The turn-to-target data member (**myTurnToTarget**) is a *DtRwBoolean* (a Boolean object, derived from *DtReaderWriter*). We register this data member with the component descriptor's parent registry in the constructor, to make it possible to read the parameter from a file, as part of a component descriptor's entry in the object parameter database. (For details, please see Chapter 14, *Reading and Writing Files*.)

In *stopFireDesc.h*:

```
class MyStopToFireComponentDescriptor : public DtComponentDescriptor
{
public:
  // real constructor
  MyStopToFireComponentDescriptor(const DtString& name,
    DtReaderWriterRegistry * parentRegistry = 0);

  . . .

  // Provide an accessor and mutator for the parameter value
  virtual void setTurnToTarget(bool turnToTarget);
  virtual bool turnToTarget() const;

  . . .

protected:

  DtRwBoolean myTurnToTarget;
};

// real constructor
MyStopToFireComponentDescriptor::MyStopToFireComponentDescriptor(
  const DtString& componentName,
  DtReaderWriterRegistry * parentRegistry) :
  DtComponentDescriptor(componentName, parentRegistry),
  myTurnToTarget("turn-to-target", & myRegistry)
{
  myTurnToTarget = true;
}
```

We registered the **myTurnToTarget** data member with its *DtReaderWriter* name in the constructor turn-to-target. This is the name of the parameter as it will appear in the component descriptor's entry in the object parameter database, for example (from *./data/simulationModelSets/developer_toolkit_examples/stopToShoot/vrfSim/systems/movement/ground-tracked.sysdef*):

```
(controllers
  . . .
  (stop-to-fire
    (component-descriptor-type "stop-to-fire-descriptor")
    (component-type  "stop-to-fire-controller")
    (min-tick-period  -1.000000)
    (min-tick-period-variance  -1.000000)
    (process-state-repository-name  "")
    (process-state-repository-type  "")
    (turn-to-target True)
  )
. . .
)
```

Like the MyStopToFireComponent, we must provide a type() member function to uniquely identify the new class, and a static creator() member function to be registered with the component descriptor factory.

6.3.3. Adding New Controllers to the Object Parameter Database

You can use the new controllers with any entity that has a ballistic gun. In this example, we add the new controllers to the T-72 entity. The key to remember when you add controllers to a controller list, is that controllers establish their priority (the order in which they are ticked) by their order in the list. Figure 6-1 illustrates the default components for the T-72 (from *ground-tracked.sysdef* and *125mm-gun.sysdef*). The priority for ticking controllers is from top-to-bottom. In particular, note the order of controllers that affect the powertrain and main gun.

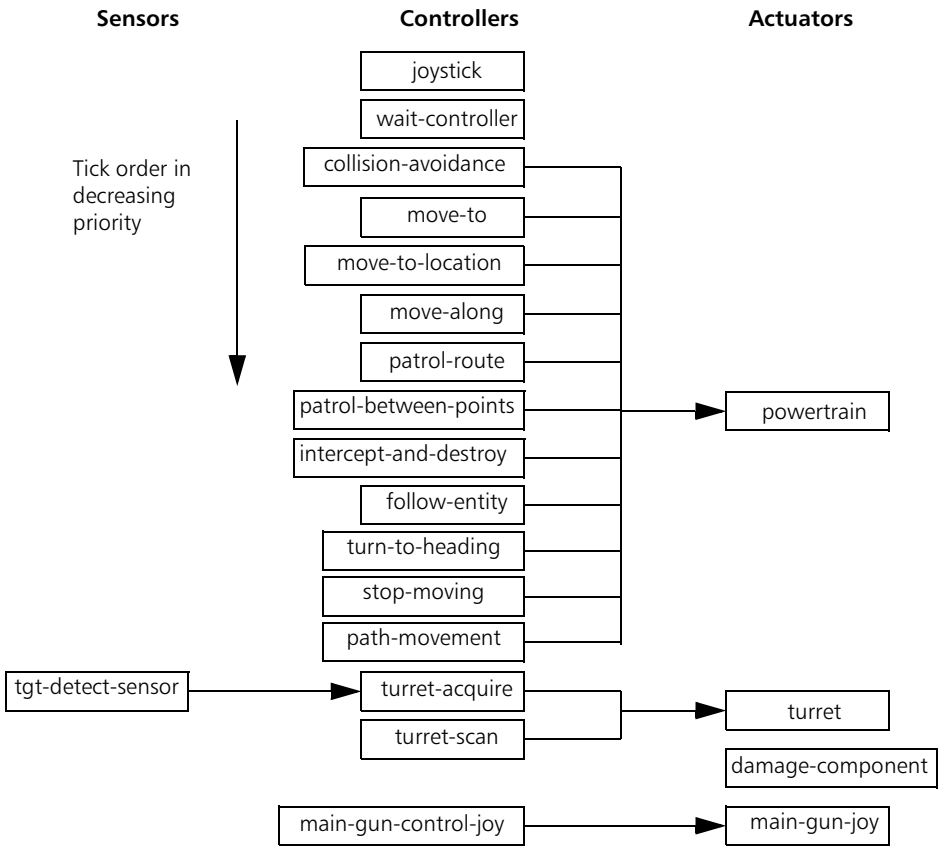


Figure 6-1. Default component list for T-72

Since we want the T-72 to stop when it fires, the stop-to-fire controller needs to control the powertrain when the entity wants to fire. So, we add the stop-to-fire controller to the controller list before the other powertrain controllers (Figure 6-2).

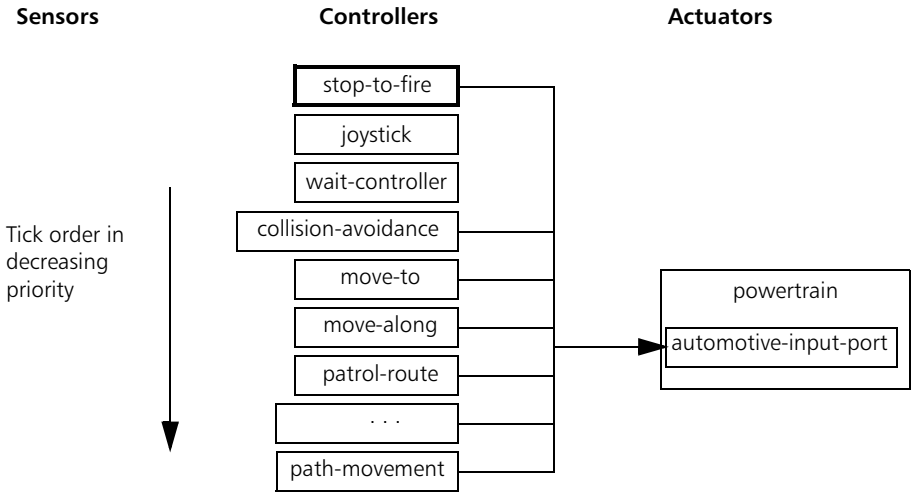


Figure 6-2. stop-to-fire controller with highest powertrain priority

The same logic applies to the suppress-moving-fire controller. If the entity is moving, you do not want to let it shoot, so we place the suppress-moving-fire controller ahead of the main-gun-control controller as input to the main gun (Figure 6-3).

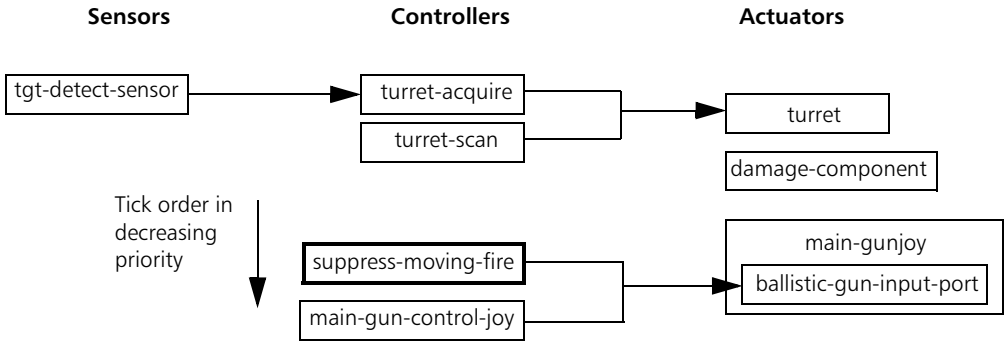


Figure 6-3. suppress-moving-fire controller with top priority for main gun

After we add the new controllers to the appropriate system definition files in the object parameter database, they look like Example 6.1 and Example 6.2.

Example 6.1

In *./data/simulationModelSets/developer_toolkit_examples/stopToShoot/vrfSim/systems/-movement/ground-tracked.sysdef*:

```
(controllers
  . . .
  (stop-to-fire
    (component-descriptor-type "stop-to-fire-descriptor")
    (component-type  "stop-to-fire-controller")
    (min-tick-period  -1.000000)
    (min-tick-period-variance  -1.000000)
    (process-state-repository-name  "")
    (process-state-repository-type  "")
    (turn-to-target True)
  )
  . . .
)
(connections
  ;; Connect the new stop-to-fire controller with the auto actuator
  (connect stop-to-fire:automotive-control
  powertrain:automotive-control)
  ...
)
```

Example 6.2

In *./data/simulationModelSets/developer_toolkit_examples/stopToShoot/vrfSim/systems/-weapons/125mm-gun.sysdef*:

```
(controllers
  ;; Add the suppress moving fire controller
  (suppress-moving-fire
    (component-descriptor-type "ballistic-gun-ctlr-descriptor")
    (component-type  "suppress-moving-fire-controller")
    (min-tick-period  -1.000000)
    (min-tick-period-variance  -1.000000)
    (process-state-repository-name  "")
    (process-state-repository-type  "")
    (load-ammo-time  0.000000)
    (unload-ammo-time  0.000000)
    (ammo-select-table-file
    (filename  "$(ammoselect-dir)\T80MainGun.asl")
  )
)
...
(connections
  ;; Connect the new suppress-moving-fire with the gun actuator
  (connect suppress-moving-fire:gun-control main-gun-joy:gun-control)
  ...
)
```

The controller components are connected to their corresponding actuators by specifying the names of the component and the port group to connect to in the connections list. Please see "Connecting Components," on page 5-12 for a discussion of how to connect components through their ports and port groups. The stop-to-fire controller (MyStopToFire) is connected to the powertrain (*DtAutomativeActuator*) through the port group named automotive-control (line 16 in Example 6.1). The suppress-moving-fire controller (MySuppressMovingFire) is connected to the main-gun (*DtBallisticGun*) through the port group named gun-control (line 19 in Example 6.2).

The component descriptor for the stop-to-fire controller is of type stop-to-fire-descriptor (the new *MyStopToFireComponentDescriptor* class.) The component type for the controller (MyStopToFireController) is stop-to-fire-controller (line 5 in Example 6.1). The optional behavior of turning to the target for the stop-to-fire controller is specified in the turn-to-target parameter (line 10 in Example 6.1).

6.4. Writing Components for a Multi-Resolution Model

Creating or extending components in VR-Forces is explained and demonstrated in a number of examples. However, when writing components that are intended to be used in a multi-resolution model, there are some additional steps you must take to ensure your model will work in this context.

While the aggregate model is currently the only example of a multi-resolution model in VR-Forces, the guidelines for writing components that are designed to be dynamically enabled and disabled can be applied to any type of entity.

6.4.1. Components and Systems can be Enabled and Disabled Dynamically

Components can exist in two states: enabled or disabled. In the enabled state, the component's tick() member function is periodically called by the Component Manager. In the disabled state, the tick() function does not get called. The *DtSimComponent* class has enable() and disable() member functions that can be called to enable or disable a given component. Components are enabled by default.

Similarly, you can enable or disable entire systems of components, by calling DtComponentSystem::enable() or DtComponentSystem::disable().

In the aggregate model, the *DtAggregateModelResolutionController* is responsible for enabling or disabling the appropriate sets of components and systems. It triggers the switch between models by calling enable() and disable() on the aggregate's other components and systems.

6.4.2. Handling Handover of Tasks between Components

Normally, just one task controller component registers to handle a given task type in an entity. In a multi-resolution model, there may be more than one. You will want to prepare for the situation in which your task controller is disabled or enabled, due to a switch to another set of components or systems. To accomplish this, you must write your task callback function to take into account the current task state at the time of the switch. When the Task Manager, invokes a task callback, it passes in a *DtSimTaskState* (*simTaskState.h*) object, containing the current execution status of the task. The Task Manager manages the handoff of task state between the component being disabled and the newly enabled components. If a task was being executed at the time of an enable/disable transition, the Task Manager obtains a filled-out *DtSimTaskState* object from the now-disabled task controller and hands it into the newly enabled task controller (as an argument to its task callback). The new task controller can use that *DtSimTaskState* data to decide how it should initiate its task behavior (that is, start in the middle of the task, instead of the beginning).

There is a *DtSimTaskState* class for every *DtSimTask* class that needs one (that is, in cases in which it makes sense to keep track of some intermediate state). For example, the *DtSimWaitTaskState* class holds the length of time already executed in the task. The *DtPatrolBetweenPointsTaskState* class holds the identity of the waypoint the entity is moving toward.

To support the transfer of current task state, in your derived task controller you must implement member functions in the following way (declared in *taskCtrlCmpt.h*):

- Implement currentTaskState(). This is the function that the Task Manager invokes prior to disabling a component. In this function, you must create a new instance of the appropriate type of *DtSimTaskState* object (the one that corresponds to the type of task your controller is executing). For example, if your controller executes a *DtMoveAlongTask*, you must instantiate a *DtMoveAlongTaskState* object. Fill out the task state object with the current task status (probably obtained from the component's process state repository). This function should return a pointer to the new *DtTaskState* instance.
- Implement a task callback function. It initiates the task behavior for the component. Use data in the incoming DtSimTaskStatePtr to initialize the controller, configuring it to begin executing the task at the appropriate point in its overall execution state (that is, continue along a route, rather than starting over again at the beginning). The prototype for the callback function (in *taskCtlrCmpnt.h*) is:

  ```
  typedef void (*DtTaskMessageCallbackFcn)(DtSimMessage * msg,
    void * usr, DtSimTaskStatePtr taskState);
  ```

  In your callback function, dynamically cast the incoming **taskState** parameter to the expected type. For example, if you have registered a callback for handling *DtMoveAlongTasks*, then the corresponding task state will be of type *DtMoveAlongTaskState*. For example,

  ```
  DtMoveAlongTaskState* taskState =
    dynamic_cast<DtMoveAlongTaskState*>(taskStatePtr.get());
  ```

This discussion assumes that you are writing a task controller component that handles tasks that have some intermediate task state that must be maintained across a transfer between components. However, there are many cases in which this is not necessary. For example, a *DtMoveToLocationTask* does not have any task state associated with it that needs to be transferred between components. For this task, there is no corresponding *DtSimTaskState* class. There is no need to implement currentTaskState(). In your task callback, there is no need to use the incoming *DtSimTaskStatePtr* to initialize the controller with the current task state. (It will point to a base *DtSimTaskState* object that contains no specific task state data.)

6.4.3. Registering and Unregistering for Tasks

To ensure that your task controller component registers for the appropriate tasks, whenever the component is enabled do the following:

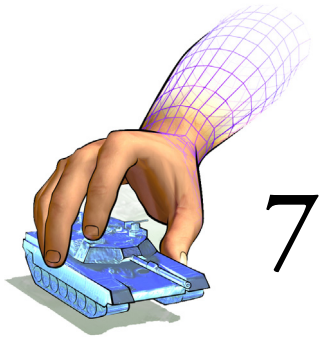
- Register any task callback functions you want to handle in your *DtTaskControllerComponent* by overriding DtTaskControllerComponent::registerTaskMsgCallbacks(). Be sure to call down to the base class in your derived version.
- Unregistration of tasks occurs automatically, whenever the component is disabled. It is sufficient to unregister for any task callbacks in the component's destructor.

6.4.4. Registering and Unregistering for Set Data Requests

To ensure that your component registers and unregisters for the appropriate set data requests, automatically registering for them when the component gets enabled, and deregistering for them when it gets disabled, do the following:

- Override DtControllerComponent::enable(), and register any set data request callbacks there. Be sure to call down to the base class in your derived version.
- Override DtControllerComponent::disable(), and remove registration of any set data request callbacks there. Be sure to call down to the base class in your derived version.

If you have multiple components that will potentially handle the same *DtSetDataRequest*, registering and unregistering set data requests in the above hook functions ensures that your component will be able to handle set data requests appropriately, whenever the component gets enabled or disabled. This will avoid the situation in which multiple components simultaneously attempt to handle a given set data request.



7

Sensors

This chapter describes sensor concepts and how to create them. It assumes that you are familiar with the preceding chapters, particularly Chapter 5, *Component Concepts* and Chapter 6, *Creating Actuators and Controllers*.

7.1. Sensors

Entities in VR-Forces are configured with sensors that use the VR-Forces Signature Sensor system to determine what other objects in the simulation they are able to sense. This system allows VR-Forces to simulate sensors in many different domains, such as visual, radar, infrared, and sonar. The sensors delivered with VR-Forces are organized as systems in system definition files. For details about systems and system definition files, please see Section 5.6, "Editing an Entity's Systems" and Section 7.4, "System Definition Files", in *VR-Forces Configuration Guide*.

7.1.1. Signature Sensor Concepts

Signature sensors are based on the following concepts:

- A sensor domain is a particular class of sensor, such as visual, radar, sonar, infrared, and so on. You can add new sensor domains by editing configuration files.
- A sensor signature is a floating point number that represents the ease of being sensed in a particular sensor domain. The higher the signature value, the easier it is to sense an entity in that domain.
- An absolute signature is the sensor signature of a particular entity, in a particular domain, at a particular time, independent of the observer. This is a property of the entity being sensed.
- An apparent signature is the sensor signature of a particular entity, in a particular domain, from a specific viewpoint. This is the absolute signature corrected for the distance and obstructions between the entity being sensed and the viewpoint.

The sensor model is implemented through the following parameter blocks in the object parameter database and configuration files:

- Target object – Each object that is potentially observable produces an absolute signature value for any sensor domain in which is it detectable. Entries in the object parameter database set the base signature value, and optionally, some rules about how that signature is altered based on the state of the entity.
- Signature propagation – Each sensor domain has a global propagation model. The global propagation model applies to all entities using sensors in a given sensor domain. These models modify absolute signatures from target objects to correct for the viewer's location, as well as any environmental effects, such as being obscured by terrain, and return the relative signature. These models are registered and configured in the *./data/simulationModelSets/<model_set>/physicalWorldParams.mtl* file.
- Sensor – The sensor component configured on an entity that is doing the sensing takes the relative signature returned from the propagation model and determines probability of detection. Sensor parameters are configured in the descriptor for each sensor in the object parameter database.

7.1.2. Target Object

Every object type in VR-Forces can be configured with sensor signature values for any of the sensor domains. These are configured in the object parameter database entry for the object type, and apply both to local objects and remote objects. For each domain, an object has a base signature, and a set of modifiers. The base signature is a single value that represents the signature for that domain. The modifiers specify multipliers for that base value based on the current state of the object. For example, an entity might have a base value of 5.0 for its visual signature, and a modifier of 1.5 when moving. This means the signature of the entity will be 7.5 when moving and 5.0 when not moving. Signature modifier rules can be placed in an external file, and referenced from many different object types.

The absolute signature values configured in the target object have no units. When used with the standard propagation model and a standard sensor, the absolute signature is the maximum distance (in kilometers) from which the object can be spotted. However, in general, the absolute signature is not necessarily expressed in units of distance. It is a function of the combination of a particular propagation model and sensor models.

Absolute signatures are specified separately for each sensor domain, and are completely independent of each other.

7.1.3. Signature Propagation

The signature propagation portion of the sensor system models the spatial relationship between the sensor and the target and the effects of the environment on detection. This is modeled by converting the absolute signature calculated by the target object into the relative signature given an observer position.

Signature propagators are configured for each of the sensor domains in the physical world parameter file. Each domain can use a different propagation model, or have its models configured differently.

The VR-Forces standard signature propagator, which is used by default for all sensor domains, returns a relative signature equivalent to the absolute signature * 1000 / range between observer and target. Optionally, the standard propagator can be configured to check for terrain or entities blocking the direct line-of-sight between the target and observer, and return a relative signature of 0 when there is an obstruction.

7.1.4. The Sensor Component

The sensor is the component that is configured on the observing entity. The basic version of this sensor is called the signature-object-sensor. It is configured for a specific sensor domain and uses the signature propagator for that domain to determine the relative signatures of any other simulated objects within its maximum range. Sensors are configured with detection probability tables, that map relative signature values to probability of detection. The VR-Forces default configuration is the "standard" sensor, which can detect any object with a relative signature of 1.0 or greater. Higher relative signatures have a greater probability of detection, so they will be seen sooner. More sensitive sensors could specify smaller relative signature values in their detection probability files.

7.1.5. Adding Sensor Domains

Sensor domains are specified using a string name of the domain, such as "visual." To work, the sensor domain must be listed in all three of the sensor system's components – the target object configuration, the propagator configuration, and the sensor itself. By default, VR-Forces has includes sensor domains for "visual", "radar", "infrared", and "sonar". You can add new sensor domains by choosing a new unique sensor domain name and adding it to the configuration files. For example, to add a new acoustic sensor to an entity, so that it could hear nearby entities:

1. For all entity types that would potentially be detected by the acoustic sensor, add a new entry to the "sensor-signatures" block in the object parameter database entry with the name "acoustic" and the base signature value. Modifiers could be added as well.
2. In the *physicalWorldParams.mtl* configuration file, add a new entry for an acoustic propagator, using "acoustic" as the signature name.
3. Add a signature-object-sensor to the sensors list in the object parameter database entry for the entity that will detect objects this way. It would specify "acoustic" as its sensor-domain.

If the standard propagator model, or the signature rules defined by VR-Forces are not sufficient, new ones can be defined and added though factories.

7.1.6. The Radar Sensor

VR-Forces provides a radar sensor, which is a specialized version of the signature object sensor. It has all the same functionality of the signature object sensor, plus it publishes an emitter system and one or more emitter beams, which are configured though the descriptor for the sensor.

7.2. Adding a Sensor Component

This section describes how to add a new sensor component and connect it to controller components. The radarWarnRx example demonstrates how to add the following new capabilities and behaviors to existing entities in VR-Forces:

- Radar warning receiver
- Radar sensor
- Reactive behavior in response to a radar threat detected by the radar warning receiver.

This example extends the CIS generic attack/strike fixed-wing entity to have a radar warning receiver. It also adds a reactive behavior to the plane, such that when its radar warning receiver detects a radar, the entity moves to a designated safe location. The safe location is a control point positioned out of range of the radar threat. Once it has reached the safe location, it resumes the task it was executing (if any).

The sensors and behaviors implemented in this example are not realistic.The purpose of the example is to show how and where you would incorporate your own algorithms and models into VR-Forces.

7.2.1. Source Files, Project Files, and Scenario Files

The source and project files for the radarWarnRx example are in *./examples/RadarWarnRx*. The example creates a plug-in for the back-end.

To run the example, start vrfSim as follows (along with any other relevant arguments):

```
vrfSim --loadPlugin radarWarnRx
```

An example scenario demonstrating the new behaviors is in *./examples/RadarWarnRx/scenario/RadarWarnRx.scn*. The scenario loads an object parameter database that configures the CIS generic attack/strike fixed-wing entity with new sensor and behavior components. The example contains a single CIS generic attack/strike fixed-wing entity. The entity has a plan that contains a task to move-to waypoint Alpha. As it moves toward the waypoint, it comes within detection range of the radar. When it comes into range, it reacts by moving to a safe location (a control point named Safe Haven). Upon reaching the control point, it resumes its original task to move toward waypoint Alpha.

7.2.2. Creating the New Sensor Components

VR-Forces provides a basic emitter, *DtEmitterComponent* (*emitterComp.h*), so we can derive our radar sensor from this component. Anytime you create a new component, you have to provide the following member items in your new class:

- A constructor of the form:

```
DtSimComponent(const DtString & name,
  DtVrfObject* owner,
  DtSimManager* simManager,
  DtComponentDescriptor* desc = 0,
  DtReaderWriterRegistry* parentRegistry = 0);
```

- A type() member function that returns a string that uniquely identifies the new class
- A static creator function that returns a newly created instance of the class.

The string returned by the type() function is the key into the component factory that VR-Forces uses to instantiate new components. Please see *compTypes.h* for the set of string types for VR-Forces component classes.

7.2.3. Implementing the Radar Warning Receiver Class

The radar warning receiver class, *DtSimpleRadarWarningReceiver* (*simpleRadarWarnRx.h*), needs to detect the emissions and notify any interested controllers that it has detected one or more radars. In this example it merely checks for anything sending out emissions within a 5 Km radius. The class has a VR-Link *DtReflectedEmitterSystemList* data member, **myReflectedEmitterList**, which it uses to gain knowledge of all entities generating emission PDUs over the network.

If it finds any, it extracts the host ID of the emitter and places that host ID on an emitter list port. The radar warning receiver uses the existing *DtEntityListOutputPort* class to communicate the identity of the set of radars it has detected in the current tick. It will ultimately be connected to the radar warning response controller that will receive the data on the emitter list port (a list of detected radars) and use that information to drive its behavior.

As a provider of data (the list of detected emitters), the *DtSimpleRadarWarningReceiver* has a *DtEntityListPort* data member, **myEmitterListPort**. It is responsible for creating and setting values on the port. Each tick, it does the following:

1. Calls attemptToActivate() on its emitter list output port, to check if the connection through this port group is active. This checks to see if the recipient will receive data placed on the ports in the group (this will only happen if there is no component with a higher priority currently connected).
2. Iterates through its reflected emitter list.
3. If it finds an emitter that is within 5 Km of the entity's current location and at least one has a power greater than zero, it places it on the emitter list port.

```
void DtSimpleRadarWarningReceiver::tick()
```

```
{
  if(!entity())
  {
    return;
  }

  // Safety check - this should have been created in init()
  if(!myEmitterListPort)
  {
    return;
  }

  // Attempt to take control of the emitter list output port
  if (myEmitterListPort->attemptToActivate())
  {
    // Clean out list of emitter systems on port
    myEmitterListPort->emptyList();
    // Iterate through list of current emitter systems
    DtReflectedEmitterSystem* emitterSystem;
    for(emitterSystem = myReflectedEmitterList->first(); emitterSystem;
      emitterSystem = emitterSystem->next())
    {
      DtGlobalObjectDesignator emitterSystemHostId =
        emitterSystem->emitterSysRep()->hostId();
      // Make sure it doesn't belong to us
      if(emitterSystemHostId == myGlobalId)
      {
        continue;
      }
      // Look up the entity with the emitter
      DtVrfObject* emitterEntity = mySimManager->vrfObjectManager()->
        lookupVrfObjectByGlobalId(emitterSystemHostId);
      // If for some reason we can't find it, move on to the next one
      if(!emitterEntity)
      {
        continue;
      }
      // Check for at least one beam with >0 ERP.
      DtListItem *beamItem = emitterSystem->emitterSysRep()->
        beamList()->first();
      DtEmitterBeamRepository *beamSR = NULL;
      bool found = false;
      while (beamItem && !found)
      {
        beamSR = (DtEmitterBeamRepository*) beamItem->data();
        if (beamSR->ERP() > 0)
        {
          // Check the distance to the emitter - if it falls in a 5 km
          // range, add it to our list of detected emitters
          double rangeSquaredToEmitter = DtDistSqr(
            entity()->vrfState()->localPosition(),
            emitterEntity->vrfState()->localPosition());
          if(rangeSquaredToEmitter < 25000000.0)
          {
            // Consider the emitter system to be detected. Put it on
            // our port.
            myEmitterListPort->addEntity(
              emitterEntity->objectIdentifier());
            found = true; // found 1, so don't check for others.
          }
```

```
            }
            beamItem = beamItem->next();
          }
        }
      }
    }
```

7.2.4. Creating the New Controller Component

The new controller component, *DtSimpleRadarWarningResponse* (*simpleRdrWrnResp.h*), provides a simple response when it receives a warning from the radar warning receiver. It attempts to move toward a waypoint named Safe Haven, if an object by that name exists in the simulation.

The controller for handling the reaction to a threat detected by the radar warning receiver is intended for a fixed-wing entity. All of the movement-related controllers for fixed-wing aircraft are derived from *DtFixedWingPilotController*, so we derive from that class for our new controller. Among other things, we inherit the ports required to control a *DtFixedWingActuatorComponent*, the actuator used by the F-16 entity.

Like the sensor components created in "Creating the New Sensor Components," on page 7-6, the following member functions must be implemented for the component to function correctly within the component architecture:

- A constructor of the form:

```
DtFixedWingPilotController(const DtString & name,
  DtVrfObject* owner,
  DtSimManager* simManager,
  DtComponentDescriptor* desc=0,
  DtReaderWriterRegistry* parentRegistry = 0);
```

- A type() function that returns a string that uniquely identifies the new class
- A static creator function that returns a newly created instance of the class.

The string returned by the type() function is the key into the component factory that VR-Forces uses to instantiate new components. Please see *compTypes.h* for the set of string types for VR-Forces component classes.

7.2.5. Connecting the Sensor to the Controller

We need to connect the radar warning receiver sensor to the radar warning response controller. As a consumer of data, the radar warning response class has a *DtEntityListInputPort* member variable, **myEmitterListPort**, through which it receives the identity of any detected radars (generated by the radar warning receiver). We override createPorts() to instantiate the input port member and add it to our list of input ports. The name of the port, emitter-list, identifies the port when connecting components to one another in the object parameter database. Similarly, in the sensor controller, we override createPorts() to create its output port member variable, **myEmitterListPort**, also with the name emitter-list. These two components are connected to one another through these ports by name, in the object parameter database entry.

```
(US-attack-strike-fixed-wing-platform
(parameter-type "fixed-wing-entity-param")
. . .
(sensors
  . . .
  (simple-radar-warning-rx
  . . .
  )
)
(controllers
  . . .
  (simple-radar-warning-reaction-controller
  . . .
  )
)
  (connections
    . . .
  ;; connect <component-name>:<port-name> <component-name>:<port-name>
  ;;
    (connect system:emitter-list
      simple-radar-warning-reaction-controller:emitter-list)
    (connect simple-radar-warning-reaction-controller:
      fixed-wing-control flight-kinematics:fixed-wing-control)
    . . .
  ) ;; end connections
```

To implement the reactive behavior in *DtSimpleRadarWarningResponse*, we override the tick() function to perform the following:

1. Check to see if we can find our safe location. The safe location is a control point named Safe Haven (mySafeControlPointName in the code). Look it up in the Object Manager.

```
if(!safeControlPoint)
{
  DtWarn("DtSimpleRadarWarningResponse::tick() :
    Invalid safe control " "point %s\n",
    mySafeControlPointName.string());

  return;
}

DtVector safeLocation = safeControlPoint->vrfState()->
  vrfKinematicState()->localPosition();
```

2. If we are in the process of reacting to a previous threat, move toward our safe location. Check to see if we have arrived at our safe location. If we have, we are done and we no longer need to control the actuator. Otherwise, continue moving toward the safe location.

```
// Take control of the port group - this gives us exclusive control
// over the fixed-wing actuator until we give that control up or a
// higher priority controller overrides us
if (myFixedWingControlPortGroup->attemptToActivate())
{
  if (inVicinity(safeLocation))
  {
    // We're done
    DtInfo("DtSimpleRadarWarningResponse::tick(): we're done! "
      "Resume normal activity\n");
    myReactingToWarning = false;
    // Give up control of actuator's port group so other
    // components may control it.
    myFixedWingControlPortGroup->deactivate();
  }
  else
  {
    // Move toward the safe location
    flyTo(safeLocation, myFixedWingEntityState->orderedSpeed());
  }
}
```

3. If we are not currently reacting to a detected radar, check the port to see if any have been detected and are in range. We are looking for the first emitter we find in range. If we find one, take control of the actuator's port group and fly to a safe location.

```
// Get 1st emitter system on list of emitters on the port
const DtList* emitterList = myEmitterListPort->value();

// If there aren't any emitters out there, there's nothing to react to
if (!(emitterList->first() && emitterList->first()->data()))
{
  // Free control of the port so that sensor can take
  // control next tick
  myEmitterListPort->dataReceived();
  return;
}

// We have an emitter out there
DtEntityIdentifier * id =
  (DtEntityIdentifier *)emitterList->first()->data();

// Look up the entity with the radar
DtVrfObject* emitterSystemHost =
mySimManager->vrfObjectManager()->lookupVrfObject(*id);

if (!emitterSystemHost)
{
  // Free control of the port so that sensor can take control next tick
  myEmitterListPort->dataReceived();
  return;
}
```

```
// We are now reacting to an emitter
myReactingToWarning = true;
DtInfo("Simple radar warning rx detected emitter %s\n", id->string());
DtInfo("Move away from the source!\n");

// Take control of port group - this gives us exclusive control
// over the fixed-wing actuator for the duration of the frame (no other
// controllers that come after this one will be able to control it).
if (myFixedWingControlPortGroup->attemptToActivate())
{
  // Move toward the safe location
  flyTo(safeLocation, myFixedWingEntityState->orderedSpeed());
}
// Free control of the port so that sensor can take control
// next tick
myEmitterListPort->dataReceived();
```

7.2.6. Adding the New Components to VR-Forces

To get your new sensor and controller components into VR-Forces you must register the creators with the component factory. For details, please see "Adding the Actuator Component to the Component Factory," on page 6-6. You do this in *main.cxx*:

```
app->init();

// Register some new creators for the components we are adding to the
// back-end.

// Register the radar warning receiver sensor. The string passed into
// this function is the same as the string returned by
// DtSimpleRadarWarningReceiver's type() function.
app->cgf()->factoryManager()->componentFactory()->addCreatorFcn(
  "simple-radar-warning-receiver",
  DtSimpleRadarWarningReceiver::creator);

// Register the radar warning reaction controller. The string passed
// into this function is the same as the string returned by
// DtSimpleRadarWarningResponse's type() function.
app->cgf()->factoryManager()->componentFactory()->addCreatorFcn(
  "simple-radar-warning-response",
  DtSimpleRadarWarningResponse::creator);

// Register the scanning radar sensor. The string passed into this
// function is the same as the string returned by DtScanningRadar's
// type() function.
app->cgf()->factoryManager()->componentFactory()->addCreatorFcn(
"scanning-radar", DtScanningRadar::creator);

app->mainLoop();
```

7.2.7. Configuring Entities to Use the New Components

Now that we have incorporated the new components into the application, we need to configure specific entities to use the new components. In our new sensor and controller classes we did not add any new parameters, so we use the existing parameter and component descriptor classes associated with each component. (For an example of adding new parameters see "Creating a New Controller," on page 6-6.)

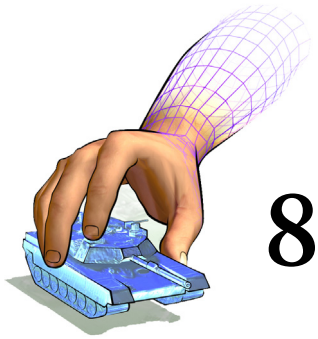
In this example, we extended the CIS generic attack/strike fixed-wing entity with the new radar warning receiver and corresponding response controller.

```
;;
;; Entry for CIS generic attack/strike fixed wing platform
;;
(CIS-attack-strike-fixed-wing-platform
  (parameter-type "fixed-wing-entity-param")

  ...

  (sensors
    . . .
    (simple-radar-warning-rx
      ;; We don't have any special parameters - the base class component
      ;; descriptor will suffice
      (component-descriptor-type "component-descriptor")
      ;; This identifies the class type to create for this sensor. It
      ;; should match the string returned by
      ;; DtRadarWarningReceiver::type().
      (component-type "simple-radar-warning-receiver")
      ...
    )
  )
  ...
) ;; end US-attack-strike-fixed-wing-entity
```

In the *fixed-wing-fighter-jet.sysdef* file:

```
(controllers
. . .
  ;;
  ;; Add our new controller component
  ;;
  (simple-radar-warning-reaction-controller
    (component-descriptor-type "fixed-wing-pilot-controller-
descriptor")
    (component-type  "simple-radar-warning-response")
    (attached-to-list )
    (min-tick-period  -1.000000)
    (min-tick-period-variance  -1.000000)
    (process-state-repository-name  "")
    (process-state-repository-type  "")
    (max-turnrate-cutoff-angle  0.523600)
    ; Additional parameters for the descriptor class we're using
    ...
  )
)
(actuators
  . . .
) ;; end actuators

(connections
  . . .
  ;; Configure new controller inputs (emitter-list) to be system
  ;;inputs
  (connect system:emitter-list
    simple-radar-warning-reaction-controller:emitter-list)
  ;; Connect the new controller to the fixed wing actuator
  (connect simple-radar-warning-reaction-controller:
    fixed-wing-control flight-kinematics:fixed-wing-control)
  . . .
) ;; end connections
```

A modified object parameter database (configured with the new components) is in *./data/simulationModelSets/developer_toolkit_examples/radarWarnRx/vrfSim/RadarWarnRx.opd*.



8

Messages

This chapter describes the message interface and how to use radio messages and interface messages.

8.1. The Message Interface

This chapter focuses on the message-passing mechanism for exchanging interface messages.

VR-Forces applications communicate all information through DIS PDUs or HLA RPR FOM objects and interactions. In addition to the standard protocol messages, VR-Forces sends messages that VR-Forces applications use to communicate with one another, such as scenario management, creating and deleting objects and entities, and executing plans. This data is sent using instances of the *DtVrfMessageInterface* (*vrfMsgIf.h*) class. VR-Forces applications have a *DtVrfMessageInterface*, maintained by the *DtSimManager* class. Applications that use the Remote Control API also have a *DtVrfMessageInterface*, maintained by the *DtVrfRemoteController* class.The *DtVrfMessageInterface* sends and receives *DtSimInterfaceMessages* (*ifaceMsg.h*). The *DtVrfMessageInterface* is initialized with a pointer to the *DtExerciseConnection* and a simulation address. The actual *DtSimInterfaceMessages* are wrapped inside of VR-Link *DtDataInteractions*. Through this mechanism, VR-Forces effectively extends the DIS and HLA RPR FOM protocols to support additional application-specific messages.

A *DtSimInterfaceMessage* has a pointer to an instance of a *DtSimInterfaceContent* (*ifaceCont.h*). All interface messages are derived from *DtSimInterfaceContent*. The *DtSimInterfaceMessage* acts as an 'envelope' capable of containing various kinds of content. The different kinds of interface content types are identified by integers. The integers for *DtSimInterfaceContent* types are specified in *msgTypes.h*.

8.2. Sending an Interface Message

You can send an interface message to a VR-Forces application using the *DtVrfMessage-Interface's* createAndDeliver() member function. The specification of the member function is:

```
bool DtSimManager::createAndDeliverMessage(const DtSimulationAddress &
  recipient, const DtSimInterfaceContent & content);
```

Example 8.1 shows an application sending a create object message to a back-end, telling it to create a new phase line. Since the *DtIfCreateVrfObject* is declared on the stack, it will be deleted automatically.

Example 8.1

```
// Address of the VR-Forces application to send the create message
DtSimulationAddress backendAddr;

 . . .

DtVector start(0., 0., 0.);
DtVector end(0., 100., 0.);

DtIfCreateVrfObject phaseLineContent;

// Set the address of the back-end we want to have simulate the object.
phaseLineContent.setVrfProcessAddress(backendAddr);
phaseLineContent.setObjectName("Phase Line Alpha");
phaseLineContent.addVertex(start);
phaseLineContent.addVertex(end);

mySimManager->createAndDeliverMessage(backendAddr, phaseLineContent);
```

8.2.1. Addressing a Message

The recipient portion of a *DtSimInterfaceMessage* directs the message to a particular VR-Forces application. Specific objects within a VR-Forces application receive messages by registering callbacks for them. Interface messages are addressed to a VR-Forces application by its *DtSimulationAddress* (*hostStructs.h*). The simulation address is the site ID:application number pair that uniquely identifies an application participating in a simulation exercise.

You can broadcast an interface message to all VR-Forces applications by specifying the address *DtSimSendToAll*. It represents the simulation address (site ID: -1 application number: -1), indicating all VR-Forces applications on the network.

In a VR-Forces application, you can get your own simulation address through the Simulation Manager, for example:

```
DtSimulationAddress* address = simManager->messageAddress();

DtInfo("My simulation address site Id: %d application Id: %d\n",
  address->siteId(), address->applicationId());
```

In an application using the Remote Control API (such as a GUI), you can get the simulation address of all VR-Forces applications currently participating in an exercise through the *DtVrfRemoteController*. The *DtVrfRemoteController* class has a backend() member function that returns a *DtList* of simulation addresses of all of the VR-Forces applications on the network in the same session. For example, to print out the addresses of all of the VR-Forces applications currently participating in an exercise:

```
DtListItem* item;

for(item = remoteController->backends().first(); item; item =
  item->next())
{
  DtSimulationAddress* addr = (DtSimulationAddress*)item->data();
  DtInfo("VR-Forces application address %s\n", addr->string());
}
```

8.3. Receiving Interface Messages

Messages are received by registering callback functions for the interface message content types of interest. Objects can register a callback to be fired upon receipt of a given kind of interface message through the *DtVrfMessageInterface*. Message callback functions are specified as:

```
static void messageCallback(DtSimMessage * msg, void * usrData);
```

The callback receives a pointer to a *DtSimMessage* (*simMsg.h*), which is the base class from which *DtSimInterfaceMessage* is derived. (This is because the radio message callback mechanism use the same kind of callback – but with a different derived type of *DtSimMessage*, a *DtVrfObjectMessage* (*vrfObjectMessage.h*).) The message received can be cast to a *DtSimInterfaceMessage* so that its contents can be retrieved. A callback must not delete the messages it receives. The specification for the *DtVrfMessageInterface* member function for adding a callback is:

```
virtual void addMessageCallback(int type, DtMessageCallbackFcn fcn,
  void* usrData);
```

where:

- `type` is the interface message type (message types are in *msgTypes.h*)
- `fcn` is a callback as specified above
- `usrData` is a pointer to the data that will be passed to the callback.

Example 8.2 shows a callback being registered for a *DtIfDeleteVrfObject* with an instance of a *DtSimManager* pointed to by mySimManager. It provides its "this" pointer as usrData.

Example 8.2

```
mySimManager->messageInterface()->addMessageCallback(
  DtIfDeleteVrfObjectType, MyClass::deleteObjectCallback, this);

void deleteObjectCallback(DtSimMessage* msg, void * usrData)
{
  // We know the type of DtSimMesssage is a DtSimInterfaceMessage.
  DtSimInterfaceMessage * ifMsg = (DtSimInterfaceMessage *) msg;

  // We know that the content of the incoming message will be of type
  // DtIfDeleteVrfObject.
  DtIfDeleteVrfObject * deleteVrfObjectInfo =
    (DtIfDeleteVrfObject*)ifMsg->interfaceContent();

  DtInfo("Received delete object message for object %s\n",
    deleteVrfObjectInfo->objectName().string());
}
```

8.4. Creating New Interface Content

Interface messages use *DtSimInterfaceContent* objects to store the message data. If you want to create a new interface message, do the following:

1. Derive a class from *DtSimInterfaceContent*.
2. Give it a unique type() string and a clone() function.
3. Set pertinent parameters.
4. Create a network representation for the class.
5. Implement netRepSize(), setFromNet(), and setToNet() to send and retrieve the network representation.
6. Register the new *DtSimInterfaceContent* with the *DtInterfaceContentFactory*. If NewType is an integer constant that corresponds to your type() function, and NewContentClass::creator is a pointer to your creator function, you can change the *main.cxx* of your front-end and back-end to register the new content as follows:

```
// Front-end:
DtVrfGuiAppEventController::appController()->remoteController()->
  vrfMessageInterface()->factory()->addCreatorFcn(NewType,
  NewContentClass::creator);
// Back-end:
app.cgf()->factoryManager()->interfaceContentFactory()->addCreatorFcn(
  NewType, NewContentClass::creator);
```

7. Insert it into a *DtSimInterfaceMessage*.

8.4.1. Implementing the type() and clone() Member Functions

DtSimInterfaceContent is an abstract class. Any derived class must include a type() member function that returns the type of interface message this content is used for. The type() member function has the following specification:

```
virtual int type() const;
```

A class derived from *DtSimInterfaceContent* needs to provide this type() member function, and should provide mutators and accessors for the information (if any) it will convey. A class derived from *DtSimInterfaceContent* must also provide a clone() member function. The clone() member function is specified as:

```
virtual DtSimInterface* clone() const;
```

Clone functions are typically implemented by calling the class's copy constructor, for example,

```
MyClass::clone()
{
  return new MyClass(*this);
}
```

8.4.2. Setting Parameters

If your new message has additional parameters, add class member variables to represent the data. Make sure the clone() function that you define handles the copying of any new member data. Any member variables that you add are copied to or from a network representation for transmission over the network.

8.4.3. Creating the Network Representation

To make the new message capable of being sent over the network, you must provide a network representation for your class. To do this you must serialize all parameters of your message into a character buffer, and then be able to unserialize them back into the class data members.

You may do this in a number of ways, including making a C-style structure using the MÄK "Net" types, such as DtNetInt32 (defined in *NetTypes.h*). These type manage byte order swapping when necessary to ensure platform independence.

VR-Forces also supplies a number of functions in the DtBufferSerialize namespace for serializing data. This is the recommended way of creating the network representation for your new messages.

If you use a structure, make sure its size is a multiple of 8 bytes to prevent misalignment. You do not need to do this when using DtBufferSerialize.

8.4.4. Implementing netRepSize()

The netRepSize() member function of your class must return the total number of bytes that will be needed to serialize your class in its current state. This may vary depending on the current values of arguments. For example, if you class has a string argument, the size needed to serialize your class will vary depending on the length of the string value.

```
int MyMessageContent::netRepSize()
{
  int size = 0;
  size += DtBufferSerialize::getSize(myInt);
  size += DtBufferSerialize::getSize(myString);
  return size;
}
```

8.4.5. Implementing setFromNet()

The setFromNet() function sets class member data from a network buffer. Using the DtBufferSerialize utility member functions, your member function would look something like this:

```
bool MyMessageContent::setFromNet(const char* contentBuffer, unsigned int
  contentSize)
{
  contentBuffer = DtBufferSerialize::decode(myInt, contentBuffer);
  contentBuffer = DtBufferSerialize::decode(myString, contentBuffer);
  return true;
}
```

8.4.6. Implementing setToNet()

The setToNet() function fills in the network representation of the interface content class from the content arguments. This function must call netRep() at the start to allocate the network buffer (myNetRep), if it has not already been created. Using the *DtBufferSerialize* utility functions, your member function would look like this:

```
bool MyMessageContent::setToNet()
{
  If (!netRep())
  {
    return false;
  }
  char* buffer = myNetRep;
  buffer = DtBufferSerialize::encode(myInt, buffer);
  buffer = DtBufferSerialize::encode(myString, buffer);
  return true;
}
```

It is important that the encode and decode calls be made in the same order in the setFromNet() and setToNet() functions.

Delete the class member myNetRep and set it to NULL any time the size of the net representation changes. If, for example, the length of myStringData changes, delete myNetRep. It will get reallocated to the correct size in setToNet() before it gets used again. Delete myNetRep and set it to NULL in any mutators you create that you know will affect the size of the variable data in the net representation.

For example, if we provide a mutator for myStringData in MyMessageContent:

```
void MyMessageContent::setString (const DtString& string)
{
  // Set the value in our data member
  myStringData = string;
  // Our string may have changed size and therefore the size of
  // the network representation may have changed. Delete myNetRep
  // and set it to NULL. We will recreate it with the correct size
  // when we need it in setToNet().
  if(myNetRep) {
    delete myNetRep;
    myNetRep = NULL;
  } }
```

8.5. Message Classes

This section describes the following message classes and their functions:

- *DtSimMessage*
- *DtInterfaceMessageExecutive*
- *DtMessageExecutive*
- *DtVrfObjectMessageExecutive*
- *DtReportMessage*
- *DtSetDataRequestMessage*
- *DtSimInterfaceMessage*
- *DtVrfObjectMessage*
- *DtTaskMessage*.

The constants in the message classes are defined in *msgTypes.h*.

Message objects do not have factories, because it is assumed that message creators know which message they want to send and will create it directly.

8.5.1. DtSimMessage

DtSimMessage (*simMsg.h*) is an abstract base class that provides common functionality for *DtSimInterfaceMessage* and *DtVrfObjectMessage*. These classes are derived from *DtSimMessage* and implement interface messages and radio messages respectively.

8.5.2. DtInterfaceMessageExecutive

DtInterfaceMessageExecutive (*ifaceMsgExec.h*) manages callbacks on interface messages. The member function indexType() extracts the interface message content type from a *DtSimInterfaceMessage* and returns it as a string.

8.5.3. DtMessageExecutive

DtMessageExecutive (*msgExec.h*) is an abstract base class. Classes derived from *DtMessageExecutive* are used to manage callbacks on particular message types. For example, the radio message executive has a list of callback lists indexed by radio message type.

8.5.4. *DtVrfObjectMessageExecutive*

The class *DtVrfObjectMessageExecutive* (*vrfObjectMessageExecutive.h*) is used to manage callbacks of *DtVrfObjectMessages*. Callbacks can be registered on specified message content types such as Move To Task, or on general message categories such as Task or Report.

8.5.5. *DtReportMessage*

Entities use *DtReportMessage* (*reportMsg.h*) to send *DtSimReports* (*simReport.h*) to one another. The radioMessageType() for this kind of message is DtReportMessageType.

In VR-Forces, only the type DtTaskCompletedReportType of *DtSimReport* is implemented, however you can create your own reports by deriving from *DtSimReport* and implementing the clone() and type() functions. The DtTaskCompletedReportType report message is sent by DtSimLocalEntity::taskComplete() to notify either the entity's superior or the *DtSimManager* of task completion (depending on who issued the task in the first place).

8.5.6. *DtSetDataRequestMessage*

Entities use *DtSetDataRequestMessage* (*setDtaRqMsg.h*) send *DtSetDataRequests* (*setDataReq.h*) to one another. The radioMessageType() for this kind of message is DtSetDataRequestMessageType. Classes derived from *DtSetDataRequest* are used to implement the messages for setting parameters, such as location.

DtSetDataRequestMessages are sent to entities:

- By their superiors
- By set data request statements in a *DtPlan*
- By the user from the GUI.

8.5.7. *DtSimInterfaceMessage*

Use *DtSimInterfaceMessage* (*ifaceMsg.h*) to send an interface message. It derives from a standard *DtSimMessage*, and returns DtInterfaceMessageType as its messageType(). It contains a pointer to a *DtSimInterfaceContent* (analogous to the *DtTask* in a *DtTaskMessage*).

Derived kinds of DtSimInterfaceContent can be categorized as follows:

- Object management
- Object organization
- Plan messages
- Session management
- Radio messages
- Other.

The Remote Control API uses many of these messages to communicate with VR-Forces applications. Exchanging these messages happens transparently within the implementation of the Remote Control API. If you use the Remote Control API, you do not necessarily need to concern yourself with the specific messages to control remote VR-Forces applications. However, if you want to modify or extend the remote control API, you may need to know about some of the underlying messages.

Please see the VR-Forces class documentation for details about individual interface content classes.

8.5.8. *DtVrfObjectMessage*

Messages sent to and from entities are sent using a *DtVrfObjectMessage*. VR-Forces implements four types of VRF Object Messages:

- Set data requests
- Tasks
- Reports
- Administration.

The *DtVrfObjectMessage* class contains the addressing information for a message between entities in the form of the name of the sending entity and the name of the receiver. If the receiver is specified to be “DtBroadcast” the message is considered to be addressed to any entity. Message subclasses contain the actual data of the message.

8.5.9. *DtTaskMessage*

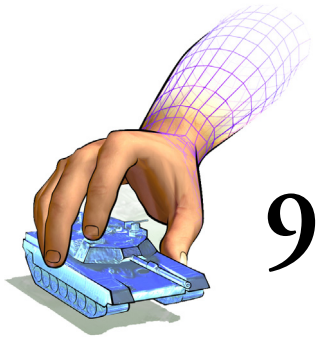
Entities use *DtTaskMessage* (*taskMsg.h*) to send *DtSimTasks* (*simTask.h*) to one another. The radioMessageType() for this kind of message is DtTaskMessageType. Classes derived from *DtSimTask* are used to implement task messages, for example, Move To.

DtTaskMessages are sent to entities:

- By their superiors
- By *DtSimTask* statements in a *DtPlan*
- By the user from the GUI.

8.5.10. *DtAdminMessage*

Entities use *DtAdminMessage* (*adminMessage.h*) to send administration message to each other, or to or from the GUI. These are messages that are not task, set requests, or reports, but which are used for administration of the simulation. The notify level and file logging options in the Object Console tab of the Entity Information dialog box use *DtAdminMessages*.



9

Entity Communication

This chapter describes how to send communications between two or more entities, and the different methods of message transmission available.

9.1. Overview

Entities communicate with each other and with the VR-Forces GUI by sending messages. All inter-entity messages are packaged inside a *DtVrfObjectMessage* (*vrfObject-Message.h*), and sent using one of the message passing systems in VR-Forces.

There are two ways to send a message between entities: the radio message system and the simulation internal message system. Messages that represent radio communication between two entities should be passed though the radio system. Messages that are for simulation purposes only, are sent though the simulation internal system.

All communication within a particular *DtVrfObject* is managed by the *DtVrfObject-CommunicationInterface* class (*vrfObjectCommunicationInterface.h*). This class has member functions for sending messages and for registering callbacks for message receipt.

9.2. Sending Messages

All messages are sent using the following generic procedure:

1. Create and fill in a message content class.
2. Create a *DtVrfObjectMessage* subclass to wrap the content class.
3. Fill in the *DtVrfObjectMessage* subclass with the content and recipient information.
4. Call one of the sendMessage() member functions on the *DtVrfObjectCommunica-tionInterface* of the sending object.

The communication interface fills in the proper data for the message sender.

Messages can be addressed to a specific entity by specifying its name in the recipient field, or addressed to every entity by setting the recipient string to “DtBroadcast”.

Example 1 constructs a message with a *DtTextReport* (*textReport.h*) message content that is ready to send to an object named “receiverObject”.

Example 9.1

```
DtTextReport textReport;
textReport.setText(“text”);
DtReportMessage reportMsg;
reportMsg.setRecipient(“receiverObject”);
reportMsg.setReceiverIsEchelonId(false);
reportMsg.setReport(&textReport);
```

9.2.1. Sending Simulation Internal Messages

To send the message in Example 9.1 using the simulation internal message system, make the following call:

```
vrfObject->communicationInterface()->
  sendSimInternalMessage(&reportMsg);
```

9.2.2. Sending Radio Messages

An object can have multiple radios, so when you send messages using the radio system, you must choose which radio to send the message on. You can specify the radio by name, ID, or as the default radio. The radio name is the string that identifies the desired radio in the entity's OPE file. The radio ID is the index of the radio on the local entity, starting with 0, and reflects the order the radios are defined in the entity's OPE file. The default radio is always the first radio listed in the OPE file, and is equivalent to radio ID 0.

To send the message in Example 9.1 using the default radio, use the following call:

```
vrfObject->communicationInterface()->sendRadioMessage(&reportMsg);
```

To send the message with a named radio, use the following call:

```
vrfObject->communicationInterface()->sendRadioMessage("radio-name",
  &reportMessage);
```

To send the message with a specific radio ID, use the following call:

```
vrfObject->communicationInterface()->sendRadioMessage(2, &reportMessage);
```

The default configuration for most VR-Forces entities has one radio, so it is usually appropriate to send messages using the default radio.

9.3. Receiving Messages

Entities receive messages based on callback functions registered with message processing classes. The message processing class that you register a callback with depends on which message system is being used to transport the message. All message receipt is handled by the *DtVrfObjectCommunicationInterface* class on the *DtVrfObject* that is receiving the message.

9.3.1. Message Receipt Callback Member Functions

The signature for the callback member function registered with any of the message receipt callback lists is:

```
void processMessage(const DtVrfObjectMessage* message, void* usr);
```

As with other callbacks in VR-Forces, this must be a static function. The `void* usr` parameter is an arbitrary pointer that can be passed in when registering the callback, and will be returned as a parameter to the callback function. NULL can be passed if no callback pointer is needed. Typical usage is to pass a pointer to a specific instance of a class, and have the static callback member function call a non-static member of that class when it is invoked.

9.3.2. Receiving Specific Types of Messages

Message receipt callbacks can be registered in the following ways:

- To register a callback that will be invoked when any message is received by an entity, register the callback like this:

```
vrfObject->communicationInterface()->addMessageCallback(
  processMessage, this);
```

- To register for a particular type of message, such as a text report message, register the callback by type:

```
vrfObject->communicationInterface()->
  addMessageCallback(DtTextReportType, processMessage, this);
```

- To register a callback for all messages of a particular category, such as all task messages or all report messages, use the form:

```
vrfObject->communicationInterface()->addMessageCallbackByCategory(
  DtReportMessageType, processMessage, this);
```

9.3.3. Receiving Messages from Specific Sources

Messages arrive at an entity through a specific message source. The source will be either the simulation internal message system, or one of the radios. Callbacks that are registered directly with the communication interface of an entity are invoked when a message arrives from any source. This is the most common way to receive messages, and the way that all internal VR-Forces components work. However, it is sometimes useful to register callbacks directly with one of the source systems within the communication interface. You can do this by querying the communication interface for the desired source object, which will be either a *DtVrfRadio* (*vrfRadio.h*) or the *DtSimInternalCommunicationInterface* (*simInternalCommunicationInterface.h*), and then registering the callback with that object. Callbacks are registered with these objects the same way that they are registered with the *DtVrfObjectCommunicationInterface* object, as described in "Receiving Specific Types of Messages," on page 9-4.

9.4. The VR-Forces Simulation Internal Message System

Messages that are used for internal simulation purposes, which are not simulations of real-world messages, are sent though the simulation internal message system. All *DtVrfObjects* can send and receive simulation internal messages. Examples of this type of message are Set Location messages sent by the VR-Forces GUI when an object is repositioned on the map, or tasks sent by the GUI user. These messages do not have a real-world analog. Simulation internal messages always reach their destination and do not pass though any connectivity checks. They are sent over the network as data interactions or data PDUs. There are no configuration options for the simulation internal message system.

9.5. The VR-Forces Radio Message System

Messages that are sent between entities that represent actual radio traffic are sent though the radio message system. Only other entities that have radios configured on them are able to receive radio messages. An entity can have more than one radio, and messages can be sent and received on any of the radios on the entity.

Radio messages that VR-Forces sends using the radio message system include spot reports and text reports that an entity can be tasked to send from the GUI.

Radio messages are sent over the network inside signal interactions or signal PDUs.

The radio system in VR-Forces has two major pieces: the radio on the entity (*DtVrfRadio* or subclass) and the communication model (*DtCommModel*) (*commModel.h*) associated with the radio. The radio handles message operations for a specific radio on a specific object. When a message is sent from a radio, that radio passes the message to the communication model with which it is associated. The communication model then processes the message, and based on communication model parameters, passes the message to any recipient radios. The radios on the recipient entities then process the message and invoke any callbacks that have been registered on that radio.

Both the radios and the communication models can be configured. For details, please see Section 8.6, "Configuring Radios", and Section 2.4, "Configuring Communication Models" in *VR-Forces Configuration Guide*.

9.5.1. Creating Custom Communication Models

You can create new communication models and install them in VR-Forces. This is useful if you want to define your own connectivity model, or write your own algorithms for determining if radio messages get delivered.

Communication model classes are derived from *DtCommModel* and are generated from the communication model factory based on the string types specified in *./data/simulationModelSets/<model_set>/commModelParams.mtl*. A communication model entry contains parameters for the type of communication model descriptor and communication model class being used.

To create a custom communication model, create a new class derived from *DtCommModel*, or one of its existing subclasses, and implement the sendMessage() member function. This is the member function that is called by a radio when that radio is sending a message. The communication model must then locate other radios on its list of registered radios, and call receiveMessage() on each radio that should receive the message. The communication model can delay the receipt of a message for as long as it likes, and can choose not to deliver a message at all.

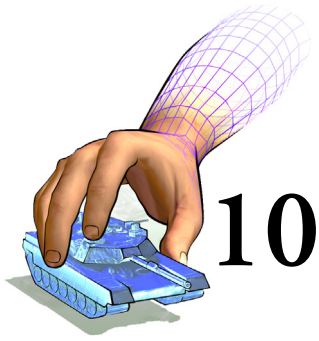
If all you want to do is change the connectivity model of the simple radio communication model, derive from *DtSimpleRadioCommModel* (*simpleRadioCommModel.h*) and implement the checkRadioConnectivity() member function, which determines if two entities are on the same network or not.

After you define the communication model, register it with the *DtCommModelFactory*, as follows:

```
cgf()->factoryManager()->commModelFactory()->addCreatorFcn("custom-comm-
  model", myCommModel::create);
```

Add the new communication model to *commModelParams.mtl*, as follows:

```
(default-radio-model
  (comm-model-descriptor-type "simple-radio-comm-model-descriptor")
  (comm-model-type "custom-comm-model")
   …
)
```



10

Tasks, Sets, Commands, and Reports

This chapter describes classes for tasks, set data requests, simulation commands, and reports, and explains how to add a new task.

10.1. Tasks and Set Data Requests

Entities can be assigned tasks, and their state can be changed using set data requests. You can issue tasks and set data requests from the front-end, as part of plans, and programmatically. Tasks and set data requests are wrapped in data messages and sent over the network to other VR-Forces applications.

10.1.1. Tasks

Tasks represent an action for an entity to perform. Tasks can be sent to entities using *DtVrfObjectMessages* (*vrfObjectMessage.h*). One or more controller components typically register for specific tasks. You can add new tasks by deriving from *DtSimTask* (*simTask.h*) and using the standard factory mechanism. Tasks always contain the task type (task types are listed in *radioMsgTypes.h*) and whatever arguments are needed, such as the name of a control object. Once you have created a new kind of task, you must create a new controller or controllers that can implement that task.

All tasks are derived from the abstract base class *DtSimTask* (*simTask.h*). A task commands an entity to execute a particular action, such as moving to a waypoint or patrolling along a route. *DtSimTask* inherits from the *DtReaderWriter* (*rdrWritr.h*) class. This gives a task message the ability to save itself to and restore itself from a file. To enable the read/write capability, a derived class registers its readable/writable member data items. Classes derived from *DtSimTask* add the parameters needed to completely specify the task, such as the name of the waypoint in a Move To Waypoint task, or the name of the entity to follow in a Follow Entity task.

DtSimTasks also need to be able to be included in *DtTaskMessages*, so they can be sent over the network in the form of radio messages. They also may be included as part of a *DtIfPlan* message, which also is sent over the network. To provide this capability, classes derived from *DtSimTask* implement functions to provide a platform-independent network representation of the task data.

The *DtSimTask* class has a static pointer to an instance of a *DtSimTaskFactory* (*taskFactory.h*). This is a factory for creating new instances of *DtSimTask*, given a string task type. The *DtSimCreator* class, which is responsible for creating all of the various object factories in VR-Forces, creates the *DtSimTaskFactory* instance and then sets a pointer to it in the *DtSimTask* class.

Entities that can respond to and execute tasks, typically register an interest in task messages in their *DtTaskControllerComponents*. For example, ground vehicle entities register for Move To tasks in their *DtGroundMoveToControllerComponent*.

The Task Manager manages the execution of tasks (skipping a task, reporting task completion, and so on). (For details, please see "The Task Manager," on page 4-28.)

10.1.2. Types of Tasks

VR-Forces understands the following types of tasks:

- Task: Any of the named tasks provided with VR-Forces and any new task derived from *DtSimTask*. (New tasks derived from *DtSimTask* cannot be executed from the VR-Forces GUI unless you also extend the GUI to include them.)
- User task: Any task derived form *DtUserTask* (*userTask.h*). *DtUserTasks* can be invoked from the VR-Forces GUI using the generic User Task command.
- Subtask: A task that is part of a parent task.

10.1.3. Subtasks

Subtasking is a mechanism by which a task controller that handles a particular type of task (behavior) can implement that behavior in terms of one or more subtasks. Rather than including the logic for handling all of the behavior in a single controller, parts of the behavior can be managed by different controllers. A task controller can delegate parts of a behavior to other controllers by issuing subtasks to the entity. Through subtasking, developers can write higher-level or more complex behaviors composed of existing behaviors. It can reduce the amount of code a developer has to write, and may help simplify the logic when writing higher-level controllers.

Subtasks can be nested, so it is possible for a task controller to be implemented in terms of one or more subtasks, and these subtask controllers to be implemented in terms of one or more subtasks, and so on.

You do not have to use subtasking when developing new behaviors. Many task controller components in VR-Forces do not issue subtasks (they are for the most part simple behaviors). If your behavior is simple, or you want to keep all of the logic associated with a behavior in one place in the code, then it may be more appropriate not to use subtasking.

10.1.4. How to Use Subtasks

The subtask example (*./examples/subtask*) demonstrates how to develop a task controller that implements a new behavior in terms of subtasks. In the example, a HMMWV is configured with a controller that knows how to handle a new kind of task, a turn-and-move task. The turn-and-move task directs an entity to 1) turn toward a destination point, and 2) move to the point. To implement this new behavior, it has a new type of controller, the *DtTurnAndMoveController*, that responds to turn-and-move tasks, and implements the behavior in terms of a turn-to-heading subtask and a move-to subtask.

To use subtasking in a controller component, you must derive your task controller from *DtTaskControllerComponent* or one of its subclasses.

Starting a Subtask

To issue a subtask, create and fill out the desired *DtTask*, and call sendSubtask(). It returns an integer identifier for the subtask. This identifier will be included in the task-complete report issued by the subtasking controller and is used to determine when a particular subtask is complete.

In the example, the first subtask issued is a *DtTurnToHeadingTask*, issued at the beginning of the turn-and-move behavior. This task is received and executed by another controller (for the entity in this particular example that is the *DtGroundTurnToControllerComponent*).

```
DtTurnToHeadingTask turnTask;
turnTask.setHeading(desiredHeading);

int id = sendSubtask(turnTask);
myProcessState->setSubtaskId(id);
```

In the example, the subtask ID is saved in the process state repository. It is usually a good idea to cache the subtask ID in the process state repository, so it can be saved as part of a scenario or checkpoint.

Determining when a Subtask is Complete

To determine when a subtask is complete, you must listen for the task complete report associated with that subtask. The task-complete report contains the integer identifier returned by the sendSubtask() call when the subtask was issued. To identify the correct task-complete report, check to see that the report was issued by this entity and contains the matching subtask ID.

- Subtask IDs are unique within a single entity, but are not unique between entities. When processing task complete reports for subtasks, make sure the task complete report you are processing is from your entity.
- When you receive a task complete report from your entity with the correct subtask ID, you must remove this subtask ID from the list of pending subtask IDs by calling `void removePendingSubtaskId(int subtaskId)`.

In the subtask example, when the task-complete report for the Turn-to-Heading subtask is received, the controller issues a Move-To subtask. As with the Turn-to-Heading task, it caches the subtask ID and waits to hear when this task is done. When this final subtask is complete, the top-level turn-and-move behavior is considered complete and the controller calls taskComplete(). For details, please see the implementation of DtTaskControllerComponent::processTaskComplete().

Canceling a Subtask

If you want to prematurely stop execution of a subtask for some reason, call clearSubtask(int subtaskId).

- Subtasks are automatically cleared if the top-level task is canceled for some reason (for example, upon receipt of a skip-task message).
- Do not call skipTask() to stop a subtask. skipTask() is intended for top-level (non-subtask) tasks.

To see a demonstration of the turn-and-move behavior, run the VR-Forces GUI as you would normally, but replace the *vrfSim* executable with the subtask executable for the back-end. Load the scenario *./examples/subtask/scenario/turnAndMove.scn*. It contains a HMMWV configured with a simple plan containing a turn-and-move task.

Limitations of Subtasking

An entity can only execute one top-level task at a time. It can however, execute any number of subtasks at the same time.

Subtasks should only be issued by a task controller if the controller is currently executing a (top-level, or non-subtask) task.

Entities should never be tasked to execute more than one type of task at a given point in time. Task controllers are only designed to handle one instance of a given task at any point in time. There are two cases to be considered:

- An entity should never issue a subtask of the same type as the top-level task. When creating behavior composed of nested subtasks, be sure to confirm that no subtasked controllers ever issue subtasks of the same type as the top-level task.
- Subtasks of the same type should never be assigned such that they would be executed at the same time during execution of the behavior. When selecting the task controllers to handle subtasks, note which subtasks they may issue over the lifetime of the task. Make sure that there will be no attempt to execute the same type of subtask at the same time.

It is perfectly acceptable to issue a sequence of the same type of subtask, as long as the previous instance of the subtask is complete before issuing the next one. For example, suppose you have a movement controller that breaks a complicated maneuver down into a sequence of move-to-location behaviors. As long as the task-complete report for a given move-to-location subtask is received before issuing the next move-to-location subtask, having the controller issue multiple subtasks of this same type (move-to-location) is fine.

10.1.5. Set Data Requests

You can set several specific data values for an entity, including:

- Resources
- Position
- Heading
- Sector of responsibility.

Set data requests are very similar in structure and usage to tasks, and are treated as a type of radio message, so controllers can register for set data request messages just as they do for task messages. Set data requests use the standard factory mechanism, so you can add your own.

10.2. Task Messages

Instances of *DtTaskMessage* (*taskMsg.h*) are derived from *DtVrfObjectMessage* (*radioMsg.h*) and are used to convey information about a task. Entities receive tasks on their radio in the form of *DtTaskMessages*. These messages may be sent to an entity as a result of:

- The planning engine executing a task statement in the entity's plan
- A directive from a superior entity
- A task command in the GUI.

DtTaskMessage is one of three types of *DtVrfObjectMessage*. The *DtTaskMessage* class overrides the member function radioMessageType() to return the string type DtTaskMessageType. The radio message types are defined in *radioMsgTypes.h*.

In all cases, the task (a *DtSimTask* object) is packaged inside of a *DtTaskMessage* and sent to the appropriate entity by radio. (For details, please see "Overview," on page 9-2.) The *DtTaskMessage* class maintains its own copy of the *DtSimTask* it is carrying. You set the task in the message by calling its setTask() member function, giving it a pointer to a *DtSimTask*. The *DtTaskMessage* class makes its own copy of the message, so the memory pointed to by the *DtSimTask* that you passed into the *DtTaskMessage* class should be deleted in the calling application code.

10.3. Adding a New Task for an Entity

There are two ways to add a new task for an entity:

- Derive a new task from *DtSimTask*
- Use the *DtUserTask* class to wrap a new task.

If you are using VR-Forces as a back-end only, if you have created your own front-end through which you can configure your own user-defined tasks, or if you are willing to extend the VR-Forces GUI using the GUI API, we recommend that you derive a new task from *DtSimTask*.

If you want to use a new task without extending the VR-Forces front-end, use *DtUserTask* as a wrapper around the new task. It has a simple interface that takes a string task name, which identifies the new kind of task being defined, and up to four string arguments. You must also create one or more controllers to handle a *DtUserTask* containing the new task. Users will be able to specify the new task in the User Task dialog box to task an entity or add it to a plan.

The advantage of deriving a new task from *DtSimTask* is that you can customize the interface to the task class, making the application code cleaner.

10.4. Deriving a New Task from DtSimTask

If you want to derive a new task from *DtSimTask*, you have to complete the following general steps:

1. Derive the new task from *DtSimTask*.
2. Implement *DtSimTask* functions for the task.
3. Register the task with the task factory.
4. Add a controller for the task.
5. Extract task data from the task message.

The first step in creating a new task type is to create a new task class derived from *DtSimTask*. You need to assign it a unique integer type to identify the task (task type names are defined in *radioMsgTypes.h*). The entity's controllers use the task type to determine how to handle an incoming task.

The *DtSimTask* class has the following pure virtual member functions that you must override in the derived task:

- The type() member function returns a unique integer identifying the class type
- The clone() member function must return a pointer to a newly created instance of the derived class
- You must implement the string representation function (stringRep()) to return a human-readable character string representation of the task for use in the plan editor and viewer
- You must implement a static creator function for use by the task factory.

To make your new *DtSimTask* capable of being sent over the network, as part of a *DtTaskMessage* or as part of a *DtIfPlan* message, you must override the following member functions in your derived task:

- The netRepSize() member function returns the size in bytes of the network representation of the task
- The sendToNet() member function writes the member data of the class to a buffer in a form suitable for transmitting over a network in a platform-independent manner
- The setFromNet() member function takes an incoming buffer from a network message and sets the class member data from the network representation.

Once you have defined a new task and have registered the task with the factory, add a new controller to the entity to carry out the action described by the task. To do this, you must derive a new controller from *DtTaskControllerComponent*. (For details, please see "Controller Components," on page 5-5.) The new controller must have a unique string type and must be registered with the corresponding component factory.

In your new task controller component, you must override registerTaskMsgCallbacks() to register your callback function for handling the new task. Once it is registered, your callback function will be called when a message of the new derived type is received on its radio net. In your callback (actually, in a virtual member function of your controller, called from the static callback), extract the necessary data from the task message parameters and start executing the task.

The addTaskSim example demonstrates how to add a new kind of *DtSimTask* to the back-end.

10.5. Handling Unachievable Tasks

The API has functions that let you write code to handle situations in which a task is unachievable. For example, suppose you have a ground vehicle and you give it a task to move to a waypoint, but there is already an entity parked at that waypoint. In this situation, the entity would never be able to successfully reach the waypoint. It would continuously try to reach that point, until it ran out of fuel. Therefore, you would like to add some reasoning to the entity, so that as it approaches the waypoint, it can evaluate the situation and take some more sensible alternative action, such as parking nearby.

Handling unachievable tasks is done on a per *DtTaskControllerComponent* basis. It has member functions, decideToGiveUpTask() and giveUpTask(), that let you add code for handling unachievable tasks. The decideToGiveUpTask() is a function that returns a boolean flag indicating whether or not the entity should give up its current task. It is called immediately after every tick when the simulation is running (in play mode) and entity is tasked. If it returns true, then it calls giveUpTask(). The giveUpTask() function provides a place where any special handling, such as calling taskComplete(), can be added. By default, the function decideToGiveUpTask() returns false, and the giveUpTask() function has an empty implementation.

The decideToGiveUp example demonstrates how to use this feature.

10.6. Adding a User Task

The alternative to deriving a new task from *DtSimTask*, is to use *DtUserTask* (*userTask.h*) message. Use this method if you want users to be able to specify user-defined tasks in the front-end. Instead of deriving a new *DtSimTask* class to convey the data about the new task, you can use the *DtUserTask* message as a wrapper around the new task data. You still need to create your own derived controllers to handle the new task.

10.6.1. Configuring a User Task from the GUI

The *DtUserTask* class has an interface that takes a string *userTaskName* (required), and up to four string arguments (optional). The *userTaskName* string is a unique string used to identify the task.

This string is different from the DtUserTask::type() string used to identify the task message. You can use the *DtUserTask* for any number of user-defined tasks. The *userTaskName* serves to distinguish between the different types of messages wrapped inside the *DtUserTask* message.

The User Task dialog box is used to specify a *DtUserTask*.

10.6.2. Adding a Controller for a User Task

The task described in this section is the Retreat task. The complete code for this example is in *./examples/addTaskSim*. Like all the examples, the directory includes sample project files so that you can build it.

You need to add a new controller to an entity to carry out the task contained by the *DtUserTask*. Since the retreat behavior is just a variation on behaviors already present in *DtGroundAutoControllerComponent*, we derive the new controller from *DtGroundAutoControllerComponent*, and register for a *DtUserTask* message. To do this, override registerTaskMsgCallbacks() to register an interest in incoming *DtUserTask* messages.

The following header file shows how to create a task that is specified through the User Task dialog box:

```
#include "taskCtlrCmpnt.h"
#include "gndAutoCntlr.h"

// String which uniquely identifies this controller
const char TestRetreatControllerType[] = "retreat-controller";

class DtRetreatController : public DtGroundAutoControllerComponent
{
public:
    // constructor
  DtRetreatController(const DtString & name, DtVrfObject* owner,
    DtSimManager * simManager, DtComponentDescriptor* desc = NULL,
    DtReaderWriterRegistry * parentRegistry = 0);

  // destructor
  virtual ~DtRetreatController();

  // copy constructor
  DtRetreatController(const DtRetreatController& orig);

  // assignment operator
  DtRetreatController& operator=(const DtRetreatController& orig);

  // This returns the string TestRetreatControllerType (defined above),
  // used to uniquely identify the type of component.
  virtual DtString type() const;

  // Register a callback for notification of DtUserTask tasks.
  virtual void registerTaskMsgCallbacks();

  // Callback function (registered in registerTaskMsgCallbacks()) for
  // DtRetreat task messages. Calls processRetreatTask() to actually
  // handle the task.
  static void userTaskCallback(DtSimMessage * msg, void * usr);

  // Called by userTaskCallback() when a new DtRetreatTask message is
  // received.
  virtual void processUserTask(DtSimMessage * msg);

  // Update control values
  virtual void tick();

  // This is called by the factory
```

```
  static DtSimComponent* creator(const DtString & name,
    DtVrfObject* owner, DtSimManager * simManager,
    DtComponentDescriptor* desc,
    DtReaderWriterRegistry * parentRegistry);

  // Determine which way to flee based on the current target
  virtual double getHeading();

protected:
  DtReal myCurrentHeading;
  // determined by DtUserTask arg1, please see above
  bool myHoldGround;
};
```

The type() Function

To uniquely identify the new controller, we have to override the type() member function to provide a unique string name for the new component. (Component names are defined in *compTypes.h.*).

```
DtString DtRetreatController::type() const
{
  // Defined in retreatCtlr.h
  return TestRetreatControllerType;
}
```

Registering a Callback on the User Task

This code shows how to register for the *DtUserTask* message on our radio net, providing a callback function for handling *DtUserTask* messages.

```
void DtRetreatController::registerTaskMsgCallbacks()
{
  // Register for any task messsages required by the base class
  DtTaskControllerComponent::registerTaskMsgCallbacks();

  // Specify the task type, the callback function, and user data.
  addTaskProcessorCallback(DtUserTaskType, userTaskCallback, (void *)
    this);

  // Specify the task type, the callback function, and user data.
  addTaskProcessorCallback(DtRetreatTaskType, retreatTaskCallback,
    (void *) this);
}

void DtRetreatController::userTaskCallback(DtSimMessage * msg,
  void * usrData)
{
  DtRetreatController * controller = (DtRetreatController *) usrData;
  if (controller)
  {
    // Call processUserTask to handle the incoming user task message.
    controller->processUserTask(msg);
  }
}
```

Checking the Task Name

In the callback processing function, some additional checking is required to see if the message contained inside the *DtUserTask* is actually the task we are looking for. The *DtUserTask* message can be used for any number of different user-defined tasks, so we need to check the userTaskName() in the callback to determine if the message is really the kind we are looking for.

```
void DtRetreatController::processUserTask(DtSimMessage * msg)
{
  if (msg)
  {
    // Retrieve the task from the task message.
    DtTaskMessage * tm = (DtTaskMessage *) msg;
    DtUserTask * userTask = (DtUserTask *) tm->task();

    // Check to see if the task is a user-defined retreat task. This
    // is the string that is matched against the User Task dialog.
    if(userTask->userTaskName() == "retreat")
    {
      // Extract parameters from arguments
      if (userTask->arg1Text() == "hold ground")
      {
        myHoldGround = true;
      };

      // Call base class method startTask(), to make a copy of our
      // task and to set our tasked flag.
      startTask(*tm);
    }

  }
}
```

The tick() Function

If you define a tick() member function for your component, it gets placed on a tick list for the entity, and is called periodically. You can use this member function to update the control values for the retreat task here. Check the 'tasked' state (**myTasked** member variable) to see if the controller is currently being tasked to determine if updates need to be made.

For examples of how the tick() member function works in controllers, please see "Controller Components," on page 5-5.

10.7. Creating a New DtSetDataRequest

The procedure for creating a new kind of set data request is very similar to creating a new task by overriding *DtSimTask*. For a detailed description of how to create a new task in VR-Forces, please see "Deriving a New Task from DtSimTask," on page 10-7. The analogous steps for creating a set data request are:

1. Create a derived *DtSetDataRequest* (*setDataReq.h*) class. You must implement the following functions in the derived class:
   - Pure virtual type(), clone(), and stringRep() member functions
   - Static creator function
   - netRepSize(), setToNet(), and setFromNet() functions.
2. Register the new set data request with the corresponding factory. Override the *DtSimCreator* class to register the new creator function with the appropriate set data request factory.
3. Add a controller for handling the new set data request. In this controller, you must register a callback for handling the set data request with the Set Data Manager as either a processor or notify callback. Please see "The Set Data Manager," on page 4-30.

The following list summarizes the differences between creating a new *DtSetDataRequest* and creating a new *DtSimTask*:

- When you add the new creator function to the factory in the *DtSimCreator* to register the new creator function, override DtSimCreator::registerSetDataRequest-Creators() to add a new *DtSetDataRequest* creator function.
- In your callback function for handling the set data request, the type of radio message passed into the function is different. You need to cast the incoming *DtSim-Message** to a *DtSetDataRequestMessage** if you are overriding a *DtSetDataRequest.*
- Set data request callbacks are registered with the Set Data Manager instead of the task controller component.

10.8. Simulation Commands

Simulation commands represent an operation for the simulation to perform, such as creating a new entity, or printing a console message. You can use simulation commands in entity plans and global plans. Simulation commands can also be sent directly to a simulation engine using the DtIfExecuteSimCommand interface content message. All simulation commands are subclassed from *DtSimCommand* (*simCommand.h*) and are registered with the *DtSimCommandFactory*.

Similar to tasks and set data requests, each simulation command includes a type identifier and a set of arguments that are used when executing the command. Some of the simulation commands are executed differently depending on whether they are executed by a global plan or by an entity plan. When simulation commands are sent directly to a simulation engine, they are executed as if they are part of a global plan.

Some of the simulation commands that are part of VR-Forces include:

- *DtCreateObjectCommand* creates a new entity or other simulation object.
- *DtConsoleMessageCommand* prints a message to the console. If executed by a specific entity, it prints to that entity's object console.
- *DtDeleteObjectCommand* deletes a simulation object.
- *DtAbandonPlanCommand* causes an entity to abandon the execution of its plan.
- *DtChangeHostilityCommand* changes an entry in the hostility matrix.

10.8.1. Simulation Command Execution

Each *DtSimCommand* subclass has a corresponding subclass of *DtSimCommandExecutor* that carries out the execution of that specific command type. This separates the representation of the command and the logic that executes the command. The command executor is responsible for processing the command and the arguments provided and making the appropriate calls into the VR-Forces API to execute the command. Command executes are registered with the *DtSimCommandExecutorFactory* and must be registered using the same type identifier that was used for the corresponding *DtSimCommand* subclass.

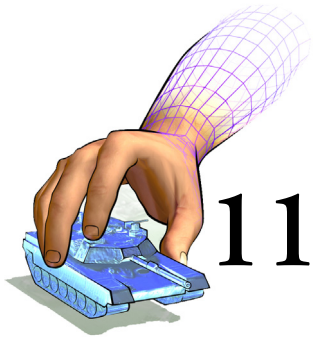
10.8.2. Adding a New Simulation Command

You can add new simulation commands through the VR-Forces API. The addSimCommand example shows how to add a new simulation command to the front-end and back-end.

10.9. Reports

Reports are radio messages sent between entities over the radio net. They are one of the three kinds of radio messages supported in VR-Forces (reports, tasks, set data requests). They are used for exchanging information between entities, such as task status, situation reports, or intelligence. VR-Forces currently supports one type of report, the Task Complete report. Entities use this message to notify the sender of a task message that they have completed the given task.

DtSimReport is the abstract base class from which all report classes should be derived. In order to send the report, it gets placed inside of a *DtReportMessage* object. The *DtReportMessage* class is used to convey a *DtSimReport*. Components register callback functions for report messages in the same way that they do for task and set data messages.



11

Plans

This chapter describes the Plan Manager, plan classes, and how to extend the planning features provided with VR-Forces.

11.1. Introduction

A plan is a set of tasks, set data requests, and global commands that an entity carries out in order. The plan for an entity is maintained by the entity's *DtVrfObjectPlanManager* (defined in *vrfObjPlanMgr.h*).

The *DtVrfObjectManager* for each back-end has a *DtVrfObjectPlanAdministrator*. This class reads and writes the plan (*.pln*) files when scenarios are loaded and saved. It also acts as a dispatcher, forwarding plan messages such as Abandon Plan to the *DtVrfObjectPlanManager* for the recipient of the message.

It is likely that you are reading this chapter for one of the following reasons:

- You want to create or manipulate plans programmatically. Read all of this chapter.
- You want to derive new conditions or conditional expressions. Read "Adding a New Type of Conditional Expression," on page 11-18.
- You are overriding task or set statements. If so, you do not need to touch the Plan Manager or make changes to plan classes. You can read this chapter for fun.

11.2. The Plan Manager

Each *DtVrfObject* has a *DtVrfObjectPlanManager*, which is responsible for maintaining the *DtPlan* for that object. When the *DtVrfObject* is ticked, it ticks its *DtVrfObjectPlanManager*, which ticks the *DtPlan*.

The Plan Manager acts as an interface between the plan and the Task Manager. When the Task Manager is ready to execute a task, it calls DtPlan::awaitingTask() through the Plan Manager. Whenever the Task Manager executes a task that is not a subtask, it calls DtVrfObjectPlanManager::taskExecuted(). This allows the Plan Manager to determine if the plan should be abandoned due to an immediate task issued by the front-end, or a task issued from a global plan.

The Plan Manager responds to plan-related commands from front-end applications that are forwarded from the *DtVrfPlanAdministrator*.

Plan Managers get created in a factory. For details about how Plan Managers get created, please see "How the Object Manager Chooses an Object's Subcomponents," on page 3-7.

11.3. Managing Plans

A *DtPlan* object represents the plan for an individual *DtVrfObject*. In VR-Forces, there is one *DtPlan* created for each *DtVrfObject* that needs one. If an entity does not have a plan, no *DtPlan* gets created. The plan for an aggregate entity is associated with the *DtVrfObject* of the aggregate, and individual members of the aggregate might not have plans of their own.

The *DtCgf* class provides member functions for performing some of the more common operations for managing plans. These include functions for loading and saving scenarios (plans are implicitly loaded and saved as part of these functions), and a member function for assigning new plans to entities. If you require more control over plans, you can access the Plan Manager for an object by calling DtVrfObject::vrfObjectPlanManager().

11.3.1. Loading a Plan File

To load a plan file into the Plan Manager, call DtVrfObjectPlanAdministrator::readPlanFile(). This instantiates a *DtPlan* object for each plan in the file and assigns it to the object's *DtVrfObjectPlanManager*. This is call should generally only be made from DtCgf::loadScenario().

11.3.2. Saving Plans to a File

To save the plans to file, call DtVrfObjectPlanAdministrator::writePlanFile(). This call should generally only be made from DtCgf::saveScenario(). When you save a scenario, a plan file is one of the files that gets saved as part of the scenario. The name of the plan file is referenced in the scenario file.

11.3.3. Creating a New Plan Using API Calls

The simplePlan example (in *./examples/simplePlan*) demonstrates how to create and assign a plan to an entity. In *main.cxx*, a simple plan consisting of a Set Speed request, a Wait Duration task, and a Move To Waypoint task, is created and assigned to a tank.

Creating and Initializing a New DtPlan

To create a new *DtPlan*, you can call the *DtPlan* constructor with no arguments. You can use this form of constructor when assigning plans through the *DtCgf* class. The name of the object passed into DtCgf::assignPlanByName() gets assigned to the *DtPlan*. You must call DtPlan::init() before you can use any other member functions in the *DtPlan* class. The init() function completes construction of the *DtPlan* object.

Adding Statements to a DtPlan

Plan statements are the individual items that get executed in a *DtPlan*. Statements represent tasks for the entity to perform, set data requests, or control-flow statements such as if(conditional expression)-then-else and when(conditional expression). They are derived from the *DtSimStatement* class (*simStmt.h*). In the simplePlan example, we create task and set data request statements. To create these kinds of statements, we first created the content for the statement (a derived type of *DtTask* or *DtSetDataRequest*), and then assigned it to a newly created corresponding *DtTaskStmt* or *DtSetDataRequest-Stmt* class.

A *DtPlan* contains a *DtPlanBlock*, which is the list of *DtSimStatements* in a plan. To add our new statements to the plan, we first obtained a pointer to the *DtPlan*'s main block of statements, and then added the statements to it. Because the plan assumes responsibility for deleting the memory associated with the plan statements, we cloned in the statements before adding them.

Assigning a DtPlan

Once you have created a *DtPlan*, you need to assign it to a particular *DtVrfObject*. The assignPlanByName() function associates the given *DtPlan* with the given *DtVrfObject*, referenced by name. The *DtPlan* will get executed as its *DtVrfObjectPlanManager* gets ticked.

11.3.4. Accessing the Plan for an Object

To look up the plan for an object, use the DtVrfObjectPlanManager::plan() accessor. If the object has not been assigned a plan, a NULL pointer is returned. For example, given a pointer to a *DtVrfObject* "myObject", the following code prints a message if the plan is complete. For information about accessing a certain *DtVrfObject*, please see "Looking Up Individual Objects," on page 3-31.

```
DtPlan* plan = myObject->vrfObjectPlanManager()->plan();
if (plan && plan->isComplete())
{
  DtInfo("Plan completed\n")'
}
```

11.4. The Plan Administrator

The *DtVrfObjectManager* class has an instance of a *DtVrfObjectPlanAdministrator*. Therefore there is one *DtVrfObjectPlanAdministrator* in each VR-Forces back-end application.

The DtVrfObjectPlanAdministrator::writePlanFile() member function is used by the *DtCgf* to create plan files when a scenario is saved and DtVrfObjectPlanAdministrator::readPlanFile() is used to load them when the scenario is loaded.

It is also used by the *DtCgf* during scenario save and load to control the execution state of plans. DtVrfObjectPlanAdministrator::disablePlanExecutionState() goes through each local object in the *DtVrfObjectManager* and calls disableExecutionState() for the *DtPlan* owned by the object's *DtVrfObjectPlanManager*. DtVrfObjectPlanAdministrator::enablePlanExecutionState() works the same way to enable plan execution.

The *DtVrfObjectPlanAdministrator* also acts as a dispatcher to read plan messages sent by a *DtVrfRemoteController* and forward them to the proper *DtVrfObjectPlanManager* based on the recipient of the message. It handles the following message types:

- DtRequestPlanMessageType
- DtPlanMessageType
- DtAbandonPlanMessageType.

11.5. Examining and Changing Plans

Plans are a list of ordered statements that are issued to an entity for execution. If you want to extend or modify plans programmatically, it is helpful to understand how the *DtPlan* class and related classes are structured.

If you are just adding a new kind of task or set data request statement, you do not have to make any plan-specific changes. The *DtTaskStmt* and *DtSetDataRequestStmt* plan statement classes accommodate new types automatically. Please see "Adding a New Task for an Entity," on page 10-7 and "Creating a New DtSetDataRequest," on page 10-14.

11.5.1. Iterating Through the Statements in a Plan

A plan contains a top-level "Main" block of *DtSimStatements* (*simStmt.h*). A block is a *DtSimBlock* (*simBlock.h*) object that contains a list of *DtSimStatements*, in the order they are to be executed. (The *DtPlan*'s main block is actually a derived type of *DtSimBlock*, a *DtPlanBlock* (*planBlock.h*).) The following is an example of a plan.

```
(Plan
  (plan-name "15 Plt, 1 Force")
  (Block
    (Set
      (set-data-request-type "set-heading")
      (heading  1.570796)
    )
    (Task
      (task-type "move-to")
      (control-point  "white")
    )
  )
)
```

The *DtSimBlock* provides accessors for looking up *DtSimStatements* by an integer statement ID, as well as iterating through the statements in the block. For example, to iterate through the main block of statements and display information about each one:

```
DtPlan* aPlan;

DtSimStatement* aStmt;

. . .

for (aStmt = aPlan->mainBlock()->firstStmt(); aStmt;
  aStmt = aStmt->nextStmt())
{

  // stringRep() returns a string representation of the statement
  // (as seen in the view and edit plan windows)
  DtInfo("String representation for statement %d: %s\n", aStmt->id(),
    aStmt->stringRep().string());
}
```

This example iterates through just the top-level statements in a plan, that is, the list of *DtSimStatements* contained in its main block of statements. Statements may contain subblocks of statements, such as the set of statements that gets conditionally executed inside `If` and `When` statements. To access the statement block in a *DtIfStmt* (*ifStmt.h*), call the thenBlock() and elseBlock() member functions. To access the block of statement block in a *DtWhenStmt* (*whenStmt.h*), call the whenBlock() member function. All of the subblock accessors return pointers to derived types of *DtSimBlocks*. You can treat these blocks in the same way as you would the *DtPlanBlock*, that is, you can get the count of statements contained in the block, look up a statement by its integer ID, and iterate through the statements in the block.

11.5.2. Modifying a Plan Programmatically

You can create and edit plans through code. You might want to do this if, for example, you want to create or modify a plan in the back-end (instead of reading it in from a file).

Adding Statements to a Plan

To add a statement to a plan, you add a statement to the appropriate block of statements in the plan. The block may be the main block of statements owned by the *DtPlan* (a *DtPlanBlock*, returned by the mainBlock() accessor()), or it may be the subblock of an `If` (*DtIfStmt*) or `When` (*DtWhenStmt*) statement. *DtIfStmts* contain thenBlock() and elseBlock() member functions, which return pointers to its subblocks. `When` statements contain a whenBlock() member function pointing to its single subblock. Using these accessors, you can add a statement to the appropriate place in a plan.

For example, to create a `When` statement that contains a single task statement, and then add the `When` statement (containing the task statement in its subblock) to the plan:

```
DtPlan* aPlan;
DtWhenStmt *whenStmt;
DtTaskStmt *taskStmt;

// Add the task statement to the When statement's subblock
whenStmt->whenBlock()->add(taskStmt);

// Add the When statement (containing the task statement) to the main plan
// block (making it a top-level statement)
aPlan->add(whenStmt);
```

The add() call adds the statement to the end of the list of statements in the block. To add a statement at the beginning of the list call addToStart(). To insert a statement after a statement already in the list, call addAfter().

When you add a statement to a block, the block assumes responsibility for deleting the memory associated with the statement, so you may want to clone the statement being passed in (if you expect to continue to need the statement outside the block).

Finding a Statement

When you create a new statement, its constructor automatically generates a statement ID (a unique integer identifier). The *DtSimStatement* class has a statementId() accessor that returns this ID. You can use the statement ID to look up a particular statement in the main block of a plan, or in a statement with subblocks. To look up a statement with a specific ID in a plan's main block of statements, do the following:

```
int stmtId;

DtSimStatement* stmt = aPlan->mainBlock()->lookupStatement(stmtId);

if(stmt)
{
  DtInfo("Found statement %d\n", stmt->statementId());
}
```

When you call lookupStatement() in a block (a plan block, or a subblock in a statement), it only searches for the statement in that block. It does not search recursively through the statements' subblocks.

Removing Statements from a Plan

To delete a statement from a plan, call the removeAndDelete() member function in the *DtSimBlock* that owns the statement. The removeAndDelete() function removes the statement from the block, and deletes the memory associated with the statement.

To delete a top-level statement in a plan, delete the statement from the plan's main block of statements, for example:

```
DtPlan* aPlan;
DtSimStatement* aStmt;

aPlan->mainBlock()->removeAndDelete(aStmt);
```

To delete a statement in a subblock of an `If` or `When` statement, call removeAndDelete() in that subblock.

To remove and delete all of the statements in a plan, call removeAllAndDelete() in the plan's main block, for example:

```
DtPlan* aPlan;

aPlan->mainBlock()->removeAllAndDelete();
```

11.6. Executing a Plan

To execute a *DtPlan*, its tick() function must be called. *DtPlans* are ticked by the *DtVrfObjectPlanManager* that owns them. When the *DtCgf* is ticked, it ticks the *DtVrfObjectManager*, which ticks each *DtVrfObject*. DtVrfObject::tick() ticks the *DtVrfObjectPlanManager*. The remainder of this section discusses the details of how a *DtPlan* advances through its statements during execution.

11.6.1. The Initial Execution State

A plan's current statement is returned by the DtPlan::currentStmt() member function and is actually just the top entry on its execution stack (myExecutionStack). The execution stack is a list of *DtStmtIndex* (*stmtIndex.h*) objects. (*DtStmntIndex* is a type of smart pointer used to reference *DtSimStatements*.) If the execution stack is empty, currentStmt() returns an empty statement index. This is the initial state of a plan prior to execution.

11.6.2. Starting the Plan

Execution of a plan begins when the *DtSimTaskManager* accesses the plan for an object through its *DtVrfObjectPlanManager* and calls awaitingTask() on it. This starts the following process:

1. The myAwaitingTaskFlag is set inside the *DtPlan*.
2. If it is at the start of the plan (the plan is not complete, the current statement on the execution stack is NULL, and there is a valid main block), the processAwaitingTask() calls executeBlock() to start execution of the main block (myMainBlock, a *DtPlanBlock*) of statements. The executeBlock() function does the following:
   a. Calls setIsComplete(false), initializing the execution status of the plan to 'not complete'.
   b. Calls setTaskRequestPending(true), indicating that the plan is waiting for a task request (an awaitingTask() call) from the entity.
   c. If the main block of statements is empty, it calls the block's DtPlanBlock::exitFromBlock() function. The plan's execution state is set to complete. If there are statements in the main block, it calls setCurrentStmt() with the first statement in the block.
   d. Calls executeStatement(). This calls the current statement's DtSimStatement::execute() function. If executeStatement() did not change the task request pending status (that is, it did not send a task to an entity), it calls continuePlan() to advance plan execution.

Each derived type of *DtSimStatement* implements execute() differently. The *DtSimTask* and *DtSetDataRequestStmts* execute() functions send *DtTask* and *DtSetDataRequest* statements to the entity, respectively. *DtWhenStmts* register themselves as triggers in the *DtPlan* when they are executed. As a trigger, the conditional expression in a `When` statement gets evaluated every tick by the *DtPlan*. When the condition evaluates to `true`, then it executes its block of statements. Triggers are discussed in more detail in "Triggers (or When Statements)," on page 11-12 and "Triggers," on page 11-21.

11.6.3. Advancing Through a Plan

The continuePlan() member function continues the execution cycle. The first time through it is called as a result of the executeBlock() member function being called on the plan's main block. (For details, please see "Starting the Plan," on page 11-10.) The protected DtPlan::continuePlan() member function continues the plan execution cycle until either the myTaskRequestPending flag is cleared, or the myIsComplete flag gets set. The execution cycle consists of alternating calls to DtPlan::stepStatement() to step the current statement pointer to the next statement, and DtPlan::executeStatement() to execute it. The loop begins with a step because the plan's current statement actually indicates the last statement executed – the one that sent the entity its previous task.

In continuePlan(), the execution cycle proceeds as follows:

1. If the plan is complete, we are done, so return.
2. Initialize the task request pending state to TRUE.
3. While there is still a task request pending, call stepStatement() to advance to the next statement in the plan.
4. If the plan is complete or there is no task request pending, we are done, so return.
5. Call executeStatement(). This calls the current statement's execute() function.

When an entity requests a task from its plan, the plan sets its internal **myTaskRequestPending** flag, which remains `true` until a statement is executed that causes a task to be sent back to the entity. The task pending flag is used to decide when to halt the execution cycle in the member function DtPlan::continuePlan().

11.6.4. Completing a Plan

When an entity reaches the end of its plan, the plan's **myIsComplete** flag is set to `true`. This halts further execution of the plan's statements. However, pending triggers are still tested and can still fire, causing execution to resume at the start of the trigger block.

11.6.5. Triggers (or When Statements)

`When` statements are special kinds of statements, known as triggers. When a `When` statement gets executed (DtWhen::execute() is called), it creates a *DtSimTrigger* object and registers it with the plan as a pending trigger by calling DtPlan::addTrigger(). Each plan keeps a list of pending triggers (**myPendingTriggers**). In its tick() function, it periodically checks their associated conditions to determine if they should be fired. If a trigger fires, the plan must keep track of which statement was interrupted so that after the trigger block completes, execution can resume with the statement following the interrupted statement. It does so using a stack of statement index objects (**myExecutionStack**), stored in a *DtList*.

11.6.6. Abandoning a Plan

You can have an entity abandon execution of its plan by accessing its *DtPlan* through its *DtVrfObjectPlanManager* and calling abandonPlan() on it.This results in the plan setting its status to complete.

11.7. Statements

This section provides additional details about plan statements.

Statements are implemented by objects derived from *DtSimStatement*. VR-Forces has the following kinds of statements: task statements, set data request statements, `If` statements, `While` statements, and `When` (trigger) statements.

- All Task statements are handled by *DtTaskStmt*. To add a new kind of task, you do not have to create a new statement class. For more information about creating tasks, please see "Adding a New Task for an Entity," on page 10-7.
- All set data request statements are handled by *DtSetDataRequestStmt*. To add a new kind of set data request, you do not have to create a new statement class. For more information about creating set data requests, "Creating a New DtSetDataRequest," on page 10-14.

Each *DtSimStatement* object represents a single statement in a plan (but not necessarily a single line in a Plan window.) Some kinds of statements have subblocks which contain other statements (for example, `If`), and these compound statements have one extra line per subblock in the Plan window (for example, `else` and `endif`).

DtSimStatements are created by either the statement factory or by cloning another statement. Normally, the parent *DtSimBlock* creates its statements when it reads a plan file, or they are created in the front-end and passed to the back-end as part of a complete *DtPlan* object.

Each *DtSimStatement* object is assigned a unique, non-zero statement ID number when it is created. This number is preserved in all copies of the original statement and is used to match up statements when comparing two (potentially modified) clones of the same original *DtPlan*. *DtPlans*, *DtSimStatements*, and *DtSimBlocks* all provide ways to help search for a statement with a particular statement ID.

Each *DtSimStatement* objects in a plan must have a unique statement ID value.

DtStmtIndex objects use statement ID values to hide the difference between a direct *DtSimStatement* pointer and a combination of a statement ID and a *DtPlan* pointer. Statement objects are normally accessed via *DtStmtIndex* objects for portability across clones of the surrounding *DtPlan*.

Statement objects do not have direct pointers to their *DtPlans*. The execute() and step() member functions are passed a *DtPlan* reference as a "context of execution". Statements can use this context to look up information in the *DtPlan*, send out messages to the *DtPlan*'s associated entity, or modify the *DtPlan*'s state (such as the current statement index).

The *DtSimStatement* class has a static pointer to an instance of a *DtStatementFactory*. This is a factory for creating new instances of *DtSimStatement*, given a string statement type. The *DtSimCreator* class, which is responsible for creating all of the various object factories in VR-Forces, creates the *DtStatementFactory* instance and then sets a pointer to it in the *DtSimStatement* class.

11.7.1. Statement Blocks

Each block of statements is represented by an object of a class derived from *DtSimBlock*. *DtSimBlock* is a *DtRwIntrusiveList* (*rwIntList.h*) of *DtSimStatement* objects.

Derived block objects are used in VR-Forces as each plan's Main Block, `When` Statement trigger blocks, and as `Then` or `Else` blocks in `If` statements.

Since the various derived block types are closely tied to, and created only by their associated *DtPlan* or *DtSimStatement* types, there is no separate factory mechanism for them.

The base *DtSimBlock* type does not contain any data members, but derived classes usually contain a pointer to their parent object (*DtSimStatement* or *DtPlan*).

Classes Derived from DtSimBlock

VR-Forces provides three classes derived from *DtSimBlock*. Each class has a pointer back to its parent *DtSimStatement* or *DtPlan*, and the only other significant difference is what they do when their exitFromBlock() member function is called.

11.8. Conditional Expressions

This section provides details about the different types of conditional expressions. The first two subsections discuss *DtSimCondExpr* and *DtEvaluator*. These objects are created by factories in VR-Forces. You can extend the set of existing conditional expressions by creating and registering new subclasses with the appropriate factories. For more information, please see "Adding a New Type of Conditional Expression," on page 11-18.

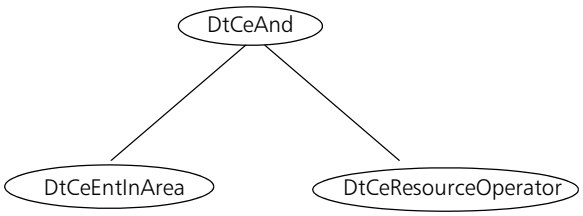
11.8.1. Conditional Expression Objects

DtSimCondExpr (*condExpr.h*) objects represent conditional expressions. They are used to test the properties of both individual entities and aggregates.

A simple conditional expression can be represented by just a single object. More complicated expressions are formed as tree structures of simpler expressions (sub-expressions) combined by the operators: AND, OR and NOT, which are themselves conditional expressions.

```
For example:
AND(Entity-In-Area Entity:"1 M1A2, 1 Force" Area:"base camp",
  resource(fuel) < 50%)
```

would be represented in objects as:



When `If` statements or `When` statements want to test the current state of a conditional expression, they call the evaluate() member function, which recursively evaluates all operators and their sub-expressions.

11.8.2. Conditional Expression Evaluators

DtSimCondExprs are evaluated by calling their evaluate() member functions. The *DtSimCondExpr* class does not actually perform the evaluation of the expression itself. Instead, in its evaluate() function, it uses a *DtEvaluator* object to actually perform the test. In the VR-Forces back-end, *DtSimCondExpr*'s are always configured with a corresponding *DtEvaluator* class (a *DtEvaluator* is always created and set in a *DtSimCondExpr* class, whenever a *DtSimCondExpr* is created). *DtSimCondExprs* that are configured with a *DtEvaluator* invoke the *DtEvaluator*'s evaluate() member function whenever its evaluate() function is called. Each *DtEvaluator* subclass is designed to work with a particular *DtSimCondExpr*, so there is a one-to-one correspondence between *DtSimCondExpr* subclasses and *DtEvaluator* subclasses.

In a remote application (an application that uses the VR-Forces Remote Control API, such as a GUI), conditional expressions are created and used without *DtEvaluators*. They only need the data aspect of a conditional express to be able to do things such as display and edit plans. They never need to actually evaluate the expression, so the *DtSimCondExpr*'s evaluate() function is never invoked. They only need to hold the parameters associated with a particular conditional expression.

11.8.3. Logical Constants

The two logical constants (True and False) are implemented by the same class: *DtCeConstant* (*ceConstant.h*). It stores its value as a `bool` flag. The *DtCeConstant* class is configured with the *DtEvalConstant* (*evalConstant.h*) evaluator.

11.8.4. Logical Operators

The three logical operators (*DtCeAnd* (*ceAnd.h*), *DtCeOr* (*ceOr.h*), and *DtCeNot* (*ceNot.h*)) are very similar. They only differ in:

- The operation they perform.
- Their readable string representation.
- The number of sub-expressions they have. (The NOT operator only has one subexpression.)

Each logical operator stores its sub-expressions as pointers to other *DtSimCondExpr* objects, which could also be logical operators. The AND and OR operators sub-expressions are called leftCondExpr() and rightCondExpr(). The NOT operator sub-expression is called condExpr().

The following evaluators are configured with the three logical operators:

- The *DtCeAnd* class is configured with a *DtEvalAnd* (*evalAnd.h*) evaluator
- The *DtCeOr* class is configured with a *DtEvalOr* (*evalOr.h*) evaluator
- The *DtCeNot* class is configured with a *DtEvalNot* (*evalNot.h*) evaluator.

11.8.5. Resource Operators

The *DtCeResourceOperator* (*ceResourceOp.h*) class represents a relational comparison (for example, <, >) between an entity's resource value and a floating point constant. Each *DtCeResourceOperator* contains a:

- Resource name string
- Relational operator as a string
- Real-number comparison value
- Flag that indicates whether the comparison value is a percentage.

When the evaluate() function is called, the resource operator looks up the named resource in the localState() of the entity associated with the plan context.

If the **percentFlag** flag is true, the resource's percentageFull() value is used, otherwise the resource's amount() value is used. The resource value is then converted into a real number and compared to the specified comparison value, using the specified relational operator.

The supported operators are the same as in C++:

- < (less than)
- <= (less than or equal to)
- == (equal to (= is also accepted))
- != (not equal to)
- >= (greater than or equal to)
- > (greater than).

The *DtCeResourceOperator* class is configured with the *DtEvalResourceOp* (*evalResourceOp.h*) evaluator.

11.8.6. Conditional Expressions for Testing Entity Status

VR-Forces comes with several conditional expressions for testing entity status, such as:

- Entity Destroyed (*DtCeEntDestroyed* (*ceEntDstroyed.h*))
- Entity In Area (*DtCeEntInArea* (*ceEntInArea.h*))
- Entity Left Of Line (*DtCeEntLeftOfLine* (*ceEntLOfLine.h*)).

Each test is given an entity name string (the string returned by DtVrfObject::objectName()) and an optional modifier string. (Currently only the "ANY" modifier is implemented.) The name string is the name string for VR-Forces entities, and is the marking text for non-VR-Forces remote entities. The tests take other test-specific input.

The "ANY" modifier changes the way a test operates when it is applied to an aggregate or the force level. When ANY is not applied, a test is true only if it is true for all members of the aggregate. Therefore, the test terminates the first time it sees an entity that does not match the criteria. With the "ANY" modifier, a test is true if any one member of the aggregate satisfies the condition. Therefore, the search terminates the first time it finds an entity that matches the criteria.

Conditional tests can be applied to:

- A single individual entity
- An aggregate
- All the individual members of an aggregate (with the "ANY" flag)
- All local or remote entities that directly report to a specific Force (top-level entity)
- All local or remote entities of a specific Force (with a top-level entity and the "ANY" flag).

To apply their tests to the appropriate combination of entities, the conditional expression entity tests use matching classes derived from *DtSimSubordinateSearch* (*subSearch.h*) to do the actual evaluate() work. *DtCeEntDestroyed* uses the *DtEntDestroyedSearch* (*entDstroySrch.h*) class; *DtCeEntInArea* uses the *DtEntInAreaSearch* (*entInAreaSrch.h*) class; and *DtCeEntLeftOfLine* uses the *DtEntLeftOfLineSearch* (*entLOLineSrch.h*) class.

- The *DtCeEntDestroyed* class is configured with the *DtEvalEntDestroyed* (*evalEntDestroyed.h*) evaluator
- The *DtCeEntInArea* class is configured with the *DtEvalEntInArea* (*evalEntInArea.h*) evaluator
- The *DtCeEntLeftOfLine* class is configured with the *DtEvalEntLeftOfLine* (*evalEntLofLine.h*) evaluator.

11.8.7. The Random Operator

The random logical operator makes a random draw between 0 and 1, and compares the result against a given real probability parameter (if the result is less than the parameter, return True, otherwise returns False). It is implemented by the class *DtCeRandom* (*ceRandom.h*). It stores a *DtReal* probability parameter.

The *DtCeRandom* class is configured with the *DtEvalRandom* (*evalRandom.h*) evaluator.

11.8.8. Adding a New Type of Conditional Expression

To add a new kind of conditional expression to VR-Forces, you must create two new derived classes: a new conditional expression data class (*DtSimCondExpr*), and a new conditional expression evaluator class (*DtEvaluator*). The conditional expression data class represents the (optional) data parameters used in your conditional expression (for example, the name of an area to test in the entity-in-area class, *DtCeEntInArea*). The evaluator class contains an evaluate() member function that performs the test for the given *DtSimCondExpr* (for example, the *DtEvalEntInArea* class tests for the entity in area condition for a given *DtCeEntInArea* object).

Adding a New Conditional Expression Data Class

To add a new kind of conditional expression data class, you must derive a new class from *DtSimCondExpr* (*condExpr.h*) or one of its subclasses. In your new class you must provide the member functions described in Table 11-1.

Table 11-1: *DtSimCondExpr* functions

Function	Description
type()	Returns an integer uniquely identifying the kind of conditional expression. Choose a type that does not conflict with any existing types (existing types are defined in *condExprTypes.h*).
clone()	Returns a newly allocated copy of this object.
netRepSize()	Returns the size of the network representation of the object.
stringRep()	Returns a string representation for the expression.
setFromNet()	Sets the member data in the class from a network representation of the class.
setToNet()	Maps the member data in the class to the network representation for the class.
creator()	Returns a newly created instance of the class.

You must implement the netRepSize(), setToNet(), and setFromNet() member functions to make your conditional expression capable of being sent over the network. For examples and a discussion of how to implement these functions, please see "Creating the Network Representation," on page 8-7.

After you create your new *DtSimCondExpr* class, you must register your new conditional expression class with the conditional expression factory. Adding it to the factory makes the class available for use any time the new type is encountered (such as when reading from a plan file, or when receiving a plan containing the new expression over the network). For more information about how to register your new conditional expression class with the factory, please see "VR-Forces Factories," on page 2-10. Registration of your new *DtSimCondExpr* will look something like this:

```
cgf.factoryManager()->condExprFactory()->addCreatorFcn(MyCondExprType,
  MyCondExpr::creator);
```

where MyCondExpr is your new *DtSimCondExpr* class, and MyCondExprType is the new integer type you return in your implementation of MyCondExpr::type().

Adding a new Conditional Expression Evaluator

To add a new evaluator class for your new conditional expression, you must derive a new class from *DtEvaluator* (*evaluator.h*) or one of its subclasses. In your new class you must provide the member functions described in Table 11-2.

Table 11-2: Functions required for conditional expressions

Function	Description
evaluate()	Evaluates the given *DtSimCondExpr*.
clone()	Returns a newly allocated copy of this object.
creator()	Returns a newly created instance of the class.

Just as with the new *DtSimCondExpr* class, you must register the new evaluator class with the evaluator factory. When you register the new evaluator class type, you must register it with the same integer type that you assigned to the new *DtSimCondExpr* class, for example:

```
cgf.factoryManager()->evaluatorFactory ()->addCreatorFcn(MyCondExprType,
  MyEvaluator::creator);
```

where MyEvaluator is the new *DtEvaluator* class, and MyCondExprType is the integer type you return in your implementation of MyCondExpr::type().

The evaluate() function is where you implement the conditional expression test you want to perform. The evaluate() function takes a *DtSimCondExpr** as one of its arguments. If you have correctly registered your new *DtEvaluator* class with the evaluator factory using the same integer type as your new *DtSimCondExpr*, you can safely downcast the incoming base *DtSimCondExpr** to your new derived class type.

You must register the new *DtEvaluator* class with the factory using the same integer type as you use when registering your new *DtSimCondExpr* class. Failure to do so will result in undefined behavior.

Once you create and register your new *DtSimCondExpr* and *DtEvaluator* subclasses, you can use the new expression in your simulation. VR-Forces takes care of creating and configuring your new conditional expression and its corresponding evaluator whenever it encounters them (such as when reading in plan files, or when receiving plans over the network).

11.9. Triggers

Trigger (*DtSimTrigger*, in *simTrigger.h*) objects are created by `When` statements (*DtWhenStmt*) when they are executed.

Triggers register with a *DtPlan* object so that they can get ticked. They keep a pointer to the plan so that they can unregister when they fire and are deleted.

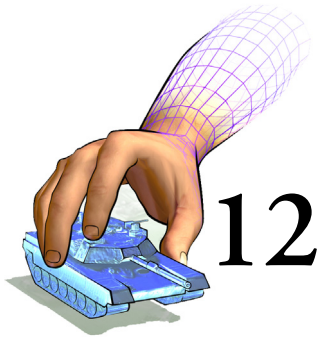
When a *DtSimTrigger* is created, it saves a pointer to the `When` statement that created it. When DtSimTrigger::tick() is called, the trigger uses the *DtWhenStmt* statement pointer to get the conditional expression so that it can test it. When the trigger condition becomes true, the following occurs:

1. The trigger's activate() function is called. The activate() function uses its *DtWhenStmt* pointer to look up the matching *DtWhenBlock* pointer.
2. If the trigger is a “Retask” trigger (the only kind in the VR-Forces front-end), the trigger tells the *DtPlan* to pushExecutionStack().

In a Retask trigger, a trigger behaves like an independent task. When the trigger fires, the entity stops executing the current task and begins to execute the trigger block. When the trigger is complete, the entity executes the first plan statement after the task it was executing at the time the trigger fired.

As an alternative to the retask model, an entity could return to the task it had been executing, could consider the plan complete when it is finished with the trigger block, or could take some other action.

3. The trigger tells the *DtPlan* to execute the *DtWhenBlock* by calling DtPlan::executeTrigger().
4. The trigger sets its **isDone** flag, which indicates to the *DtPlan* that it should be deleted.



12

The VR-Forces Remote Control API

The VR-Forces Remote Control API allows an application, such as a GUI front-end, to send control messages to a VR-Forces application.

12.1. Introduction

The VR-Forces Remote Control API is a set of classes that allows an application to easily control one or more remote VR-Forces applications (that is, an application that is using the VR-Forces simulation engine to simulate entities and other objects.) The VR-Forces GUI (the *vrfGui* executable) uses the VR-Forces Remote Control API to control the *vrfSim* application. The Remote Control API is also meant to be used in custom VR-Forces front-ends, simulation managers, or Instructor/Operator Stations.

When you use the Remote Control API, you do not need to worry about the details of the network messages being exchanged between your application and the VR-Forces applications. These are handled transparently, in the implementation of the Remote Control API. On the other hand, the API provides access to these messages, so that you can extend or modify the way the Remote Control API communicates with VR-Forces simulation engines if you need to.

You can use the Remote Control API to drive a VR-Forces simulation engine that is running as a part of the same application (that is, building a GUI and simulation engine into a single executable). However, when you do so, you are controlling the simulation engine as if it were remote. You cannot take advantage of detailed knowledge of the internals of the simulation engine, as you could by using the VR-Forces simulation engine's API directly. But in exchange, you gain the flexibility to easily separate your GUI into a separate networked executable later.

VR-Forces also has a mechanism for controlling remote front-ends through MTL commands.

12.2. Using the Remote Control API

The Remote Control API consists of several libraries. The *vrfControl* library contains most of the classes that provide the top-level API for controlling VR-Forces applications. This library also uses code from several other libraries that the Remote Control API shares with the VR-Forces API. For example, both the Remote Control API and the VR-Forces API use the same classes to represent the messages that are exchanged between them.

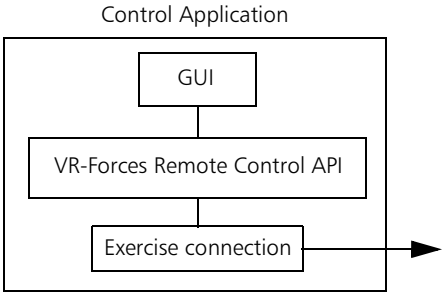


Figure 12-1. Control application using VR-Forces remote control API

The central class in the Remote Control API is *DtVrfRemoteController*, defined in *vrfController.h*. An application that wants to control remote VR-Forces applications needs to instantiate *DtVrfRemoteController* and use its member functions to send instructions to the simulation engine, for example, loadScenario(), pause(), run(), createEntity(), modifyRoute(), deleteObject(), setEntitySpeed(), or assignTask(). Please see the comments in *vrfController.h* for the complete list.

For example, the following code implements an application that tells remote VR-Forces applications to load the scenario called “foo.scn”.

```
DtExerciseConn conn(3000, 1, 1, 1);
DtVrfRemoteController controller(&conn);
controller.loadScenario("foo.scn");
```

A *DtVrfRemoteController* uses a VR-Link exercise connection (*DtExerciseConn*) to send its control messages to VR-Forces applications (Figure 12-1). Therefore an exercise connection must be passed to its constructor. An alternate constructor allows you to pass a *DtVrfMessageInterface* – a wrapper around *DtExerciseConn* that is able to decode some VR-Forces protocol extensions. (When you use the main constructor, the *DtVrfRemoteController* instantiates its own *DtVrfMessageInterface*.)

Like all DIS and HLA applications that are based on VR-Link, applications that use the VR-Forces Remote Control API must call drainInput() periodically on the *DtExerciseConn*. In HLA, this allows VR-Link to give the RTI some processing time by calling RTItick(). In both DIS and HLA, it ensures that the *DtVrfRemoteController* can get necessary feedback from remote VR-Forces applications, acknowledge messages, find out if VR-Forces applications have joined or left the exercise, and so on.

12.3. Choosing Which VR-Forces Applications to Control

Most of *DtVrfRemoteController*'s member functions take an optional simulation address argument. (In a custom VR-Forces application, you set the simulation address by passing it to the *DtCgf* or *DtVrfApp* constructor.) For most of *DtVrfRemoteController*'s functions, the simulation address argument defaults to *DtSimSendToAll*, meaning that the command should apply to all VR-Forces applications that are currently part of your exercise.

If there is only a single VR-Forces application running, it does not matter if you send commands to the specific address of that simulation engine, or if you just use the default *DtSimSendToAll*. But if there are multiple remote simulation engines, you must be sure that commands are addressed appropriately. For example, when you want to create a new scenario, use *DtSimSendToAll*, so that all simulation engines can load the terrain, close any previously loaded scenario, and prepare to accept object creation commands. But when you create an object, you must indicate which simulation engine should instantiate and simulate the object. (If you mistakenly pass *DtSimSendToAll* to any of the functions that request that an object be created, the command will be sent only to the first address on the Remote Controller's list of known simulation engines. This prevents multiple applications from trying to instantiate the same object.)

It is safe to pass *DtSimSendToAll* to the functions that implement modify and delete commands, since the simulation engine that is simulating the designated object will know that the command is designated for that application. Figure 12-2 illustrates sending messages to specific back-ends and to all back-ends.

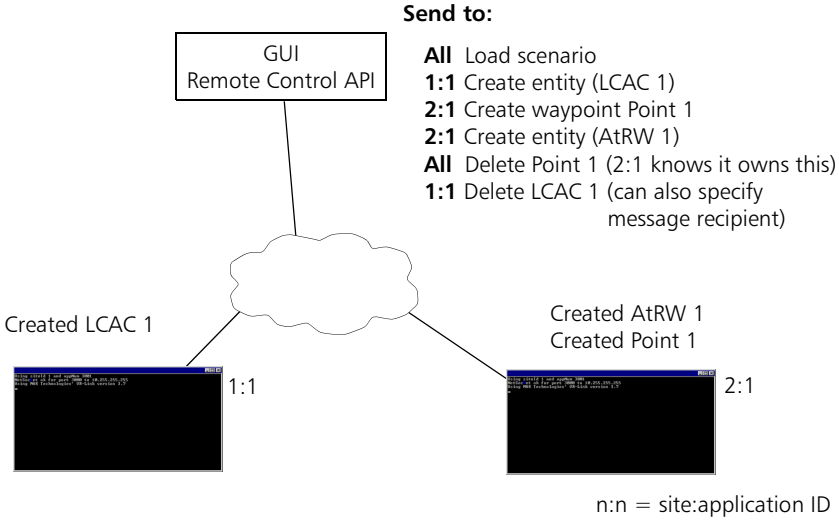


Figure 12-2. GUI sending control messages to back-ends

12.4. Finding Out About Remote VR-Forces Applications

DtVrfRemoteController uses a *DtVrfBackendListener* (*vrfBeListener.h*) to keep track of all currently available remote VR-Forces applications. VR-Forces applications send status heartbeats to alert remote control applications like *vrfGui* of their presence. The *DtVrfBackendListener* receives these messages, and maintains a list of *DtSimulationAddress* pointers identifying these applications. You can access this list using DtVrfRemoteController::backendList(). For example, to send a newScenario() command to just the first VR-Forces application on the list, do the following:

```
DtVrfRemoteController controller(&conn);
DtListItem* item = controller.backendList().first();
if (item)
{
   DtSimulationAddress* addr = (DtSimulationAddress*) item->data();
   controller.newScenario(*addr);
}
```

VR-Forces remote control applications, such as *vrfGui*, need to be notified of the status of VR-Forces applications, so that they can update their GUIs, or otherwise notify the user which back-ends are available, for example, when issuing a createEntity() command. To do this, register back-end discovery callbacks or back-end removal callbacks with the *DtVrfBackendListener* (through the *DtVrfRemoteController*). Use *DtVrfRemoteController*'s add/removeBackendDiscoveryCallback() and add/removeBackendRemovalCallbacks() to register and unregister your callbacks.

As an alternative to using *DtVrfBackendListener*, you can find out information about back-ends through *DtBackEnd* (*backend.h*). You can ask for information about a specific back-end, which may be more convenient than using *DtVrfBackendListener* and *DtVrfRemoteController*. The *DtVrfRemoteController* maintains a *DtBackEnd* for each back-end in the session. The *DtBackEnd* get updated via the status messages that each back-end sends.

To look up information about a back-end, use the DtVrfBackendListener::lookupBackend() member function.

12.5. Loading a Scenario

When you call loadScenario(), your remote controller application does the following:

1. If the object map file associated with the scenario indicates that the scenario uses multiple simulation engines, it takes the order of battle and plan files specified by the scenario file, and breaks them up into smaller files, each containing the object or plan information necessary for a particular simulation engine.
2. It sends those files to the appropriate simulation engines, through dedicated TCP sockets.
3. The loadScenario() function sends a message to all back-ends, indicating that they should load the scenario that they have just received. So when loadScenario() returns, all necessary data has been sent to the remote VR-Forces applications, but they may still be in the process of loading those files and initializing. Therefore, you usually do not want to immediately start issuing other commands through the remote controller.

Figure 12-3 illustrates the load scenario process for multiple back-ends and shows that back-ends may load scenario files at different rates.

To find out when the various back-ends have completed loading the scenario and initializing, you can register for callbacks through the *DtVrfRemoteController*. You can use add/removeBackendLoadedCallback() to register or unregister callbacks that you would like to have executed each time a remote VR-Forces application has indicated that it has finished loading the scenario. Or, you can use add/removeScenarioLoaded-Callback() for callbacks that you want to have called only when the scenario has been completely loaded by all participating back-ends.

These callbacks are used by the front-end that loads a scenario. Other front-ends can find out if a back-end has loaded a scenario with the DtBackEnd::isLoaded() member function.

The loadScenario() member function takes two optional arguments that indicate what you want to happen if all back-ends called for by the Object Map File are not available. The *DtVrfController* knows (through its *DtVrfBackendListener*) the simulation addresses of all VR-Forces applications that are currently running. By default, partial loads are allowed, meaning that loadScenario() will ignore missing back-ends, and just ask those applications that are available to load their portions of the scenario. If a value of `false` is passed as the **allowPartial** argument, then loadScenario() will fail (returning `false` without asking any back-ends to load anything) unless all expected simulation engines are present. If partial loads are allowed, and if a non-NULL *DtList* pointer is passed as the **missingBackends** argument, the *DtList* will be filled with pointers to the *DtSimulationAddresses* of the missing applications.

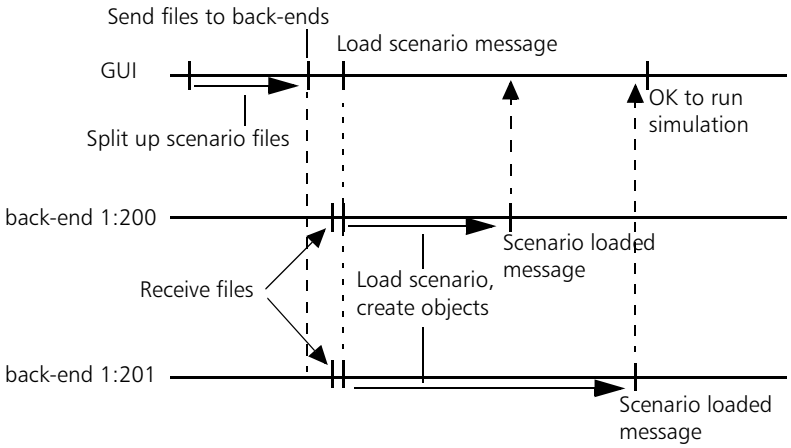


Figure 12-3. Load scenario timeline: two back-ends

12.6. Saving a Scenario

When you use saveScenario() to ask remote VR-Forces applications to save the current scenario, the reverse of the load process occurs. The *DtVrfController* sends a message to all simulation engines asking each to save order of battle and plan files containing information for just the objects that it simulates. Once those saves are complete, those simulation engines send their files to the remote controller that initiated the save. The *DtVrfController* then combines the individual files into a single scenario-wide order of battle file and plan file, and saves the scenario locally using the indicated file name. So when saveScenario() returns, the scenario will not have actually been saved yet. Make sure to continue calling drainInput() on your *DtExerciseConn* after saveScenario() returns, so that the *DtVrfController* can receive saved information from the simulation engines (through callbacks on *DtExerciseConn*), and complete the save. Figure 12-4 illustrates the save process.

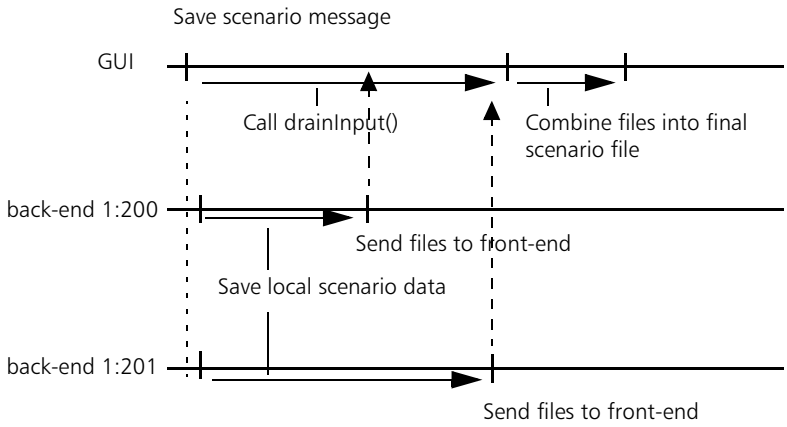


Figure 12-4. Save scenario timeline: two back-ends

After the remote control application sends out a save command, it waits a specific amount of time for all the back-ends to reply. If all back-ends reply before the time is up, the files are saved. If some back-ends do not reply within the time interval, the Remote Controller prints a warning and saves whatever information it has. You can configure the time-out period with the DtVrfRemoteController::setMaxWaitTime() member function.

You can keep track of back-ends that failed to save by registering a backendSave() callback. This callback gets invoked once for each back-end involved in the save and indicates whether the save was successful or not.

To find out when you have received save information from each back-end, you can register for callbacks through the *DtVrfRemoteController*. You can use add/removeBackendSavedCallback() to register/unregister callbacks that you would like to have executed when we have finished receiving save information for a particular back-end. Or, you can use add/removeScenarioSavedCallback() for callbacks that you want to have called only when the scenario has been completely saved.

12.7. Managing VR-Forces Objects

Using the Remote Control API, applications can manage VR-Forces objects. Users can command back-ends to create new objects, modify them, and delete them. The Back-end Remote Control Example (*./doc/classdoc/simclassref/remotecontrol.html*) shows how to use the *DtRemoteController* to perform these actions.

12.7.1. Creating Objects

As mentioned in "Choosing Which VR-Forces Applications to Control," on page 12-4, the Remote Control API allows you to send messages to all back-ends or to a specific back-end. When you create an entity, you must ensure that a single back-end address is specified. Otherwise, multiple VR-Forces applications will try to instantiate the same object. If a value of *DtSimSendToAll* (the default) is passed to one of *DtVrfRemoteController*'s create methods, *DtVrfRemoteController* issues a warning and sends the command to the first address in its back-end list.

When you create an entity using the Remote Control API, *DtVrfRemoteController* sends a command message to the appropriate back-end telling it to create a new object. The back-end responds by creating the object and replying with an "object created" message. Figure 12-5 illustrates this process.

When you call a create*() method, you can register a callback that is invoked when the "object created" message arrives from the back-end. The callback function contains the object ID and object name of the new object. The callback is unregistered automatically by *DtVrfRemoteController*.

When you create an object, you can specify the name that the VR-Forces back-end should assign to the object. However, since names must be unique, if the name you request is already in use, VR-Forces assigns a name using the default naming convention.

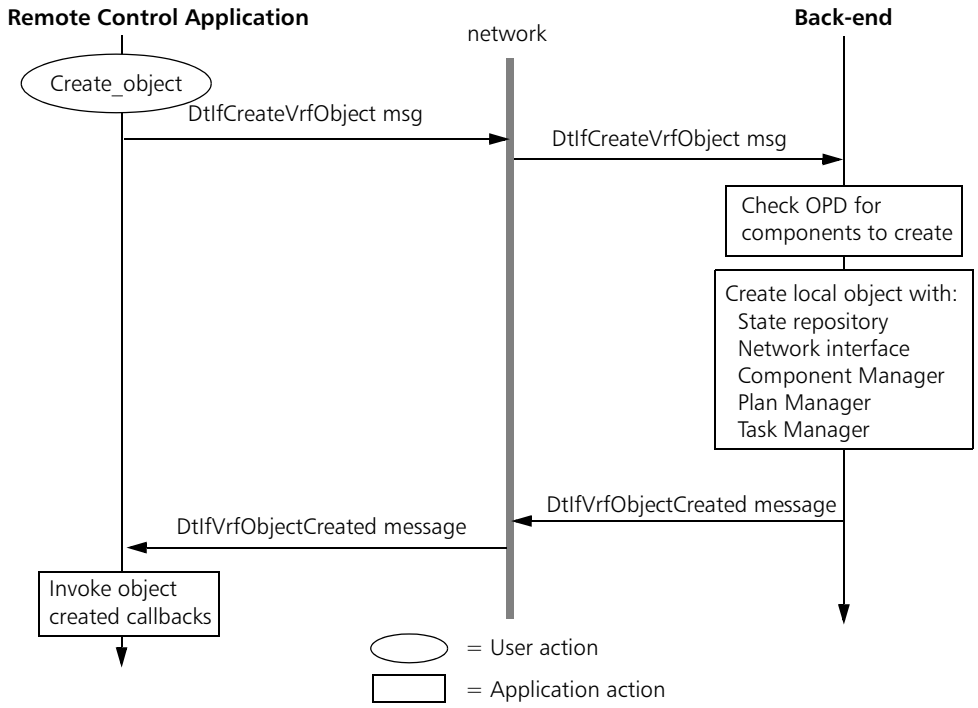


Figure 12-5. Creating an entity

12.7.2. Modifying and Deleting Objects

You can change the name, label, or vertices that define a control object. The modify methods each take an object identifier to indicate which object should change.

For entities and aggregates, values can be modified through *DtVrfRemoteController*'s set*() methods. These methods allow users to set and modify entity and aggregate attributes such as location, altitude, and aggregate formation. For the full list, please see *vrfController.h*.

Any VR-Forces object can be deleted by calling deleteObject() with either the object name or object ID.

12.7.3. Changing the Entity Hierarchy

The Remote Control API has function calls that reorganize entities. These calls send messages to all of the simulation engines, and therefore all Organization Managers involved in the exercise.

The createAggregate() member function creates a new aggregate entity of the type and force specified. Then it calls setSuperior() for each of the subordinates to assign them as subordinates of the new aggregate.

The addToOrganization() member function causes a setSuperior() call to be made to the Organization Manager. The specified subordinate is detached from its current superior, and added to the new superior.

The addToForceLevelOrganization() member function causes a setSuperiorToForceLevel() call to be made to the Organization Manager. The specified subordinate is detached from its current superior and added to the force level aggregate matching the entity's force type.

The removeFromOrganization() member function causes a detachFromSuperior() call to be made to the Organization Manager. The specified subordinate will not have any superior.

12.8. Tasks and Plans

To assign a task using the Remote Control API, make the appropriate *DtVrfRemoteController* member function call. Please see the *DtVrfRemoteController* header file (*vrfController.h* for more information).

The Remote Control API allows you to assign, request, and abandon plans:

- You can register several types of callbacks to monitor changes and to discover the current status of assigned plans.
- You can subscribe and unsubscribe to an echelon ID's plan messages using subscribePlan() and unsubscribePlan(). The function passed to subscribePlan() is invoked whenever the current plan is changed or whenever a request has been made for the plan.
- You can register and unregister callbacks to notify you when individual statements are executing (using add/removePlanStatementCallback()).
- You can register and unregister callbacks to notify you when a plan is complete (using add/removePlanCompleteCallback()).

When assigning plans to entities, the *DtVrfRemoteController* uses VR-Forces plan data structures to represent the plan.

12.9. Running VR-Forces Applications in Batch Mode

The *DtRemoteBatchManager* (*remBatchMgr.h*) class controls the execution of a batch of scenarios on one or more remote VR-Forces back-ends. It takes a pointer to a *DtVrfRemoteController*, which it uses to interact with the back-ends. The VR-Forces back-ends used to execute the scenarios in the batch do not need a *DtBatchManager* to run with the Remote Batch Manager.

The *DtRemoteBatchManager* has exactly the same functions available as the *DtBatchManager* (loadBatch(), runBatch(), stopBatch(), and so on. For details, please see "Running VR-Forces Applications in Batch Mode," on page 15-19.

The *DtRemoteBatchManager* also contains a *DtScenarioBatchIterator* and a *DtRemoteLoggerController* just as the *DtBatchManager* does. These operate in the same way.

12.10. Building an Application Using the Remote Control API

This section describes the various settings for building an application with the Remote Control API.

On Windows, set the paths to VR-Forces and VR-Link's *include* and *lib* directories in your project settings. Also, you need the paths to your RTI tree for HLA. VR-Link and RTI distributes both debug and release versions of its libraries.

HLA only

For HLA, link against the following libraries:

- From VR-Forces:

```
vrfcontrolHLA13RT.lib vrfobjparamRT.lib vrfcoreHLA13RT.lib
  vrfmsgsRT.lib vrfplanRT.lib vrftasksRT.lib vrfutilRT.lib
  vrfgdbRT.lib vrfExtProtocolRT.lib
  vrfMsgTransportHLA13RT.lib
```

For each library with HLA13 in its name, you can substitute HLA1516 to build for the RTI 1516 specification. You can also specify the HLA13 and HLA1516 libraries together. This note applies to the VR-Link libraries.

- From VR-Link:

```
vlHLA13RT.lib matrixRT.lib mtlRT.lib vlutilRT.lib
```

- From the RTI:

```
libRTI-NG.lib libfedtime.lib
```

DIS only

For DIS, link against the following libraries:

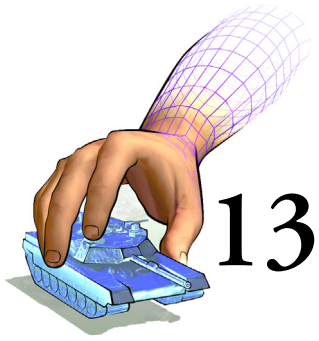
- From VR-Forces:

```
vrfcontrolDISRT.lib vrfobjparamRT.lib vrfcoreDISRT.lib
   vrfmsgsDISRT.lib vrfplanRT.lib gdbRT.lib
   vrfvrlinkExtDISRT.lib
vrftasksRT.lib vrfutilRT.lib
```

- From VR-Link:

```
vlDISRTd.lib matrixRTd.lib mtlRTd.lib vlutilRTd.lib
```

An example project file is in *./examples/remoteControl.*



13

The Terrain API

This chapter describes how to use the Terrain API, including how to create terrain databases and work with vector data.

13.1. Introduction

The Terrain API is a set of classes that enable applications to read, write, and query terrain databases. Through the Terrain API you can:

- Load terrain databases in MÄK GDB (native) format as well as a variety of external formats (such as OpenFlight, ESRI Shapefile, and CTDB)
- Save terrain databases to MÄK GDB format
- Access and manipulate the terrain geometry (polygon) and vector data
- Perform intersection tests with the terrain geometry and vector network
- Query the vector network for information about feature data.

The VR-Forces GUI uses the terrain API to access the geometry and vector data it needs to display the terrain data. The VR-Forces simulation engine uses the terrain API to perform terrain intersection tests for modeling ground vehicle movement, and line of sight in its sensor model.

13.2. Using the DtTerrainDatabase Class

The *DtTerrainDatabase* class (*terrainDb.h*) contains the representation of terrain skin and terrain features. The terrain skin consists of the geometry of the terrain (typically represented as polygons) and its surface characteristics (that is, soil type). Terrain features include natural and man-made items such as trees, rivers, roads, buildings, and so on. Terrain features are represented as a vector network, comprised of points, lines, and areas.

The *DtTerrainDatabase* class organizes the terrain surface and feature data in tree structures comprised of various types of nodes and elements. At the top of the trees are two key nodes and a key data structure: a terrain group node, a features group node, and a vector network. All of the items in the terrain and feature node trees are of type *DtGdbNodes*. Among the types of *DtGdbNodes* used are:

- *DtGroup* (*group.h*) – Contains any number of child nodes
- *DtTriangleIndirect* (*triInd.h*) – Triangle representation that holds references to vertices and surfaces to allow for efficient sharing of duplicated vertices and surfaces with other *DtPolygonIndirects* and *DtTriangleIndirects*
- *DtBaseCoordinateSystem* (*baseCoord.h*) – Provides a transformation (translation and rotation) for all nodes beneath it in the tree.

Additionally, the *DtTerrainDatabase* contains the following member data structures:

- *DtSurfaceManager* (*surfMgr.h*) – Manages the surface descriptions (soil types) referenced by *DtPolygonIndirect* and *DtTriangleIndirect* objects
- *DtVertexManager* (*vtxMgr.h*) – Manages the vertices referenced by *DtPolygonIndirect* and *DtTriangleIndirect* objects
- *Coordinate_System* (*coordSystem.h*) – Stores the coordinate system information for the entire terrain database.

The nodes specific to the vector network are:

- *DtVectorNetwork* (*vecNetwrk.h*) – Contains a set of vectors, points, and edges that represent feature data.
- *DtSpecificationManager* (*specMgr.h*) – Manages the specifications (collections of attributes describing a feature) associated with items in the vector network. Specifications are referenced by *DtNetworkNodes*, *DtNetworkSegments*, and *DtNetworkEdges* in the vector network.
- *DtNetworkNode* (*netwrkNode.h*) – Point features (such as a tree or building), or junctions between edges.
- *DtNetworkEdge* (*netwrkEdge.h*) – Sequence of segments, with a start and end node. Edges connect to one another to form the network.
- *DtNetworkSegment* (*netwrkSeg.h*) – Vectors (line segments) used in the representation of linear or area features.

DtNetworkSegment is an implementation detail and its use is strongly discouraged. Instead, use the point interface exposed by the *DtNetworkEdge* class.

13.2.1. Accessing the *DtTerrainDatabase* in VR-Forces

You can load a terrain database from a file into the VR-Forces back-end by calling the *DtCgf* class's newScenario() or loadScenario() member functions. In the case of newScenario(), you explicitly specify the name of a terrain database file (a *.gdb* native format file, or supported format such as OpenFlight) to load. In the case of loadScenario(), you specify a scenario file (*.scn* file), which contains the name of a terrain database file to load. Once the database is loaded, you can access it by calling DtCgf::terrainInterface().

```
// Create a new scenario
cgf.newScenario("../data/terrain/ground-db.gdb");
DtInfo("Loaded terrain database file %s\n",
  cgf.terrainInterface()->fileName().string());
```

VR-Forces uses *DtTerrainReader* classes that know how to read terrain data in a given file format. It uses a factory mechanism to create the appropriate kind of reader to load a terrain file (keying off of the file suffix to determine the type of input data). For more details about terrain readers, please see "Terrain Readers," on page 13-26.

Once the terrain database is loaded, you can access the terrain geometry data (the top-level node in the terrain group) and the vector network as follows:

```
DtGroup* terrain = cgf.terrainInterface()->terrain();
DtGroup* features = cgf.terrainInterface()->features();
DtVectorNetwork* vectorNetwork = cgf.terrainInterface()->terrain()
  ->vectorNetwork();
```

13.2.2. Terrain Intersection Tests

The *DtTerrainDatabase* class has an interface that lets you query for information about the terrain data. One of the most common queries made to the terrain database is for a terrain intersection. Terrain intersections are used in the ground entity models to determine how to position and orient entities on the terrain surface. The Physical World performs terrain intersection queries, if configured to do so, when performing such operations as line-of-sight determination and determining the terrain height at a specified location. Objects should not access the terrain database directly; they should go through the physical world.

DtNetworkSegment is an implementation detail and its use is strongly discouraged. Instead, use the point interface exposed by the *DtNetworkEdge* class.

The following example shows how to query the terrain database for an intersection with the terrain surface, returning the location of the intersection only:

```
// Setup chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

DtPoint isectPoint;
double intersectionTime;

// Calculate intersection with the terrain skin
if(terrainDb->intersect(chord, isectPoint, intersectionTime) == true)
{
  DtInfo("Intersected terrain at location %s\n", isectPoint.string());
}
```

You can make more complex queries to the terrain database, requesting such information as the list of all intersections a given chord makes with the terrain, surface type information at the point of intersection, and intersections with features in the vector network. For more details about terrain intersections, please see "Querying the Terrain Database," on page 13-9.

13.2.3. Coordinate Systems

A *DtTerrainDatabase* has a coordinate system, which defines the origin and type of coordinates being used in the database. The *Coordinate_System* class (*coordSystem.h*) represents the coordinate system for the database. The *Coordinate_System* class provides member functions for transforming a coordinate between the local coordinate system (the coordinate system used in the database) and the geocentric coordinate system. The derived types of *Coordinate_System* classes include:

- *DtGeodeticCoordinateSystem* – Geodetic coordinates (latitude, longitude, altitude)
- *UTM_Coordinate_System* – Universal Transverse Mercator
- *lambertConformalProjection* – Lambert Conformal Conical.

The *DtTerrainDatabase* class has member functions for setting and retrieving the coordinate system used in the database. It also has functions for transforming from its current coordinate system to any other coordinate system, performing the necessary transformations on loaded terrain data as appropriate.

13.2.4. Terrain Geometry

The terrain skin is represented as a n-tree structure of *DtGdbNodes* (*gdbNode.h*), organized by location. *DtGdbNode* is the base class for all nodes in the terrain database. *DtGdbNodes* define the basic interface for intersection functions, and have the concept of a parent and child nodes.

The *DtTerrainDatabase* class provides functions for allocating nodes of all types, in a memory-efficient manner.

The main node types used in the terrain skin representation are:

- Triangles and polygons
- Group nodes
- Coordinate transformation nodes.

Triangles

The *DtTriangleIndirect* (*triInd.h*) node represents polygons, and is the leaf node in the terrain geometry tree that represents the actual terrain skin. It uses memory and disk space efficiently and avoids common problems such as cracks between adjacent polygons.

Each triangle maintains references to a vertex manager and a surface manager, which manages its surface and vertices. Each triangle has three vertexIDs and a DtSurfaceID, identifiers associated with specific *DtVertices* (typedef'd to *DtPoint* (*point.h*)) and *DtSurfaces* (*surface.h*) of the triangle. *DtVertices* are stored and managed by the Vertex Manager. *DtSurfaces* are stored and managed by the Surface Manager. The *DtSurface* is an object that holds characteristics of the triangle surface, such as its color and soil type. Surfaces are discussed in more detail in "Terrain Surface (Soil Type)," on page 12-7.

Group Nodes

DtGroups are the collections of child nodes in the terrain database tree. The *DtGroup* class provides an interface for adding and removing child *DtGdbNodes*. A group automatically calculates its extent based on its child nodes (used in drawing).

Coordinate Transformation Nodes

The *DtStaticCoordinateSystem* node has a transformation (translation and rotation) matrix and a single child (typically the top node of a tree). It applies the transformation matrix to the child node.

13.2.5. Terrain Surface (Soil Type)

Each polygon or triangle has a *DtSurface* associated with it. A *DtSurface* holds the characteristics of a surface, including its color and the nature of the surface. A *DtSurface* has the following attributes:

- DtColor – RGB color
- DtMedium – for example, ground, water
- DtEnvState – environmental state, for example, dry, icy, wet
- DtSoilType – for example, paved road, swamp, muddy, and so on
- DtRoughnessSoilType – derived from soil type and used in modeling ground vehicle behavior (moving factor efficiency).

Color is an RGB value associated with the surface. This color is used when drawing the terrain database in a graphical display, such as in the VR-Forces front-end.

The medium, environmental state, and soil type attributes are represented by a set of enumerated types defined in the *DtSurface* class. Please see *surface.h* for the complete set of enumerations.

The DtRoughnessSoilType is a classification of terrain surface that is suitable for use in simulators that need to model the effects of soil type on ground vehicle behavior. It is derived from the soil type parameter. (Soil types are automatically mapped to roughness soil types in the *DtSurface* class.)

VR-Forces uses DtRoughnessSoilType in its ground vehicle model. DtRoughnessSoilType gets mapped to a set of soil-list parameters (acceleration, stopping factor, and max-speed) in an entry in the object parameter database.

DtSurfaces are stored in a *DtSurfaceManager*, a member of the *DtTerrainDatabase*. Individual triangles and polygons just hold a reference to a *DtSurface* in the *DtSurfaceManager*.

13.3. Querying the Terrain Database

DtTerrainDatabase provides a set of intersection functions you can use to query the terrain database, using 3-D chords to define the segment and direction of interest for intersections.

Through the Terrain API, you can ask the following:

- Is there an intersection?
- What is the first point at which an object intersects the terrain?
- What are all of the intersections along the chord?
- What is the closest intersection, along the down/up vectors to the specified location?
- What is the height of the terrain at a specified location?

Additionally, for a given intersection query you can specify what kinds of information you want to receive:

- Boolean: indicating intersection or no intersection (no additional data)
- Point: coordinates of the intersection point
- Normal: the normal vector in the intersection point
- Triangle: a pointer to the triangle or polygon intersected and its characteristics (surface, vertices, and so on)
- Vector network intersections: first (best) intersection, or list of intersections along the chord.

GDB supports only triangles, with vertices stored in counterclockwise order for intersection calculation, but it does not enforce these properties. It is up to you to create valid polygons that are correctly oriented.

For example, to query the terrain database for the intersection with the terrain surface, returning the location of the intersection only, do the following:

```
// Set up chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

DtPoint isectPoint;
double intersectionTime;

// Calculate intersection with the terrain skin
if(terrainDb->intersect(chord, isectPoint, intersectionTime) == true)
{
  DtInfo("Intersected terrain at location %s\n", isectPoint.string());
}
```

You can use the **intersectionTime** parameter returned in the intersect() function to determine where along the chord the intersection with the terrain occurred. Values for intersection time *t*, given a chord(*p1*, *p2*) indicate the following:

- $t < 0$ – intersection lies on an extension of the chord before *p1*
- $0 <= t <= 1$ – intersection lies on the chord
- $t > 1$ – intersection lies on an extension of chord after *p2*.

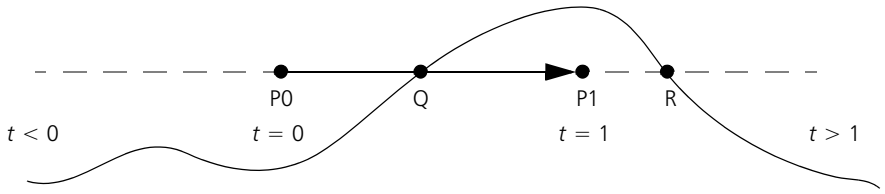


Figure 13-1. Intersection point

In the Figure 13-1, a call to intersect() with chord (P0, P1) intersects the terrain at point Q. The call will return TRUE, because point Q lies along the chord, and the intersection time at Q will be a value between 0 and 1. The intersection time at point R (which lies on an extension of the chord) has an intersection time > 1.

To get more information about the intersection point, you can call a variant of intersect() that returns a *DtChordIntersectionRecord* (*chrdIntRec.h*), instead of just the *DtPoint*. The *DtChordIntersectionRecord* can return information such as the location of the intersection, surface type, normal vector of the terrain at that point, and the polygon intersected. For example, to make an intersection test (similar to the previous example), requesting the intersection point and normal, do the following:

```
// Set up chord for intersection test. Query starts at p1 and ends at p2.
DtPoint p1(2200, 2200, 10000);
DtPoint p2(2200, 2200, -10000);
DtChord chord(p1, p2);

// Return intersection details in intersectRecord
DtChordIntersectionRecord intersectRecord;

// Calculate intersection with terrain skin, requesting that the normal
// and the location of the intersection point be retrieved.
DtGdbNodeDtIntersectRecordType flag =
  (DtGdbNodeDtIntersectRecordType)(DtGdbNodeIRT_IPOINT |
  DtGdbNodeIRT_INORMAL);

// Try to intersect the terrain to get the left triangle
if(!terrainDb.intersect(chord, intersectionRecord, flag))
{
  DtInfo("Normal at terrain intersection %s is %s\n",
    intersectRecord.intesectionPoint().string(),
    intersectRecord.normal().string());
}
```

13.4. Using Vector Networks

A vector network is a structure of vectors and points used to represent features in the database such as rivers, roads, and buildings. A *DtTerrainDatabase* represents the vector network in the following nodes:

- *DtVectorNetwork* (*vecNetwrk.h*): holds the actual set of vectors, points, and edges
- *DtSpecificationManager* (*specMgr.h*): a member of the *DtVectorNetwork* that manages the specifications (collections of attributes describing a feature) associated with items in the vector network.

Features are represented in the vector network using the following nodes:

- *DtNetworkNode* (*netwrkNode.h*): Point features (such as a tree or building), or junctions between edges.
- *DtNetworkEdge* (*netwrkEdge.h*): Sequence of points, with a start and end node.
- *DtNetworkSegment* (*netwrkSeg.h*): Vectors used in the representation of linear or area features.

DtNetworkSegment is an implementation detail. It is exposed to provide you with the underlying representation of the edge, where necessary, but it is recommended that you use the point interface exposed by the *DtNetworkEdge*.

Network nodes and edges have a *DtSpecification* (*specification.h*) associated with them. The specification holds information about the feature, such as the number of floors it has (in the case of a building), the height and width of the feature, and identifiers, such as DFAD or SEDRIS codes. For more information about specifications, please see "Feature Specifications," on page 13-13.

13.4.1. Linear Features

Linear features, such as roads and rivers are made up of one or more *DtNetworkEdge* objects. A *DtNetworkEdge* is made up of a start *DtNetworkNode*, an end *DtNetwork-Node*, and a series of points that define the smallest straight segments of the edge. (This is why, internally, the edge is represented as both a series of points and as a series of *DtNetworkSegments*.) A *DtNetworkSegment* represents a portion of a network that can be approximated by a line segment (vector).

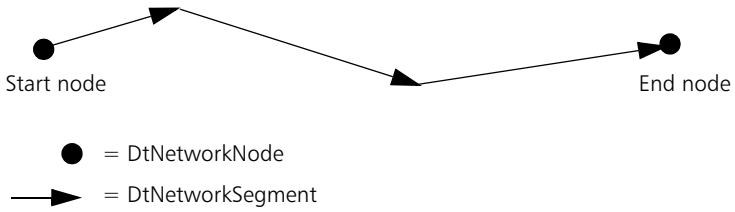


Figure 13-2. Linear feature (edge)

Edges have a direction. The sequence of segments that comprise an edge never intersect one another. A *DtNetworkSegment* never exists by itself in the vector network – it is always part of a *DtNetworkEdge*. Edges can be connected to one another (or themselves) at their endpoints.

It is recommended that you use the point interface, and not access segments directly, as they are considered to be an implementation detail and could change in the future.

Iterating Through the Points in an Edge

To iterate through the points of an edge, ask the edge for a *DtNetworkEdgeTraverser* object. You must specify whether or not to respect the directionality of the edge. If you ignore the directionality, both a forward and backward traverser may be created. However, if you respect the directionality, then for a unidirectional street, only a forward edge traverser can be created.

Once a traverser is created for an edge, ask for a *DtNetworkEdgePointIter* over that edge. The point iterator allows you to iterate over the points contained in the edge. The *DtNetworkEdgePointIter* abstracts away the direction of traversal. Thus, when moving both forwards and backwards along an edge, incrementing the *DtNetworkEdgePointIter* returns an iterator pointing to the next point as appropriate.

13.4.2. Areal Features

Areal features are represented by a *DtNetworkEdge* that forms a closed figure (initial and final point are the same), plus an additional point located approximately in the center of the feature.

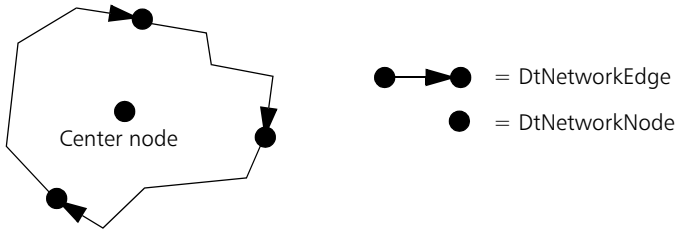


Figure 13-3. Areal feature

Each area has its own specification entry, which must include the extents of a bounding box.

13.4.3. Feature Specifications

A *DtSpecification* represents the set of characteristics associated with a feature in the vector network. A specification can include any number of attributes used to describe the feature. For example, the specification for a building might include the number of floors, while the specification for a river might include width and water level. The *DtSpecificationAttribute* class (*specAttrib.h*) represents the individual attributes held in a *DtSpecification*. A *DtSpecificationAttribute* has a string label and a value. Derived kinds of *DtSpecificationAttribute* are defined for integer, real, and string values.

You can look up an attribute in a specification by its string label. For example, to get the ECC code (a SEDRIS string identifier) for a specification attribute, do the following:

```
const DtSpecificationStringAttribute* group =
  (const DtSpecificationStringAttribute*)spec.attribute("ECC");

if(group)
{
  DtInfo("ECC = %s\n", group.string());
}
```

The ECC code is one of a number of specification attributes typically defined in terrain databases. Please see Appendix A, *The MÄK Specification Model* in *MÄK Terrain Database Tool Users Guide* for a list of attribute label/value pairs used in MÄK terrain databases.

A *DtSpecification* has some predefined attributes. They are fundamental attributes that can be associated with any feature. The API has functions for setting and retrieving these attributes directly. Table 13-1 lists basic mutator functions.

Table 13-1: Attribute accessor and mutator functions

Accessor/Mutator	Description
setLength() length()	Length of the feature
setWidth() width()	Width of the feature
setColor() color()	RGB color
setVisibility() visibility()	Boolean is visible on/off

The length and width attributes can also be accessed by the GDB_Length and GDB_Width attribute labels.

The *DtSpecification* class allows any number of additional *DtSpecificationAttributes* to be defined and added to it. For example, to create and configure a specification for a building, do the following:

```
DtSpecification spec;

// set values for the predefined attributes

spec.color(DtColor(100,100, 100, 0));
spec.setLength(100);
spec.setWidth(10);

// Add a “Type” attribute to the specification, with a value of “Building”

DtSpecificationStringAttribute sattr;
sattr.setLabel("Type");
sattr.setValue("Building");

spec.addAttribute(sattr);
```

DtSpecification stores the attributes in a hash list indexed by label. To modify a value, add the attribute again with a new value and it will replace the old one.

If you create a *DtSpecification* for an areal vector feature, that is, the specification has a FeatureType = 2 attribute, then the specification must also contain the following attributes:

- DtXMin
- DtXMax
- DtYMin
- DtYMax
- DtZMin
- DtZMax.

Each of these attributes is a double. They represent the extent of the areal feature. If these attributes are not present, VR-Forces uses 0.0, which may result in unintended consequences.

13.4.4. The MÄK Specification Model

MÄK has developed its own model based on the Digital Feature Analysis Data (DFAD) specification and SEDRIS enumerations. While you can develop your own specification model through the API, the MÄK specification model is already in place (all of our terrain readers use this model). You can use it as-is, extend it, or develop your own scheme for classifying and describing feature data. Please see Appendix A, *The MÄK Specification Model* in *MÄK Terrain Database Tool Users Guide* for a description of the specification attributes used in the MÄK specification model.

13.4.5. Querying the Vector Network

You can query the vector network to ask for intersections with features. To do this you need to provide a *DtVectorNetworkRecord* (*vecNetwrkRecord.h*) parameter, in which you can specify the criteria for selecting terrain features along the intersection chord. Intersected vector network features that meet the criteria in the record parameter are returned in a *DtVectorNetworkIntersectionRecord* (*vecNwrkIntRec.h*). For example, to query for the terrain skin intersection and for all of the vector network features along the chord, do the following:

```
// Configure vector network record parameter to only select roads
DtVectorNetworkRecord dataRec;
DtSpecificationStringAttribute specAttribute;
specAttribute.setLabel("DtFeatureGroup");// MAK-defined DtGroup attribute
                                  // used to classify features
specAttribute.setValue("MAK010"); // roads
dataRec.addEqualCondAttr(specAttribute);
DtVectorNetworkIntersectionRecord intersectionRec;
// Calculates intersection with terrain geometry and vector network
tdb2.intersect(chord, isectPoint, intersectionTime, &dataRec,
  &intersectionRec);
```

Querying the vector network is a bit different from querying the terrain skin. While the interface presented by both is named intersection, what is actually occurring with the vector network is not intersection, but rather selection based on criteria. As a result, to perform an actual intersection with vector features, the features must be selected from the vector network as described in this section, and then an actual intersection test must be done, using the properties of the features that define an intersectable object, presumably a volume (width, length, height).

13.4.6. Testing for All of the Terrain Intersections Along a Chord

You can perform intersection tests on the terrain database that return all of the points along a given chord that intersect the terrain database. This applies to terrain skin as well as features. Please see *terrainDb.h* for the set of all available intersection functions.

13.4.7. Using Metrics to Query the Vector Network

In addition to the intersection tests discussed in the previous sections, you can make more customized queries to the vector network using metrics. Metrics provide a mechanism to select elements from the vector network based on:

- Specification requirements: Qualifies a feature if the specification of the feature is valid for the current search. For example, is a road, is not a river, height is greater than zero, and so on.
- Measure: A numeric value used to qualify a feature based on some criteria, such as, distance to a point of interest, length of an edge, and so on.

DtMetric (*metric.h*) is the abstract base class for metrics. It allows feature attributes to be evaluated as equal, different, greater than, and lesser than. You can use the following predefined metrics to query the vector network:

- *DtRangeNetworkNodeMetric* (*rangeNetNodMet.h*) – Network nodes within a given range (for point features such as trees, buildings).
- *DtRangeNetworkSegmentMetric* (*rangeNetSegMet.h*) – *DtNetworkSegments* within a given range (for segments that may be part of linear features, such as a coastline or a road).

DtRangeNetworkNodeMetric and *DtRangeNetworkSegmentMetric* respect the **EvalInZ** parameter. If **EvalInZ** is false, they ignore the Z value in their calculations. If **EvalInZ** is true, they evaluate Z when calculating distance.

- *DtXyNetworkSegmentMetric* (*xyNetSegMet.h*) – *DtNetworkSegments*, sorted by increasing distance from a location, in XY only.

You can use these metrics to evaluate the vector network based on feature specification and range or distance to a point or chord of interest. They calculate intersection against the vector network and calculate paths (in the sense of paths through a graph) using the vector network.

For example, to retrieve all of the trees within range of a location:

```
// Create a range metric - we will use this to select features (nodes)
// from the vector network that fall within range of a location.
DtRangeNetworkNodeMetric nodeMetric;

// Create a specification attribute - we will use this to select
// the types of nodes in our metric. Select only those features having
// the MAK DtGroup specification of type “MAK040” (vegetation)
DtSpecificationStringAttribute sattr;
```

```
sattr.setLabel("DtFeatureGroup");
sattr.setValue("MAK040");

// Set the location of the center of our search for trees, in meters
nodeMetric.setCenterOfInterest(DtPoint(2000, 2000, 0));

// Set the range of the search, in meters
nodeMetric.setMaxRange(45);

// Take EvalInZ attribute into account in distance calculations
nodeMetric.setEvalInZ(true);

// Add a test for the specification attribute (that is, does the feature
// have a DtGroup attribute with a value of "MAK040"?)
nodeMetric.addEqualCondition(sattr);

// Query the vector network for the list of nodes using our metric. It
// returns all of the trees within 45 meters of location (2000, 2000, 0).
DtVectorNetwork* vectorNetwork = terrainInterface()->vectorNetwork();
DtList nodeList;
myVectorNetwork->collectAllNodes(nodeMetric, nodeList);
```

The node list contains pointers to all of the *DtNetworkNode* objects that meet the requirements specified in nodeMetric. The list is sorted in order of closest to furthest from the center of interest. You can add any number of specification attributes to your metric, to further refine your search.

13.5. Creating a New DtTerrainDatabase through the API

To create a new *DtTerrainDatabase* programmatically:

1. Create an empty terrain database object, for example:

```
DtTerrainDatabase* terrainDb = new DtTerrainDatabase();
terrainDb->init();
```

2. Choose the coordinate system type you want the terrain to have. This example, sets up a UTM coordinate system, with an origin of 35 degrees N, 122 degrees W.

```
DtDegMinSec latRef = {35.0, 0.0, 0.0, DtNorth};
DtDegMinSec lonRef = {122.0, 0.0, 0.0, DtWest};

SerializableUTMCS* coordinateSystem = new SerializableUTMCS(
  latRef, lonRef, 0);

terrainDb.setCoordinateSystem(coordinateSystem);
```

3. Add the terrain polygons or vector data to the terrain, as described in the following sections.

13.5.1. Adding Triangles to the Database

You add triangles to a terrain database through "create" functions exposed by the terrain database.

DtTerrainDatabase has member functions, such as createGroup() and createTriangle(), for creating the most commonly used node types. For all other node types, the *DtTerrainDatabase* has a newNode() member function, which takes a DtGdbNode::DtGdbNodeType() as an argument. It is the enumeration of all types of DtGdbNodes (*gdbNode.h*). Use of newNode() is discouraged in favor of the direct creation functions as they provide more stability in the case of future API changes.

Now you are ready to add polygons to your database. To do this, create a *DtTriangleIndirect* triangle (allocating through the *DtTerrainDatabase*), passing in its vertices and its surface, and optionally its parent node and a flag specifying whether or not to create it at the front of the parent's list of child nodes or not.

The order of vertices determines the polygon's orientation (the direction of the normal). Positive orientation follows a right-hand-rule.

```
// This triangle will have vertices, vert0, vert1, vert2, and a surface
// named surface; its parent will be the terrain databases root terrain
// group node since no other parent was specified, and it will be added to
// the end of the terrain group node's list of children nodes.
DtTriangleIndirect* tri1 = NULL;
tri1 = terrainDb.createTriangle(vert0, vert1, vert2, surface, NULL,
  false);
```

The default value for both the parent node* and the flag are NULL and false. Thus the above call could have been written as follows and had the exact same effect:

```
tri1 = terrainDb.createTriangle(vert0, vert1, vert2, surface);
if(tri1 == NULL)
{
  DtWarn("Terrain database failed to create a new triangle node.\n");
}
```

In this case, we added the triangle to the top of the terrain tree. To build up a tree structure in the terrain database, you need to add group nodes to the structure. In that case, you would add group nodes to the terrain, then add children to the appropriate groups, for example:

```
// createGroup takes an optional parent pointer as well. By leaving it
// NULL, the terrain database's root terrain node will be the parent.
DtGroup* group = terrainDb.createGroup(NULL);
// initialize the group
...
DtTriangleIndirect* triangle = terrainDb.createTriangle(  group);
// initialize the polygon

terrainDb.addTerrainNode(group);
```

You can only add a node to the database once. Nodes in the terrain database are stored in a tree structure. Graph-like loops are not permitted in the structure. However, you can have two identical nodes in two different locations.

13.5.2. Removing Terrain Nodes from the Database

To remove a node, you pass the pointer to the node and (optionally) the parent node in the tree structure.

```
terrainDb.removeTerrainNode(triangle);
```


Do not remove the node by deleting the pointer, for example:

```
delete(triangle);
```

The database has an internal memory manager that must be updated when a node is removed.

13.5.3. Adding Feature Data to the Database

The *DtTerrainDatabase* stores all of the vector network-related data in the *DtVectorNetwork*. It holds all of the network nodes and edges making up the features, as well as the *DtSpecificationManager* that contains the characteristics describing those features. To add a new feature to the vector network, allocate the appropriate kind of feature and configure its properties and specification. Then update the extents of the vector network to account for the feature as described in "Updating Extents in the Vector Network," on page 13-23.

```
// Create a specification for a tree
DtSpecification treeSpec;
treeSpec.setColor(DtColor(0,200,0,0)); // r,g,b,a (0-255)
treeSpec.setWidth(5.0);
treeSpec.setLength(5.0);
treeSpec.addIntegerAttribute("DtFidCode", 953);
treeSpec.addIntegerAttribute("DtFacNumber", 0);
treeSpec.addIntegerAttribute("DtFeatureType", 0);
treeSpec.addStringAttribute("DtFeatureName","953 Evergreen Trees
  (including Mangrove and Nipa)");
treeSpec.addIntegerAttribute("DtSmc", 12);
treeSpec.addStringAttribute("DtFeatureGroupName", "Vegetation");
treeSpec.addStringAttribute("DtFeatureGroup", "MAK040");
treeSpec.addStringAttribute("DtDescription","Trees");
treeSpec.addStringAttribute("ECC","EC030");
treeSpec.addIntegerAttribute("DtSedrisCodes", 1);
treeSpec.addStringAttribute("SEDIRSCODE1","TRE002");
treeSpec.addStringAttribute("TRE002","Evergreen");
treeSpec.addIntegerAttribute("DtOrientation", 0);
treeSpec.addIntegerAttribute("DtHeight", 5);

// Add the tree specification to the vector network's specification
// manager
DtSpecificationID treeSpecID;
if(!specificationManager.addSpecification(treeSpec, treeSpecID))
{
  DtWarn("Couldn't add tree specification to specification manager.\n");
  return false;
}

// Create a tree
DtNetworkNode* treeNode = vectorNetwork.newNode();

// Position it the upper right-hand quadrant of the terrain
treeNode->setLocation(DtPoint(750, 750, 0));
treeNode->setSpecificationID(treeSpecID);

// Add it to the database
vectorNetwork.addOn(treeNode);

// Create a river specification
DtSpecification riverSpec;
riverSpec.setColor(DtColor(0,0,200,0)); // r,g,b,a (0-255)
riverSpec.addIntegerAttribute("DtFidCode", 945);
riverSpec.addIntegerAttribute("DtFeatureType", 1);
riverSpec.addStringAttribute("DtFeatureName","945 Non-perennial Stream");
riverSpec.addIntegerAttribute("DtSmc", 6);
riverSpec.addStringAttribute("DtFeatureGroupName", "Waterway");
```

```
riverSpec.addStringAttribute("DtFeatureGroup", "MAK030");
riverSpec.addStringAttribute("DtDescription","River/Stream");
riverSpec.addStringAttribute("ECC","BH140");
riverSpec.addIntegerAttribute("DtSedrisCodes", 0);
riverSpec.addIntegerAttribute("DtOrientation", 0);
riverSpec.addIntegerAttribute("DtHeight", 0);
riverSpec.setWidth(10);
riverSpec.setLength(0);

DtSpecificationID riverSpecID;
if(!specificationManager.addSpecification(riverSpec, riverSpecID))
{
  DtWarn("Couldn't add river specification to the specification
    manager.\n");
  return false;
}

// Create the edge that makes up the river.
// Create the endpoints of our river, node1 and node2. Set the
// specification to the river specification we made above.
DtNetworkNode* riverStartNode = vectorNetwork.newNode();
riverStartNode->setLocation(DtPoint(0, 100, 0));
riverStartNode->setSpecificationID(riverSpecID);
vectorNetwork.addOn(riverStartNode);

DtNetworkNode* riverEndNode = vectorNetwork.newNode();
riverEndNode->setLocation(DtPoint(300,1000,0));
riverEndNode->setSpecificationID(riverSpecID);
vectorNetwork.addOn(riverEndNode);

// Create the edges. Like the segments, there are 2 edges (one is the
// reverse of the other). Configure it with endpoints and segments.
// Set the specification to the river specification we made above.
DtNetworkEdge* edge = vectorNetwork.newEdge();
edge->addPoint(DtPoint(0, 100, 0));
edge->addPoint(DtPoint(300,1000,0));
edge->setSpecificationID(riverSpecID);
edge->setStartNode(riverStartNode);
edge->setEndNode(riverEndNode);

// Unnecessary, since we've added on the elements above, but if not, we
// could have used the following:
vectorNetwork->calcExtent();
```

13.5.4. Updating Extents in the Vector Network

As you add nodes to the vector network, you are changing the extents of the groups making up the tree. You should update the extents before you make any searches or intersection tests with the vector network. There are two ways to update the extents:

- After you create a new element, call the function addOn() in the vector network to continuously update extents, as follows:

```
terrainDb().vectorNetwork()->addOn(node);
terrainDb().vectorNetwork()->addOn(edge);
```

- Create all elements in the vector network and call calcExtent() at the end.

The first mechanism is useful when you are adding elements dynamically and you want to maintain a valid vector network at all times. The second way to update the extents is more appropriate when you have separated the creation of the terrain database (such as in the code of a terrain reader) from the use of it. In this case, it is much more efficient to create all the elements first and then calculate the extents only once, at the end. For a more detailed set of examples, please see "Adding Feature Data to the Database," on page 13-21.

13.5.5. Removing Feature Data

To remove elements from the vector network, use the removal functions for nodes and edges.

```
terrainDb.removeFeatureNode(edge);
terrainDb.removeFeatureNode(node);
```

When you remove a feature, the element in the vector network is not physically deleted from the vector network; it is just marked as deleted.

You must consider the following issues when you remove an element from the vector network:

- If you want to remove a node, and it has any edges connected to it, then the edges will lose any connections to other edges through that node
- If an end or start node of an edge is removed, but the edge is not removed, a degenerate edge will result
- Removing an edge does not remove the nodes to which it is connected, but they will remove their references to the edge as a connection.

Restoring a Feature

You can restore (undelete) a feature in the vector network by calling the vector network's setAsRemoved() member function, for example:

```
terrainDb.vectorNetwork()->setAsRemoved(node, false);
```

This will not restore node/edge connections.

13.5.6. Postprocessing

After you build or load a *DtTerrainDatabase*, you can run a variety of post-processing routines. The *DtTerrainDatabase* class provides the following member functions for performing post-processing:

- eliminateDuplicateVertices() – deletes duplicate vertices.
- eliminateDuplicateSurfaces() – deletes duplicate surfaces.
- reduce() – deletes redundant nodes.
- balanceTerrain() – balances the tree of polygons.
- planarizeVectorNetwork() – planarizes the vector network data. It is required for DFAD and DFD data.
- clipVectorNetwork() – clips the vector data to the extent of the underlying polygonal terrain.

The choice of which processes to run depends on how the terrain is to be used. Each technique has advantages and disadvantages.

Clipping Vector Data

To clip vector data, use the clipVectorNetwork() functions.

Member function:	clipVectorNetwork()
Advantages:	Reduces the amount of memory necessary for the terrain. Vector network queries are also more efficient on a clipped network, if data was clipped out of the original.
Disadvantages:	Processing time required to clip the data. Clipped network is different from original source data.

Planarizing Vector Data

To planarize vector data, use the planarizeVectorNetwork() functions.

Member function:	planarizeVectorNetwork()
Description	Modifies the vector network, removing all intersections of edges not occurring at network nodes within the specified tolerance. Also flattens the vector network (hence planarize). It ensures that the vector network elements are properly constructed and that all invariants are maintained.
Advantages:	Fixes connectivity/intersection errors in the source data. Successful planarizing ensures that the vector network is properly constructed.
Disadvantages:	Processing time can be very long for large and/or complex vector networks. The original data is altered.

13.6. Terrain Readers

Terrain database files are imported using *DtTerrainReaders*. A *DtTerrainReader* (*terrainReader.h*) knows how to import terrain geometry data (terrain skin) from an external format into a *DtTerrainDatabase* object. VR-Forces has terrain readers for the following formats:

- Digital Terrain Elevation Data (DTED)
- MultiGen-Paradigm OpenFlight (FLT) and MetaFlight
- Compact Terrain Database (CTDB) versions 4b, 7b, and 7l
- MÄK Terrain Format (GDB)
- Shapefile (SHP)
- Extent.

For example, to load an OpenFlight database, using a terrain reader, do the following:

```
// Create and initialize an empty terrain database
DtTerrainDatabase terrainDb;
terrainDb.init();

// Load the terrain database using a terrain reader that knows how to load
// Openflight files.
DtFltReader reader;
if(!reader.loadTerrain("HL_94.flt.flt", &terrainDb))
{
  DtWarn("Couldn't load terrain database file\n");
  return 1;
}
```

DtTerrainReader is an abstract base class. It defines the interface for initializing and loading a terrain file. The key member function that derived *DtTerrainReader* classes implement is loadTerrain(). The loadTerrain() member function takes the filename of a terrain data file to import, and a pointer to a newly created and initialized *DtTerrainDatabase*, and optionally a pointer to a *DtTerrainInitializer* (a helper class that contains import parameters).

```
loadTerrain(const char* filename, DtTerrainDatabase* terrain);
loadTerrain(const char* filename, DtTerrainDatabase* terrain, const
  DtTerrainInitializer* initializer);
```

In loadTerrain(), derived *DtTerrainReader* classes open and read terrain data from a file. They make standard API calls to the *DtTerrainDatabase* object, creating and adding *DtGdbNodes* to create the terrain.

13.6.1. Creating a New Kind of Terrain Reader

To create your own kind of *DtTerrainReader*, you need to implement a few *DtTerrainReader* member functions, and then add a creator function for your new reader to the *DtTerrainReaderFactory*, so that VR-Forces can find it when it needs to load your terrain file format. The *DtTerrainReader* member functions you must implement are:

- dataType() – Returns a string used to identify the type of terrain reader (for example, "shape", "flt"). It must be unique across all *DtTerrainReader* types in the factory.
- create() – Returns a newly created instance of the derived *DtTerrainReader* class. This is the function to be registered with the factory.
- loadTerrain() – Opens and reads in data from a terrain data file, and uses that data to construct a *DtTerrainDatabase*. Creates and adds *DtGdbNodes* to a given *DtTerrainDatabase*.

After you implement a *DtTerrainReader* class, you must add it to the terrain reader factory. To register your new terrain reader with the factory, you add the create function to the factory, specifying the string returned by dataType() as the key. For example, to add a creator for a new reader class MyTerrainReader:

```
cgf.factoryManager()->terrainReaderFactory()->
  addTerrainReaderCreatorFcn("myformat", MyTerrainReader::create);
```

13.6.2. Terrain Feature Readers

Terrain features (vector data) are read in to a *DtTerrainDatabase* using *DtVectorDataReaders*. *DtVectorDataReader* (*vectorDataReader.h*) is an abstract base class. It defines the interface for initializing and loading vector data into a *DtTerrainDatabase* object from an external terrain file. VR-Forces has the following predefined vector data readers:

- ESRI Shapefile: *DtShapeVectorReader* (*shapeVectorReader.h*)
- DFAD: *DtDfadReader* (*dfadReader.h*)
- VMAP: *DtVmapReader* (*vmapVectorReader.h*)
- MultiGen-Paradigm DFD: *DtDfdDfadReader* (*dfdReader.h*).

The key member function that derived *DtVectorDataReader* classes implement is importVectorData(). The importVectorData() function takes the filename of a terrain data file to import, a pointer to a newly created and initialized *DtTerrainDatabase*, and a pointer to a *DtTerrainInitializer* (*terrainInitializer.h*)(a helper class that contains import parameters).

```
importVectorData(const char* filename, DtTerrainDatabase* terrain,
  const DtTerrainInitializer* initializer);
```

In importVectorData(), derived *DtVectorDataReader* classes open and read terrain vector data from the specified file. They make standard API calls to the *DtTerrainDatabase* object, creating and adding nodes, segments, and edges to the vector network, necessary to create the terrain. The terrain object passed in must be initialized and already hold a valid terrain geometry (terrain skin).

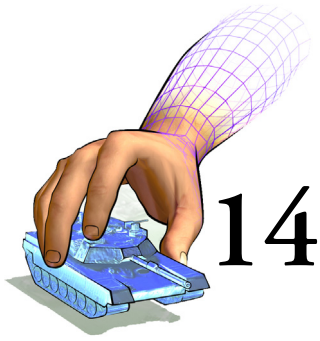
Creating a New Kind of Terrain Feature Reader

To create your own kind of *DtVectorDataReader*, you must implement the following *DtVectorDataReader* member functions, and then add a creator function for your new reader to the *DtVectorDataFactory*:

- dataType() – Returns a string used to identify the type of terrain reader (for example, “dfad”). It must be unique across all *DtVectorDataReader* types in the factory.
- create() – Returns a newly created instance of the derived *DtVectorDataReader* class. This is the function to be registered with the factory.
- importVectorData(DtFilename &path, DtTerrainDatabase *terrain, DtVector-DataInitializer *myInit) – Opens and reads data from a terrain vector data file, and uses that data to construct the vector network for the *DtTerrainDatabase*.

Once you have implemented your own *DtVectorDataReader* class, you must add it to the vector data reader factory. To register your new vector data reader with the factory, you add the create function to the factory, specifying the string returned by dataType() as the key. For example, to add a creator for a new reader class MyVectorDataReader:

```
cgf.factoryManager()->vectorDataFactory()->
  addVectorDataCreatorFcn(“myVecFormat”, MyVectorDataReader::create);
```



14

Reading and Writing Files

This chapter describes the ability of VR-Forces to read information from files and write information to files.

14.1. Readable, Writable Objects

VR-Forces uses the object parameter database, various configuration files, and scenario files to store information it needs to load scenarios, instantiate objects, and represent objects in the GUI. (For details about the object parameter database, please see "Object Parameters," on page 3-13.) To enable objects to be written to a file and read from a file, VR-Forces uses a class called *DtReaderWriter* (*rdrWritr.h*), which all readable, writable objects derive from.

DtReaderWriter provides four member functions – putSelf(), putData(), getSelf(), and getData() that read and write the object. Normally, putSelf() and getSelf() do not need to be overridden. Override getData() and putData() when you need to put and get class-specific data.

DtReaderWriter uses the VT MÄK Lisp (MTL) file mechanism for parsing the files it reads. To do so, data must be written in list format. Every readable, writable object has a name, which is used as the first element of the list. The name is provided to the constructor of *DtReaderWriter*. (A *DtReaderWriter* object that is going to be read can be specified with a null string as its name. When the object is read, it gets the name specified in the file.)

14.2. Writing Data to a File

The putSelf() and putData() member functions write data to a file, as follows:

1. Write an appropriate number of blanks based on indentLevel.
2. Write an open parenthesis.
3. Write the object name.
4. Call the putData() member function. The putData() member function is responsible for writing the data value or values for the object.
5. Tell the *DtReaderWriter*'s registry (described later in this section) to put itself.
6. Write a final closing parenthesis.

The putData() member function is specified as:

```
virtual void putData(FILE *fp, int indentLevel = 0);
```

Example 14.1 shows the putData() member function for the *DtRwString* (*rwString.h*) class, provided by VR-Forces.

Example 14.1

```
void DtRwString::putData(FILE * fp, int)
{
   fprintf(fp, " \"%s\"", string());
}
```

14.3. Reading Data from a File

The getSelf() and getData() member functions read data from a file. The getSelf() member function works similarly to putSelf(), except that it operates on an MTL stream instead of an open file pointer.

1. VR-Forces opens, reads, and converts the file being read into an MTL stream.

   The following example shows how to open an MTL file:

   ```
   FILE * fp;
   DtInputStream * fStream;
   fp = fopen("name of you file", "r");
   fStream = new DtExtFileInputStream(fp);
   DtMtlEnvironment mtlEnv;
   getSelf(DtcRead(fStream, mtlEnv.env()));
   ```

2. The routines that handle MTL (*mtl.h*) retrieve the data.

3. The getSelf() member function calls getData(), and then the *DtReaderWriter*'s registry getSelf().

The getData() specification is:

```
virtual DtlObjPtr getData(DtlObjPtr args);
```

Example 14.2 illustrates the getData() member function for *DtRwString*.

Example 14.2

```
// args points to the lisp expression => (name "dataString")
// searchPath is not used in this example
DtlObjPtr DtRwString::getData(DtlObjPtr args, const DtFilename&
  searchPath)
{
  DtlObjPtr currentObject;
  DtlObjPtr remainderList;
  DtlObjPtr dataList;

  dataList = DtcCdr(args);   // dataList now points to =>("dataString")

  currentObject = DtcCar(dataList); // currentObject now points to
                                    //=>"dataString"
  remainderList = DtcCdr(dataList); // remainderList now points to =>()

  // error check to make sure currentObject is a string
    if (currentObject ->tag() == DtpString)
  {
    setString(currentObject ->string());
  }

  else
  {
    DtWarn("DtRwString::getSelf(): bad data stream\n");
    DtcErrPrint(args);
  }

  return remainderList;
}
```

Suppose that the "args" *DtlObjPtr* in Example 14.2 points to the LISP expression:

```
(name "dataString")
```

The "car" of this expression (retrieved with DtcCar(args)) is **name** (no parentheses) and the "cdr" (DtcCdr(args)) of the expression is ("**dataString**"). You do not need the name. (getSelf() will have grabbed it if the object was created without a name.) The local variable *dataList* is set to point to the list of data arguments. (In this case, there is only one element in the list.) The "car" of this expression is the data we want, and the "cdr" of this expression gets returned.

Example 14.3 shows an excerpt of a getData() call for an arbitrary list of numbers (from *DtNLengthIntegerVector*).

Example 14.3

```
// searchPath is not used in this example
DtlObjPtr DtNLengthIntegerVector::getData(DtlObjPtr args, const
  DtFilename& searchPath)
{
  // this gives us the list of numbers (gets us past the name)
  DtlObjPtr current = DtcCdr(args);

  // setVectorLength allocates the correct-sized array of integers
  // in myIntegers, and also initializes myVectorLength; DtcLength
  // returns the number of elements in the lisp expression 'current'
  setVectorLength(DtcLength(current));

  if (myVectorLength > 0)
  {
    // get each value
    for (int i = 0; i < myVectorLength; i++)
    {
      myIntegers[i] = DtcCar(current)->fixnum();
      current = DtcCdr(current);
    }
  }

  return current;
}
```

Although it was not used in the examples in this section, you must specify searchPath in the getData() call. searchPath is the name of a directory to search when loading files specified in *DtReaderWriterFile* objects. It is passed in to all *DtReaderWriter* objects because they may need to pass it down to any child *DtReaderWriterFile* nodes in its registry.

14.4. Multiple Inheritance and DtReaderWriter

While we normally avoid the use of multiple inheritance, several of the basic readable, writable types provided with VR-Forces (such as *DtRwString*) were created from existing classes by inheriting both *DtReaderWriter* and the existing class (*DtString*, for example, in the case of *DtRwString*). Remember that if you cast an object constructed using multiple inheritance to a `void *` (for example, for inclusion on a VR-Link *DtList*) you must cast back to the original constructed type, not one of its ancestors. For example, the following will compile, but the result is undefined:

```
DtRwString * createString = new DtRwString("Test String");
void * tempVoid = createString;
...

DtString * useString = (DtString *) tempVoid;
printf (useString->string());
```

For correct results, replace the last two lines with:

```
DtRwString * useString = (DtRwString *) tempVoid;
printf (useString->String());
```

14.5. The Reader/Writer Registry

To allow readable, writable objects to easily contain other readable, writable objects, the *DtReaderWriter* creates an instance of *DtReaderWriterRegistry* (*rwRegistry.h*). The *DtReaderWriter* registry is a list of *DtReaderWriter* pointers. (Actually it is a hash list indexed by the names of the objects.) The constructor for a *DtReaderWriter* takes a registry pointer as an argument. If this pointer is non-null, the *DtReaderWriter* object registers itself with the registry pointer provided. In this way, you can compose new readable, writable objects from existing ones. It is fairly common, in fact, to have objects that generate no new readable, writable data, and therefore do not need to override putData() and getData(). Example 14.4 shows the key elements needed to make a new readable, writable class that contains a real number and an integer number, using the existing *DtRwReal* and *DtRwInteger* classes.

We have our new class's constructor take a pointer to a *DtReaderWriterRegistry* so that it could be used as part of some other readable, writable class.

Example 14.4

In the header file *realAndInt.h*, we do the following:

```
#include "rwReal.h"
#include "rwInt.h"

class RealAndInt : public DtReaderWriter
{
public:
  RealAndInt(const DtString& name,
    DtReaderWriterRegistry* parentRegistry = 0);

  ...
protected:
  DtRwReal myRealNumber;
  DtRwInt myIntegerNumner;
};
```

In the implementation file *realAndInt.cxx*, we do the following:

```
#include "realAndInt.h"

RealAndInt::RealAndInt(const DtString& name,
  DtReaderWriterRegistry* parentRegistry)  :
  DtReaderWriter(name, parentRegistry),
  myRealNumber("real-number", &myRegistry),
  myIntegerNumber("integer-number", &myRegistry)
{
  myIntegerNumber = 0;
  myRealNumber = 0.0;
}
...
```

The **myRegistry** variable referred to in the constructor is a member variable of the *DtReaderWriter* class that the *RealAndInt* class inherits. An example of its use and sample output, assuming an already open file pointer fp, as follows:

```
...
RealAndInt  twoNumbers("two-numbers");

twoNumbers.setReal(7.66);
twoNumbers.setInteger(18);
twoNumbers.putSelf(fp);
...
```

would write to the open file pointed to by fp, as follows:

```
(two-numbers
   (real-number 7.66)
   (integer-number 18)
)
```


The names used for *DtReaderWriter* objects cannot contain spaces, and must not start with a digit.

If your object does not contain any local data that it wants to write other than the readable, writable objects it uses, there is no need to override putData() and getData(), and in fact, no need for putSelf() and getSelf() to invoke these member functions. By default, these functions are not called. To indicate that an object has local data and wants getData() or putData() to be called, call the following, typically in the constructor of the object:

```
myRegistry.setLocalData();
```

14.6. Handling Unspecified Parameters

There may be times when a DtReaderWriter object expects to read a value from a file and the file does not contain the expected value. A *DtReaderWriter* object can keep track of whether or not it has received a value from a file. In Example 14.4, the class created could read the following input without error (note that the integer-number entry is missing):

```
(two-numbers
   (real-number 7.66)
)
```

The *DtReaderWriter* does not require that every registered item get a value from the data stream. When data is found for a registered entry, *DtReaderWriter* sets a flag to `true` to indicate this. If you want to detect unread entries, set this flag to false for the object before getSelf() is called (the constructor is a good place to do this). Since some objects may want this functionality and others may not, it is a good practice to let the including object set these flags, rather then setting them in the constructor of the base object itself. In our *RealAndInt* class, in the constructor, we could add the following:

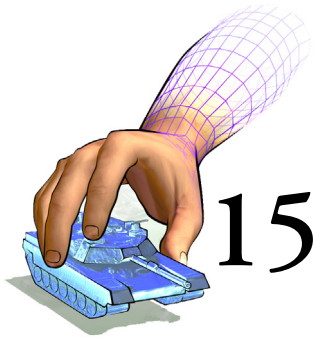
```
myRealNumber.setValueSpecified(false);
myIntegerNumber.setValueSpecified(false);
```

You could also call the following:

```
myRealNumber.setValueSpecified(true);
```

in the mutator that sets **myRealNumber**'s value, if that is also a valid way to get a value in your class. Finally you can test whether or not a value has been set as follows:

```
if (myRealNumber.valueSpecified())
{
  // do this only if a value was read from the file or set in the mutator
  // function
     ...
}
```



15

Miscellaneous Classes and Utilities

This chapter introduces you to the Simulation Manager and other top-level objects.

15.1. The Simulation Manager

Each VR-Forces application has a Simulation Manager (*DtSimManager*, in *simMgr.h*). The Simulation Manager maintains the exercise clock and the scheduler, as well as providing access to some of the top-level managers and classes. The principal components that the Simulation Manager creates and maintains are the:

- Exercise clock – keeps track of elapsed simulation time and simulated exercise time
- Scheduler – allows you to schedule events
- Message Interface – used to send VR-Forces-specific messages.

The Simulation Manager also provides accessors to the following items:

- Exercise connection – used to send/receive PDUs/HLA objects and interactions
- Terrain – access to the terrain database
- Object Manager – manages the set of all simulated entities, aggregates, and control objects
- Joystick Device Manager – provides a device-independent joystick interface that can be used to control entities.

Each time the Simulation Manager's tick() function is called, it does the following:

1. Ticks its exercise clock and scheduler.
2. Calls any functions on its tick list.

The Simulation Manager is one of the top-level classes maintained by the *DtCgf* class. You can obtain a pointer to it through DtCgf::simManager().

15.1.1. Simulation Time, Exercise Time, and the Exercise Clock

VR-Forces maintains an exercise clock, which is an instance of *DtExerciseClock* (*exClock.h*). The clock keeps track of elapsed simulation time and simulated exercise time. Elapsed simulation time is the amount of simulation time that has passed since the simulation started. Simulated exercise time is the simulated time of day, and is simply the elapsed simulation time added to a user-specified exercise start time. Time is represented by the VR-Link DtTime type (in seconds), and is advanced when the clock is ticked. The exercise clock also provides static functions for converting seconds into the days:hours:minutes:seconds format.

The exercise clock supports several different modes of operation. The mode the clock is in determines how much simulation time is advanced when the clock is ticked. These modes are described in detail in Chapter 3, *VR-Forces Simulation Concepts*, in *VR-Forces Users Guide*.

15.1.2. Scheduling Events with the Simulation Engine

As described in Chapter 2, *The VR-Forces Simulation Engine*, the only function that needs to be called periodically for the simulation engine to execute properly is DtCgf::tick(). This causes managers and objects to be ticked, for example, *DtVrfObjectManager* (and therefore *DtVrfObjects* and their components). If you have registered custom subclasses of any of the VR-Forces classes with the simulation engine (either through factories, or through *DtVrfCreator*), instances of your subclasses will automatically be ticked at the appropriate times.

However, there may be cases where you want objects that exist outside of the VR-Forces simulation engine to be automatically ticked each time the simulation engine is ticked. VR-Forces provides a tick callback mechanism to support this. You can obtain a pointer to the *DtCgf*'s *DtSimManager*, and use its addTickEntry() member function to register the callback that you would like to have executed each frame. In fact, the way that VR-Forces classes like *DtVrfObjectManager* ensure that they are ticked each frame is by registering their own tick callbacks with the *DtCgf*'s *DtSimManager*.

The following example shows how to add a function named tickCallback() that will get invoked each tick:

```
// Create your function to be invoked each tick. It must have this
// function signature:
void tickCallback(void* usr)
{
  // add code that does something each tick here
}
```

Register your function with the *DtSimManager*. You can optionally configure the callback with a pointer to some user-defined data (usr). This data pointer will be passed into the callback function when it is invoked.

```
cgf.simManager()->addTickEntry(tickCallback, usr);
```

When you no longer want your callback to be called, unregister the callback function with the *DtSimManager*, as follows:

```
cgf.simManager()->addTickEntry(tickCallback, usr);
```

Registering Periodic or Recurring Timers

You might want VR-Forces to invoke a callback whenever some amount of time has passed, but not necessarily every frame. VR-Forces uses timers for this purpose. Through the *DtCgf*'s *DtSimManager*, you can get a pointer to a *DtScheduler* (defined in *scheduler.h*) that you can use to schedule events. Events are scheduled by elapsed simulation time.

The Scheduler manages instances of *DtTimer* (*timer.h*) objects, and its use is optional. A *DtTimer* is given a time to fire, a callback function to call when it fires, and a pointer to user data that will be provided to the callback. You can tick the timer directly each simulation frame, or place the timer on the Scheduler.

The Scheduler maintains a list of *DtTimers* given to it. The Scheduler ticks its timers when it is ticked. The timers are listed in the order they will fire. It is your responsibility to remove timers from the Scheduler once they have fired.

You can designate a *DtTimer* as recurring, *DtRecurringTimer* (*recTimer.h*) or periodic, *DtPeriodicTimer* (*periTimer.h*). Recurring and periodic timers are rescheduled by the scheduler after they fire.

Recurring timers are timers that, after firing, are rescheduled to fire at a specified time after the actual firing time. Periodic timers are timers that, after firing, are rescheduled to fire at a specified time after the last time the timer was scheduled to fire. Figure 15-1 illustrates the difference.

If a timer fires at exactly the time that it is supposed to, the two types of timers are equivalent. However, timers can only fire when the scheduler is ticked. Since the scheduler is ticked once per frame, there is a potential error of +/- the current frame time between the scheduled time of fire and the actual time of fire. The two different types of timers provide two different ways of dealing with this error.

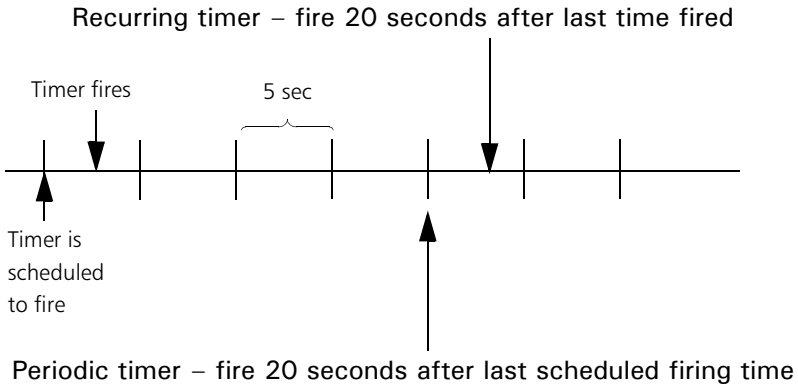


Figure 15-1. Recurring and periodic timers

15.1.3. Publishing a Back-End's Status

The *DtVrfStatusPublisher* class sends out the *DtIfStatus* message in a heartbeat and as a response to requests for status.

15.1.4. The Joystick Device Manager

The Joystick Device Manager, *DtJoyDeviceManager* (*joyDevMgr.h*), provides a hardware independent interface for using a joystick with VR-Forces. Entities that support joystick control access the *DtJoyDeviceManager* to get control input data from the device.

15.2. The Settings Manager

The Settings Manager *DtSettingsManager* (*settingsManager.h*) lets you save run-time settings in an XML file. It is based on the boost::serialization library (for details, please see "The Boost Serialization Library," on page 15-19). The *DtVrfSettingsManager* (*vrfSettingsManager.h*) is a subclass that lets you remotely query and edit settings through custom messages that are supported by *DtVrfRemoteController*. The *DtVrfSimSettingsManager* (*vrfSimSettingsManager.h*) is the subclass that is used by the VR-Forces back-end. It specifies the name of the back-end settings file (*vrfSimSettings.xml*) and registers all the independent settings used by the back-end (for details, please see "Independent Settings," on page 15-8).

Related settings are grouped into a settings object derived from *DtSettingsObject* (*settingsObject.h*). For instance, a dead reckoning object would have the translation, rotation, and aggregate threshold values. If a class wants to register a single value without creating a settings object class, it can use *DtBoolSetting* (*boolSetting.h*), *DtIntSetting* (*intSetting.h*), *DtDoubleSetting* (*doubleSetting.h*), or *DtStringSetting* (*stringSetting.h*). The settings objects are there strictly for serialization – it is expected that the current values will be stored in a class that uses the Settings Manager to save them. When the current values for the settings objects are needed (to write out the settings file or to respond to a front-end request), the settings object is passed back to the client that is responsible for updating it. This means that the client does not have to ensure that the settings object is always up to date – it just needs to provide a function to do the updating. *DtSettingsObjects* are created by a *DtSettingsObjectFactory* (*settingsObjectFactory.h*).

15.2.1. The Back-End Settings Manager (*DtVrfSimSettingsManager*)

The *DtSettingsManager* class can be used from any application. *DtVrfSimSettingsManager* is the subclass that is specific to the VR-Forces back-end. The back-end needs to have persistent storage for back-end-wide settings. The settings stored in this file represent the run-time configuration of a back-end. In most cases these are settings provided from a front-end, but any case in which a back-end setting is changed at run time would apply. These settings are not meant to affect the back-end before its initialization.

The functionality of this file is as follows:

- It is named *vrfSimSettings.xml*.
- VR-Forces does not install a default version of this file – it is written to *./appData/settings/vrfSim* the first time a back-end is closed.
- It affects all back-ends run relative to the *./config* directory where it resides. For example if you start three back-ends from the same location, all three will start with the same settings.
- Saved settings are not specific to a back-end. The last back-end to write out the file overwrites the settings saved by previous back-ends. For example if back-end A and back-end B are both running from *./bin*, if A is closed before B, the settings saved by A will be overwritten by B.
- Deleting the settings file causes the next back-end that is started to use a default configuration.

15.2.2. Creating Settings

The Settings Manager can save two types of settings – those that are owned by a client of the Settings Manager, and those that are maintained by the Settings Manager itself.

Client-Owned Settings

Any *DtSettingsObject* that will be produced by a client of the *DtSettingsManager* must be registered with the Settings Manager by the client. The client calls DtSettingsManager::registerProducer() with the type and the name of the setting and a *DtSettingsProducerFcn* and `void* user` pointer. The *DtSettingsObject* is not passed in; the client never transfers an actual object instance. The client just needs to tell the Settings Manager that it is responsible for the setting. The Settings Manager creates the object when necessary, and calls the *DtSettingsProducerFcn* any time it needs to access the value. For example, this happens when a call to DtSettingsManager::lookupSetting() or write() is made.

Independent Settings

DtSettingsManager can also maintain settings on behalf of a client. In general this is only done when there is no central object that is responsible for the setting. For example, the spot reports "on/off" state is used by every *DtVrfObject* that has a spot report generator, but there is no central spot report manager class to maintain this state. In this case DtVrfSimSettingsManager::init() calls DtSettingsManager::addIndependentBool() to add a *DtBoolSettingsObject* to the map of settings objects. There is no producer responsible for updating this value, however it can be changed by calling DtSettingsManager::setSetting() or through the remote setting mechanism.

15.2.3. Accessing Settings

Any object that wants to adjust itself based on a *DtSettingsObject* can access the *DtVrfSimSettingsManager* through the *DtSimManager*. This could happen at any time, but it is expected that this would usually happen during initialization. DtSettingsManager::lookupSetting() takes the name of the object the client is looking for. If the file fails to read, a null pointer is always returned to indicate that the value was not read in. If the name lookup in the settings map is successful, a const pointer to the requested member is returned.

15.2.4. Getting Notification of Changes to Settings

Any object that wants to get a callback when a setting is changed can register itself as a consumer of that setting by calling DtSettingsManager::registerConsumer(). If the setting is designed to be changed remotely (for details, please see "Querying Settings from Remote Applications," on page 15-9), the client registered as a producer must also register as a consumer so it can change itself when the setting is modified. The consumer function takes the name of the setting to be monitored, a *DtSettingsConsumerFcn* and a `void* user` pointer. These values are stored in a multi-map inside *DtSettingsManager*, which means that more than one *DtSettingsConsumerFcn* can be registered on a single setting. This makes it easier to support settings on an individual entity or component level, which do not have an intermediate manager class to deal them. The *DtSettingsConsumerFcn* returns a const *DtSettingsObject* and the `void* user` pointer to allow the client to update itself based on the new values.

15.2.5. Querying Settings from Remote Applications

The Settings Manager supports queries from *DtVrfRemoteController* based on string keys, and returns a copy of the same settings object being used in the back-end. This saves a lot of time because the custom messaging and callbacks required for the front-end to interrogate the back-ends are not necessary for each type of setting that needs to be transferred. The values returned are processed by the *DtVrfBackendListener*. The *DtBackend* class maintains a std::map() of *DtVrfSimSettingsObjects* for each back-end. *DtBackend* has a lookup method that takes the name of a settings object and returns a const pointer to the object, or 0 if it does not exist in the map.

DtVrfSimSettingsManager registers for the DtIfRequestSetting message, which allows retrieval of settings objects returned through the DtIfSetting message. DtIfRequestSetting contains a string that corresponds to a name of a setting registered in the manager. The DtIfSetting message contains a serialized copy of the settings object. When the DtVrfSimSettingsManager processes a DtIfRequestSetting message, the settings object is looked up through DtSettingsManager::lookupSetting() to insure that its DtSettingsProducerFcn gets called, so that the latest data from the settings producer is returned.

DtVrfRemoteController has a requestBackendSetting() method that takes a string that corresponds to the name of the DtSettingsObject, a DtSimulationAddress to address the query to, and a callback function, which will be called once the responses are received. In most cases, the address will be DtSimSendToAll, but it is also possible to pass in the address of a specific back-end. The DtVrfRemoteController passes the request to the DtVrfBackendListener. The DtVrfBackendListener receives the DtIfSettingResponse messages and stores the settings objects received in the corresponding DtBackends. It keeps track of each request made including the name of the setting requested, the back-ends that have not yet responded, and the callback function to call when the request is completed. Once the request is completed, the callback is called to indicate that all the responses have been received and it is now safe to iterate over the DtBackends and look up the setting by calling DtBackend::lookupSetting().

15.2.6. Modifying Settings from Remote Applications

Remote VR-Forces applications can update the state of client objects by sending a DtIfModifySetting message. This message carries a serialized *DtSettingsObject*. The *DtVrfSimSettingsManager* listens for the modification messages, deserializes their payloads, and updates the existing object with the new one by calling DtSettingsManager::setSetting(). Objects that do not match any existing objects by name are ignored.

DtVrfRemoteController has a modifyBackEndSettingMethod() method, which takes a DtSettingsObject and a DtSimulationAddress. The object's name must correspond to a pre-existing back-end setting. The address defaults to DtSimSendToAll, but can also be targeted at specific back-ends. DtVrfRemoteController also has specialized methods to make the modify call on behalf of the user, for example setSpotReportsGloballyEnabled().

15.2.7. Examples

The settingsManager example demonstrates many techniques for working with the Settings Manager. The settingsManagerSim portion of the example is a back-end plug-in which demonstrates adding a new setting type, extending a built-in type, and creating a new instance of a Settings Manager which writes out its own file. The settingsManagerRemote portion of the example is a minimal front-end application which queries and modifies the Settings Managers in a back-end running with the settingsManagerSim plug-in.

15.3. Spatial Subdivisions

The *DtSpatialSubdivision* class represents a uniform spatial container. The container is divided up into a lattice of cells, with resolution specified at creation. Each cell is a container of the objects in it. Its purpose is to efficiently store and search objects spatially.

The *DtSpatialSubdivision* is a templated container, similar to the std::list or std::vector classes. As such, it can hold any specified item. The intended use is to store elements that have spatial properties – elements that have an actual volume, position, orientation, and so on.

The responsibility for and intelligence necessary to perform operations on the spatial subdivision, such as intersection with primitives, insertion of elements into, and so on, is in specific classes for the distinct operations. This greatly facilitates extending existing capabilities and adding new ones. As a result, the spatial subdivision functionality is spread out across an entire family of classes, as described in this section.

15.3.1. The Spatial Subdivision Container

The *DtSpatialSubdivision* is the container class. It is defined in *spatialSubdivision.h*, in the geometry library.

Spatial Subdivision Intersectors

The spatial subdivision intersector classes know how to intersect a primitive and a spatial subdivision, and apply a specified operation to the results of the intersection through a functor interface.

The functor interface allows for tremendous flexibility in the type and complexity of operations that are performed on the spatial subdivision. By changing the functor supplied to the intersect() member function, different operations can be performed, such as inserting elements into the spatial subdivision, gathering elements of specific characteristics from the spatial subdivision, or simply accumulating all cells in the spatial subdivision that are part of the intersection. The following are examples of intersectors:

- *DtSsChordIntersector* (*ssChordIntersector.h*) – This class intersects a specified chord and spatial subdivision, applying a functor to the intersection results.
- *DtSsExtentIntersector* (*ssExtentIntersector.h*) – This class intersects a specified extent and spatial subdivision, applying a functor to the intersection results. Note that this class also provides a function for determining the min/max intersected cell indices, which can sometimes be more useful than the actual full intersection call.

Spatial Subdivision Functors

The spatial subdivision functors are simple classes that are used by the spatial subdivision intersectors to perform different operations on the results of intersection tests. These operations include insertion of elements into spatial subdivision cells (*DtSsInsertionFunctor*), accumulation of spatial subdivision cells for later analysis (*DtSsAccumulationFunctor*), among others.

While simple, these functors differ from standard functors in the following important ways:

- The functors are passed by *reference*, not by value. They do not have to maintain state, but they are expected to.
- The functors return a Boolean to indicate whether or not they are done processing inputs, and *not* whether or not an error occurred. This is to allow for efficient processing of intersection results. For instance if a functor is looking for any intersection of a chord and a polygon, it can stop processing after the first intersection is found, simply by returning true. To determine if an error occurred, the functor *must *either maintain internal error state or use exceptions.

The following are examples of functors:

- *DtSsFunctor* (*ssFunctor.h*) – This is the base class for all spatial subdivision functors. It does not perform any operation.
- *DtSsAccumulationFunctor* (*ssAccumulationFunctor.h*) – *DtSsAccumulationFunctor* contains a container, which it uses to store elements passed in to its function operator (please see the function definition for more details). Thus it can be used to accumulate the set of cells comprising the intersection of a primitive with the spatial subdivision.
- *DtSsInsertionFunctor* (*ssInsertionFunctor.h*) – *DtSsInsertionFunctor* is used to insert one object into another, for instance in inserting elements into the spatial subdivision cells.

Spatial Subdivision Inserters

The spatial subdivision inserter classes are utility classes that combine the appropriate intersector with an insertion functor, in order to facilitate inserting of elements of the appropriate type into a spatial subdivision.

15.3.2. Creating Spatial Subdivision Objects

To create a spatial subdivision, do the following:

```
DtSpatialSubdivision<DtGdbNode*>* spatialSubdivision = 0;
spatialSubdivision = new DtSpatialSubdivision<DtGdbNode*>(params);
```

To change the type of contained element, change the template parameter – *DtGdbnode**.

Inserting Elements into the Container

To insert an element into a spatial subdivision container, you use an inserter class. The following example show how the terrain database now creates the spatial subdivision for the terrain polygons. The terrain() function returns the root terrain polygon node (*DtGdbNode**).

```
DtSsGdbNodeInserter terrainInserter;
if (!terrainInserter.insert(terrain(), *myTerrainSpatialSubdivision))
{
  error handling code...
}
```

Intersecting *DtChords* and *DtExtents* Directly with the Container

To intersect a chord with an spatial subdivision, create an instance of a *DtSsChordIntersector* (*ssChordIntersector.h*, and call its intersect function with the desired chord, spatial subdivision, and functor. Please see the class documentation for more information.

Intersecting *DtExtents* with the Container

To intersect an extent with an spatial subdivision, create an instance of an *DtSsExtentIntersector* (*ssExtentIntersector.h*) and call its intersect function with the desired extent, spatial subdivision, and functor. Please see the class documentation for more information.

Extents are the only non-chord primitive for which intersectors are provided. While other intersectors exist for *DtVrfObjects* and *DtVectorObstructions*, all create extents from the base objects that are then used to intersect the spatial subdivision.

15.3.3. Extending the *DtSpatialSubdivision* Classes

Because the spatial subdivision is a templated container, creating a spatial subdivision of a new type of element is as easy as altering the declaration.

For example, the following creates a spatial subdivision of pointers to *DtGdbNodes*:

```
new DtSpatialSubdivision<DtGdbNode*> gdbNodeSpatialSub;
```

The following creates a spatial subdivision of pointers to *DtVectorObstructions*:

```
new DtSpatialSubdivision<DtVectorObstruction*>
  vectorObstructionSpatialSub;
```

Creating a New Type of *DtSsfunctor*

Derive the new functor from the *DtSsFunctor* class. To use this functor, pass it as a parameter to an existing spatial subdivision intersector or other utility class, or to a newly created spatial subdivision utility class.

Be sure that the new functor adheres to the interface of the *DtSsFunctor* classes.

Creating a New Intersector

Creating new intersector classes is probably one of the most useful extensions to the spatial subdivision classes. For instance, an intersector class for spheres or cylinders or other implicit surfaces would enable efficient selection of all elements within a specified radius of a location (or line).

To create a new intersector class, there is no need to derive from any parent class. The intersector class merely needs to know how to intersect the specified primitive with the spatial subdivision and to allow user-specified processing of the results.

So, to create a sphere intersector you could test every cell for inclusion with the specified sphere, using the properties of the spatial subdivision (cell deltas, location, and so on) to calculate the cells to test, and to pass these cells to the user-specified functor. This would be useful when selecting all cells within a blast radius. To take it a step further, a new functor could also be derived that would know how to intersect all the elements of a cell, such as *DtGdbNodes*, with the specified primitive. Used together, the new sphere intersector and sphere intersection functor could be used to select all polygons within the blast radius of a detonation.

15.4. Symbol Strings

The *DtSymbolString* class, and related classes, provide a way to treat strings as symbols instead of as ordinary strings. When a symbol string is created from a string, the string is stored inside of a table of strings and the symbol string is given a key with which to get the associated string from the table. As a result, unique strings are stored only once. This can result in significant memory savings if many instances of the same string were formerly stored in the system. Additionally, because each symbol string references its string by a key, string comparison becomes key comparison (much more efficient). In general, the symbol string classes are used in the reader/writer classes and the *DtSpecification* classes. You should not have to update code to use the *DtSymbolStrings*, other than to provide the appropriate functions if any classes were derived from reader/writer classes implementing member functions using *DtSymbolStrings*.

The symbol string classes and header files, which are in *gdb.lib*, are:

- *DtSymbolStore* (*symbolStore.h*) – The repository of unique symbols. Used, as a singleton, by the symbol string class.
- *DtSymbolString* (*symStr.h*) – The core symbol string class.
- *DtCStringHashKey* (*cstringHashKey.h*) – A string hash key based upon symbol strings, instead of full strings.

15.5. VR-Forces Sessions

The session ID is sent inside of a data interaction along with a VR-Forces interface message. When VR-Forces receives a data interaction, it first looks at this session ID. If it does not match the receiver's session ID, the message is discarded immediately. Otherwise, it is passed along for further processing. If a VR-Forces interface message does not have a session ID, the message is automatically processed. (Currently, only one message does not use the session ID – *DtIfVrfObjectData*. These interface messages contain VR-Forces-specific object information that is useful for all VR-Forces components to have.)

The default session ID is 1. A session ID of -1 indicates that all VR-Forces content messages should be processed, regardless of session ID. A user can set the session ID using the `-i` command line option.

You can specify use of a session ID programmatically using the function DtSimInterfaceContent::setUseSessionId(). This flag is set on a per message basis, not message type basis.

15.6. The Entity Editor and OPD Editor Plug-in API

The Entity Editor and OPD Editor plug-in API is a programming interface for customizing the Entity Editor and OPD Editor applications. VR-Forces provides a simulation API that allows developers to add functionality to the simulation engine, such as adding a new kind of sensor, or extending the parameters of an entity. The Entity Editor and OPD Editor Plug-in API complements the simulation API, enabling developers to customize those applications so that end-users can configure their custom extensions to the simulation engine using the Entity Editor and OPD Editor.

You can write a single plug-in that will work for both the Entity Editor and the OPD Editor. An Entity Editor or OPD Editor plug-in provides the following capabilities:

- Entity Editor: Enable new parameters and descriptors to be saved as part of simulation model set. The API does not currently support extending the graphics in the Entity Editor. However, a plug-in is still essential in order to ensure that new customer parameters and descriptors are saved as part of the simulation model set, whenever a user edits and saves their work.
- OPD Editor: Enable new parameters and descriptors to be saved as part of simulation model set and allow end-users to configure new parameters graphically.

! The Entity Editor and OPD Editor API does not support extending the editors with custom graphical elements. However, any built-in *DtReaderWriter* data member types used in extended descriptor or object parameter classes will be viewable and editable by an end-user in the OPD Editor.

15.6.1. Common Uses Cases

The most common situations in which VR-Forces customers decide they need to develop an Entity Editor or OPD Editor plug-in are:

- Configuring new kinds of components: The developer extends an entity with a new capability, for example, a new kind of sensor. This new component requires some parameters for configuration. The developer would like end-users to be able to configure the new sensor parameters using the OPD Editor.

  API examples that add new components include radarWarnRx, stopToShoot, and subtask.

- Extending object parameters: The developer extends an entity with new object parameter data, for example, color. The developer would like end-users to be able to configure the entity with this new parameter using the OPD Editor.

  The addEntity API example shows how to extend an entity with new parameter data.

In both of these use cases, the goal is to allow an end-user to configure the new component descriptors or object parameters using the OPD Editor. However, once end-users have configured their new custom extensions using the OPD Editor, they can save the work and then load it into the Entity Editor to complete other configuration activities (that do not involve the custom extensions). The only thing required to ensure that any custom extensions to the simulation model set are maintained is that the end-user loads the plug-in to the Entity Editor as well as the OPD Editor.

15.6.2. Writing an Entity Editor or OPD Editor Plug-in

To integrate a simulation engine plug-in (or an extended the simulation engine application) with the Entity Editor and OPD Editor:

1. Create a new project for your plug-in. We recommend that you copy an example project (such as stopToShoot) in the *./examples* directory and rename the project.
2. Edit the file *opdEditorPlugin.cxx*, as follows:
   a. Remove references to any component or parameter classes from the original project, and replace them with references to your custom descriptor and object parameter classes.
   b. Include the *editorPluginExtension.h* header file:

      ```
      #include "editorPluginExtension.h"
      ```

      Also include header files containing declarations of any new component descriptor and object parameter files you have created.
   c. Inside the function DtInitializePlugin(), add the calls necessary to include your custom classes in the plug-in. There are calls available to add sensors, controllers, actuators, and object parameter classes (see *editorPluginExtension.h*).

For example, in the stopToShoot example, the plug-in contains a new kind of component descriptor. In the call to addController(), it passes in the name of the component (as it will appear in the OPD entry), the name of the descriptor (as it will appear in the OPD entry), and a creator function that returns a new instance of the descriptor class. The new descriptor is incorporated into the plug-in as follows:

```
#include "editorPluginExtension.h"
#include "stopFireDesc.h"
…

extern "C" {

  DT_DLL_PLUGIN bool DtInitializePlugin(DtEditorExtensionInterface*
   opdIface, DtPluginInformation& info)
  {
    …
    opdIface->addController(MyStopToFireType,
      MyStopToFireDescriptorType,
      MyStopToFireComponentDescriptor::creator);

  return true;
  }
}
```

3. Build the plug-in. Given that you have modeled your new plug-in project on an existing project, the DLLs will be created in the *./bin* directory of your VR-Forces application.
4. (Optional) Create an MTL file to enable automatic loading of your plug-in. The easiest approach is to copy an existing MTL file from an example as a starting point. Edit the file and change the name of the original plug-in to the name of your plug-in. Alternatively, you can manually load the plug-in using the `--loadPlugin` command line argument.

   Copy the MTL file to the *./plugins* directory.

Once you have built your plug-ins and have copied the MTL file to the *./plugins* directory, your new custom descriptors and object parameters will be available for editing in the OPD Editor, and maintained across load and save operations in the Entity Editor.

15.6.3. Examples

The following example projects demonstrate how to write an Entity Editor and OPD Editor plug-in. They work in conjunction with a corresponding simulation API example that extends the VR-Forces simulation engine in some way.

- addEntity
- radarWarnRx
- stopToShoot
- subtask.

15.7. Object Console Messages

Each *DtVrfObject* has a console that is displayed as part of the information dialog box for that object. When you extend the simulation engine, you can send messages to the object console window for a particular object. These messages are also printed to the vrfSim console and to the vrfSim log file, with the name of the object added as a prefix.

To write messages to the object console, use one of the objectConsole*() stream methods on the *DtVrfObject* class or the *DtSimComponent* class. Each object console has five notification levels, and each level has a stream method. You should choose a level appropriate to the nature of the message you are printing to the console. The console will only display messages with a notification level equal to or higher than the notification level chosen by the user. Table 15-1 shows the notification levels and describes the types of messages that you should use for each level.

Table 15-1: Notification message types

Level	Function	Type of Message
Error (1)	objectConsoleError()	Always displayed to the user. Use for important errors that indicate failure of some type.
Warn (2)	objectConsoleWarn()	Displayed to the user by default. Use for situations that may lead to unexpected behavior.
Info (3)	objectConsoleInfo()	Use for important events of an object that are part of normal operation.
Verbose (4)	objectConsoleVerbose()	Use for general events or conditions for an object that are part of normal operation, but less important than Info level events.
Debug (5)	objectConsoleDebug()	Use for debugging level messages that only users attempting to debug operation of an object would be interested in.

When you call an object console method on the *DtVrfObject* class, the message is sent to that object's console. When you call an object console method on the *DtSimComponent* class, the message is sent to the console of the object to which that component is attached. The following example is sent from a class derived from *DtSimComponent*:

```
objectConsoleInfo() << "Firing missile on target" << std::endl;
```

15.8. The Boost Serialization Library

Boost is a large, varied set of free C++ libraries. At this time, VR-Forces only uses the boost::serialization library (as the basis for its Settings Manager). Due to the large size of the Boost libraries, only the serialization libraries are included with VR-Forces. Due to the cross-linking of the libraries, the entire Boost API is available in the *./include/boost* directory. To use libraries other than serialization, those libraries must be downloaded and built separately. Boost is available from http://www.boost.org. The proper version of boost to use is listed in the *VR-Forces Release Notes*.

15.9. Running VR-Forces Applications in Batch Mode

VR-Forces can run scenarios in batch mode. Batch mode allows you to run one or more scenarios multiple times, for a specified amount of time, without providing direct command input from a user interface.

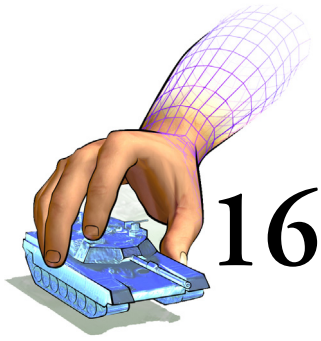
The *DtBatchManager* (*batchMgr.h*) class controls the execution of a batch of scenarios on a single back-end. It takes a pointer to a *DtCgf* class, which it uses to interact with the simulation. You use *DtBatchManager* only when you want to run a batch from the command line on a single back-end. Remote batches do not use this manager; they use the *DtRemoteBatchManager*. (For details, please see "Running VR-Forces Applications in Batch Mode," on page 12-12.)

DtBatchManager has the following basic functions:

- loadBatch() – loads the batch file specified and prepares it for running.
- runBatch() – starts the batch by running the first scenario. All scenarios are run in order, one after another.
- stopBatch() – stops running the batch before it has completed.
- running() – returns true if a batch is currently running.
- done() – returns true if a batch has completed normally.

A *DtBatchManager* contains a *DtScenarioBatchIterator* (*scnBatchIter.h*), which iterates through a batch file and generates the scenarios that are to be run. Information about what percent of the batch has been executed and related statistics can be found in the iterator.

A *DtBatchManager* also contains a *DtRemoteLoggerController* (*remLgrControl.h*), which it uses to send logger control PDUs and interactions to a MÄK Logger running on the exercise if **use-logger-control** is specified in the batch file.



16

Building the VR-Forces Back-End

This chapter explains how to build the VR-Forces back-end. For information about building the front-end, please see *VR-Forces Front-End Developers Guide*.

16.1. Options for Building Applications with VR-Forces

This chapter describes the following alternatives for building applications with the VR-Forces toolkit:

- Rebuild the VR-Forces application. Use this option if you want to use the application (front-end and back-end) supplied by MÄK, but need to add additional features using the VR-Forces toolkit.
- Build an application that uses components of the VR-Forces toolkit. Use this option if you want to use the VR-Forces toolkit as an API as you construct your own application.
- Build a plug-in. Use this option to modify VR-Forces without rebuilding the front-end or back-end.

16.2. Prerequisites for Building VR-Forces

Before you can build an example or a custom application, you must install the software that VR-Forces depends on and you must configure some environment variables, so that our example project files can find these pre-requisites. You must do the following:

- Install the version of VR-Link used to build your version of VR-Forces. A VR-Forces developer's license includes a VR-Link developer's license. (VR-Link is not installed with VR-Forces. You must run a separate installer.)
- Set the environment variable MAK_VRLDIR to the full path to the VR-Link installation (for example, *C:\MAK\vrlink3.13.2*).
- If you are building for HLA, install a supported RTI. We recommend using the MÄK High Performance RTI, but RTIs from other vendors are supported as well, as long as they comply with the appropriate versions of the HLA Standards. A VR-Forces license does not include a license for the MÄK RTI. However, the MAK RTI allows you to run federations of up to two federates without requiring a license. You can download the MAK RTI at: http://www.mak.com/products/rti.php.

  If you install an RTI, set the environment variable MAK_RTIDIR to the full path to the RTI installation (for example, *C:\MAK\makRti4.0*).

The VR-Forces example project files look for RTI header files in a directory called "include" directly under $(MAK_RTIDIR), and look for RTI link libraries in a directory called "lib" directly under $(MAK_RTIDIR). If you are using an RTI that does not follow that directory-naming convention, you may need to change the project settings in order for the build system to find your RTI headers and libraries.

16.3. Rebuilding the VR-Forces Back-End

This section describes how to use the sample project files supplied with VR-Forces to rebuild the VR-Forces back-end.

When you build a VR-Forces application, you must build it against the version of VR-Link that your version of VR-Forces was built against. Please see *VR-Forces Release Notes* for your version of VR-Forces, to find out which version of VR-Link to use.

16.3.1. Building the VR-Forces Back-End for Windows

Please review the prerequisites for building VR-Forces in "Prerequisites for Building VR-Forces," on page 16-2.

The sample project files create the release and debug executables *vrfSimUser.exe* and *vrfSimUserDebug.exe*. This naming convention ensures that you do not inadvertently overwrite the executables supplied by MÄK. The files are placed in the *.\bin* directory.

The *.\appsrc\vrfSim* directory contains MS VC++ project files. The project file contains release and debug configurations. The projects are:

- *vrfSimUserDIS* – projects that build the DIS version of the VR-Forces back-end
- *vrfSimUserHLA13* and *vrfSimUserHLA1516* – projects that build the HLA versions of the VR-Forces back-end.

The sample projects enable you to add additional features using the VR-Forces toolkit. You can extend the existing VR-Forces simulation back-end and the GUI. You can also use the sample project as a starting point for building your own application that uses components of the VR-Forces toolkit.

In order to run HLA versions, you need to have your computer configured to find the proper RTI DLLs. Refer to Section 2.5, "Installing an RTI", in *VR-Forces Users Guide*.

16.4. Building an Application that Uses the VR-Forces Toolkit

Please review the prerequisites for building VR-Forces in "Prerequisites for Building VR-Forces," on page 16-2.

To build an application that uses the VR-Forces toolkit, you must let your compiler know where to find VR-Forces header files and you must let your linker know where to find VR-Forces libraries. Because VR-Forces relies on VR-Link, you must also indicate where to find VR-Link header files and libraries. In addition, when building for HLA, you must provide paths to RTI header files and libraries. Finally, there are several preprocessor definitions that must be included depending on your build configuration.

When you build VR-Forces, you can ignore errors about undefined environment variables.

16.4.1. Building an Application for Windows

Please review the prerequisites for building VR-Forces in "Prerequisites for Building VR-Forces," on page 16-2.

Under Windows, you must include paths to the VR-Forces, VR-Link, and RTI header files and libraries within your project settings. Rather than using absolute paths, we strongly recommend that you use environment variables to indicate the locations of the various components. That way, when you upgrade to new versions of VR-Link, the RTI, and so on, you will not need to change all of your project settings. In the examples we assume that MAK_VRLDIR, and MAK_RTIDIR point to your VR-Forces, VR-Link, and MÄK RTI trees respectively.

For assistance, please contact technical support.

16.5. Deploying VR-Forces Applications

If you create applications using the VR-Forces APIs, you may need to deploy them to internal or external customers. To deploy a VR-Forces application:

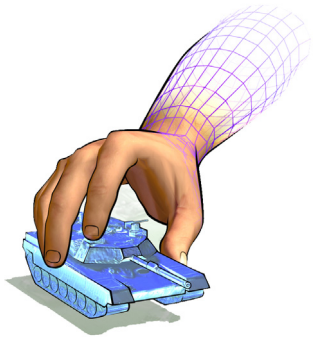
- The deployment site must run the license manager and each application must have a VR-Forces run-time license.
- On Windows, you do not have to set any Registry values.
- You do not have to set any environment variables, unless the application will be used for development. In a development environment an application would have the same requirements as described in this chapter.
- You must include all of the DLLs from the *./bin* directory for the protocol in which you are running the applications (DIS, HLA 1.3, or HLA 1516.)
- Any configuration files that you have modified from their default configurations to support your application are required, except that if you are not using the OPD Editor, you do not have to ship *dis-entity-types.csv*. Configuration files are located in *./config*, *./appData/settings/vrfGui*, *./appData/settings/vrfSim*, and in the various directories under *./data/simulationModelSets*.
- You do not have to include files from the *./data* subdirectories that you will not use, for example, any terrain databases that you do not need, example scenarios, and so on. If you customize the object parameter database, you must determine which of the files supplied by MÄK are required.

16.6. Creating and Initializing the Simulation Engine

The code in *./appsrc/vrfSim/main.cxx* instantiates a *DtVrfApp* class, initializes it, and executes its mainLoop() function. The examples in the *./examples* directory that create executables (as opposed to plug-ins) each have a similar version of *main.cxx*, that has been customized by overriding the default creator provided by VR-Forces, and configuring the *DtVrfApp* with the modified creator. Please see "Overriding DtVrfCreator," on page 2-13 for more information.

16.7. Building Examples

To build and debug example applications, follow the procedures for building VR-Forces applications in the online class documentation. Follow the Example Code link in the Additional API Documentation section. If you are having problems running examples, remember to set the working directory to the location of the example before you debug it.



Index

Most classes, header files, and functions are indexed in *Index of Classes, Functions, and Header Files*.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

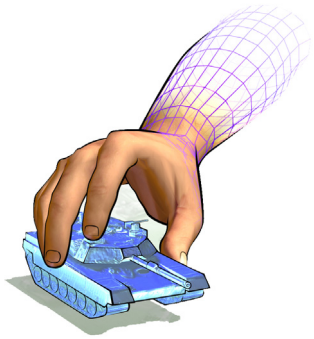
T

U

V

W

X-Y-Z



Index of Classes, Functions, and Header Files

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Link - Simulate - Visualize

68 MOULTON STREET CAMBRIDGE, MA 02138 617.876.8085 www.mak.com