



VR-Link

The Simulation Networking Toolkit

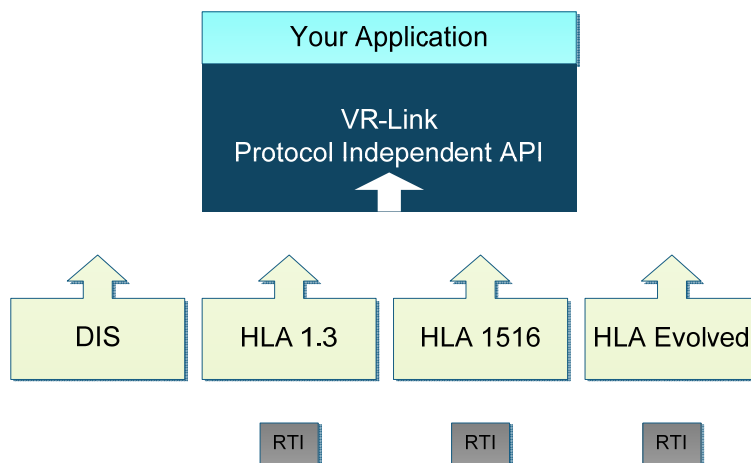
Are you building a new simulation application that must be HLA or DIS-compliant? Do you need to take an existing standalone simulation and make it work in a multi-player environment? Do you have a legacy DIS application that you need to convert to HLA? Have you struggled to implement first DIS, then HLA, but become frustrated with all of the time and money you are spending just keeping up with new RTI releases, new FOMs, and changes to the HLA interface specification? Are you concerned about the cost of moving to the IEEE 1516 version of HLA, or to HLA Evolved?

If you answered yes to any of these questions, then VR-Link is for you!

What is VR-Link?

VR-Link is an object-oriented C++ toolkit that simulation developers use to quickly and easily build HLA, and DIS compliant applications. It provides a set of libraries that implement a stable, consistent, and documented API (Application Programmer's Interface) above the various protocols – an SDK (Software Developer's Kit) for interoperability. When you link your application you simply pick a protocol specific library to use. To use a different protocol, just relink your application. When you use VR-Link, your developers are free to concentrate on the specific goals of your project, rather than spending their time worrying about the details of networking protocols. A few calls to VR-Link's functions replace thousands of lines of code that you would otherwise need to write yourself, saving time and money on development and maintenance.

VR-Link's top-level API abstracts away from the networking protocols. For example, application code sets the current state of locally simulated entities and objects, and any needed information is automatically sent to other applications either through the HLA's RTI or DIS network. On the incoming side, VR-Link processes information from other applications, and provides access to the current state of remote objects, without the application even having to know when individual updates arrive.



If you are using *either* HLA, or DIS, there is great benefit in using VR-Link, since it saves you the time it would take to write all of the code that is necessary to interoperate using your protocol of choice. But if you need to be able to speak *more* than one protocol, (e.g. HLA *and* DIS), or might have to in the future, then VR-Link is *doubly* valuable. That is because VR-Link implements all three protocols largely through

the same API. This means that you can write code once, and switch between DIS, HLA 1.3, IEEE 1516, and HLA Evolved just by recompiling and linking with the appropriate versions of the VR-Link libraries.

Is VR-Link a Protocol Translator?

No. VR-Link's libraries are linked directly into your application, making it natively compliant with whichever protocol you have built for (depending on which version of the libraries you choose). VR-Link does not, for example, "go through DIS" when you are participating in an HLA exercise. In contrast to some middleware toolkits that always work in terms of DIS PDUs internally, VR-Link decodes incoming HLA updates directly into the state repository object that you use to inspect the data. Similarly, we generate outgoing HLA updates directly from the state repository that you use to set the data.

The VR-Link toolkit does not translate between protocols, so if you only want to translate between DIS and HLA, you will need to use a product like VR-Exchange to do this. VR-Exchange is a separate application that listens for DIS and HLA traffic on a network, translates from one protocol to the other, and sends the data back out. Typically, a translator such as VR-Exchange is not necessary when all applications are built using VR-Link, since VR-Link applications can play directly in both HLA and DIS exercises without translation. But if you have a legacy DIS application that does not use VR-Link, and it is not feasible to convert it to HLA (or perhaps even to modify it at all), VR-Exchange will allow that application to play in an HLA exercise. Please contact us at info@mak.com for more information about VR-Exchange.

Does VR-Link dictate or constrain my application design?

VR-Link is a toolkit. It supplies you with the classes, functions, and utilities you need to write a simulation application, but unlike a framework, it doesn't dictate an application structure that you are required to use. You, not VR-Link, have control of your application. For example, you will probably want to set the state of a locally simulated entity at periodic intervals, but VR-Link doesn't care whether you do that using a loop, by using a timer, or some other mechanism.

What does the API look like?

This question is probably best answered by walking through two simple VR-Link examples. The first is a listen-only application that observes an exercise without simulating any entities on the network. The second is a send-only program that does not process information about remote entities. The source for both examples is included with VR-Link

A listen-only example:

This example illustrates a simple, listen-only VR-Link application. This application can be compiled for DIS or HLA, and contains no protocol-specific code.

With each iteration of the loop, the program prints an entity's updated, dead-reckoned position in topographic coordinates. In addition, if a fire PDU or interaction is detected on the network, the program prints a message showing the entity ID of the attacker.

```
1 // Define a callback to process fire interactions
2 void fireCb(DtFireInteraction* fire, void* usr)
3 {
4     std::cout << "Fire Interaction from " << fire->attackerId().string() << std::endl;
5 }
6
7 int main(int argc, char** argv)
8 {
9     // Create a connection to the exercise or federation execution
10    // based on initialization arguments passed as command-line arguments
11    // This causes the application to connect to the DIS network,
12    // HLA execution.
13
14    DtVrlApplicationInitializer appInit(argc, argv, "VR-Link Listen");
15
16    DtExerciseConn exConn(appInit);
17
```

```

18 // Register a callback to handle fire interactions
19 DtFireInteraction::addCallback(&exConn, fireCb, NULL);
20
21 // Create an object to manage remote entities.
22 DtReflectedEntityList rel(&exConn);
23
24 while (1)
25 {
26     // Tell VR-Link the current value of simulation time
27     exConn.clock()->setSimTime(exConn.clock()->elapsedRealTime());
28     // Process any incoming messages
29     exConn.drainInput();
30
31     // Find the first entity in the reflected entity list
32     DtReflectedEntity *first = rel.first();
33
34     if (first)
35     {
36         // Grab its state repository, where we can inspect its data
37         DtEntityStateRepository *esr = first->entityStateRep();
38
39         // Print the position
40         std::cout << "Position: " << esr->location().string() << std::endl;
41     }
42
43     // Sleep till next iteration
44     DtSleep(0.1);
45 }
46 }

```

Connecting to an Exercise

In lines 14-16, the program creates a *DtExerciseConnection*. This connection serves as the program's interface to an exercise. Depending on whether the application is compiled for DIS, HLA 1.3, HLA 1516, or HLA Evolved, the *DtExerciseConn* constructor will ask the *DtVrlApplicationInitializer* for the appropriate initialization parameters, such as DIS port number or the HLA federation execution name. If you do not want to use VR-Link's command-line and configuration file-based initializer class, you can call alternate, protocol-specific constructors to create a *DtExerciseConn*.

Managing State and Interaction Information

Applications based on VR-Link typically use callbacks to handle incoming interactions such as fire, detonations, and collisions. For example, a callback named *fireCb* is registered with the *DtFireInteraction* class at line 19. This callback (defined at line 2) prints a message containing the attacker ID. It executes whenever the exercise connection receives a Fire interaction or PDU during a call to *drainInput()*.

Tracking Entities

We create a reflected entity list in line 22 to keep track of entities found on the network. The entity list tracks the arrival and departure of entities, performs dead reckoning, manages time outs, and performs other entity-tracking tasks.

Listening to the Network

At the start of each iteration, the program sets VR-Link simulation time (line 27) to provide a common time value for use by time-related operations that occur within an iteration of the loop (such as the dead-reckoning of multiple entities). The *drainInput()* call (line 29) reads and processes any messages arriving through the exercise connection. This call triggers the execution, if needed, of any callbacks you have registered for that exercise connection. In HLA, this is where the RTI gets ticked. In line 32, the program finds the first entity in the entity list, then in line 37, retrieves the pointer to the entity's entity state repository, where VR-Link stores the values describing the remote entity's state. Line 40 obtains and prints the dead-reckoned entity location.

A send-only example:

This example illustrates a simple send-only application that simulates the flight of an F18 aircraft. The program begins by sending an HLA fire interaction or a DIS fire PDU. Thereafter, the F18 flies north for 10 seconds, updating its position by sending HLA attribute updates or DIS entity state PDUs.

```
1 int main(int argc, char** argv)
```

```

2 {
3     // Create a connection to the exercise or federation execution
4     // based on initialization arguments passed as command-line arguments
5     // This causes the application to connect to the DIS network or
6     // HLA execution.
7
8     DtVrlApplicationInitializer appInit(argc, argv, "VR-Link Listen");
9
10    DtExerciseConn exConn(appInit);
11
12    DtEntityType f18Type(DtPlatform, DtPlatformDomainAir,
13        DtUnitedStates, DtFighter, DtF18, 0, 0);
14
15    // Create an entity publisher for the entity we are simulating
16    DtEntityPublisher f18EntityPub(f18Type, &exConn, DtDrDrmRvw, DtForceFriendly);
17
18    // Hold on to its state repository, where we can set data
19    DtEntityStateRepository *esr = f18EntityPub.entityStateRep();
20
21    // Create a topographic view on the state repository, so we
22    // can set position information in topographic coordinates
23    double refLatitude = DtDeg2Rad( 35.699760);
24    double refLongitude = DtDeg2Rad(-121.326577);
25    DtTopoView f18TopoView(esr, refLatitude, refLongitude);
26
27    // We can use the ESR to set state
28    esr->setMarkingText("VR-Link");
29
30    DtVector position(0, 0, -100);
31    DtVector velocity(20, 0, 0);
32
33    // Send a Fire Interaction
34    DtFireInteraction fire;
35    fire.setAttackerId(f18EntityPub.globalId());
36    exConn.sendStamped(fire);
37
38    // Main loop
39    DtTime timestep = 0.05;
40    DtTime simTime = 0;
41    while (simTime <= 10.0)
42    {
43        // Tell VR-Link the current value of simulation time
44        exConn.clock()->setSimTime(simTime);
45
46        // Process any incoming messages
47        exConn.drainInput();
48
49        // Set the current position information
50        f18TopoView.setLocation(position);
51        f18TopoView.setVelocity(velocity);
52
53        // Call tick, which insures that any data that needs to be
54        // updated is sent.
55        f18EntityPub.tick();
56
57        // Set up for next iteration
58        position[0] += velocity[0] * timestep;
59        simTime += timestep;
60
61        // Wait till real time equals simulation time of next step
62        DtSleep(simTime - exConn.clock()->elapsedRealTime());
63    }
64 }

```

Connecting to an Exercise

Like the listen-only example, this program creates a *DtExerciseConnection* to provide an interface to the RTI or DIS network (lines 8 through 10).

Managing Entities

Line 12 defines the entity type the F18 will use. To be visible to other applications in the exercise, each locally-simulated entity requires a *DtEntityPublisher*, created in line 16. The entity publisher manages the

generation of messages for this particular entity. It provides an entity state repository where you can set state values, and a tick() function, which causes state information to be sent to the network if necessary. Line 19 sets up a pointer to the entity state repository, then line 25 creates a topographic view on that repository. This lets us set the entity's positional data using topographic coordinates, rather than the default geocentric coordinates. (Lines 23 and 24 hard code the coordinates for this example.) Line 28 shows an example of storing a non-positional state value in the repository.

Sending interactions

An example of sending an interaction appears in lines 34-36. You can send the interaction using the exercise connection's sendStamped() function. You can use sendStamped() to send state updates as well, but it is preferable to let the entity publisher send state updates for you, using data in the entity state repository.

Sending State Messages

The main loop executes twenty times per second for ten seconds. As in the listen-only example, this program sets simulation time at the start of each iteration (line 44). While this program's main purpose is to demonstrate the sending of data to the network, it is not a true send-only application. Incoming data is also processed with the drainInput() call on line 47. This call is required for HLA, because this is where we tick the RTI. The program updates the F18's positional data in its entity state repository in lines 50 and 51, and ticks the entity publisher in line 55 to send the updated data onto the network. Thereafter, the only remaining tasks are to increment the F18's position, increment the simulation time, and sleep until it's time to begin the next iteration.

Is VR-Link just a wrapper around the RTI?

This is a common question: If the RTI already provides an API for communicating through HLA, why do I need *another* API like VR-Link? Clearly, there would be little benefit to using VR-Link if it just meant substituting VR-Link calls for RTI calls (and paying for the privilege!) Obviously, then, VR-Link is not just a wrapper around the RTI. Asking this question is similar to asking "Is MFC just a wrapper around the Win32 API?" "Is a scene-graph API just a wrapper around OpenGL?" or "Is the DIS version of VR-Link just a wrapper around sockets?" In all of these cases, the higher-level toolkits provide a lot of additional power compared to the corresponding low-level API. Anything you can do with the higher-level APIs, you can do with the lower-level APIs, but it would take you much longer, and it would cost you much more.

The RTI provides a general, low-level API for HLA communication. It treats all attributes as opaque data, providing neither type-safety nor data marshalling. It requires all federates to do a large amount of bookkeeping, such as routing all updates for a particular object to the place where you're storing the object's current state. And the RTI has no specific support for any particular simulation domain. VR-Link's top-level API provides a type-safe, FOM-independent interface to the data typically exchanged by real-time platform-level simulations. It handles data marshalling, manages conversion to the representation dictated by your FOM, and does most of the bookkeeping that the RTI requires of federates. In short, you end up writing a *lot* less code using VR-Link than you would using the RTI directly.

Here is one (we think) impressive example:

Suppose you wanted to write a small test federate that simply sends an empty fire interaction to the exercise each time it is run. Using the RTI API alone, you would need to create the federation execution if it doesn't already exist, join the federation execution, tick the RTI to give it time to complete the joining handshaking, ask the RTI for the InteractionClassHandle of the class you want to use, publish the interaction class by its handle, create an RTI::ParameterHandleValuePairSet (the RTI representation of an interaction), write a subclass of RTI::FederateAmbassador, write a callback for the startInteractionGeneration service so that the RTI can tell you whether any remote federates are interested in the interaction class you are publishing, query the RTI for the parameter handle of each parameter of the interaction class, add each parameter value to the RTI::ParameterHandleValuePairSet (even if the value is just zero), send the interaction using the sendInteraction service, delete the RTI::ParameterHandleValuePairSet, resign from the federation execution, and destroy the federation execution if there are no other federates remaining besides you. In addition, you would probably want to

catch the various exceptions that the RTI generates so that you can print useful diagnostics or exit gracefully when something goes wrong. You would probably end up with several hundred lines of code.

The same federate, which implements every task described above, can be written using VR-Link as follows:

```
int main()
{
    DtExerciseConn conn("VR-Link", "MyFederate");
    conn.sendStamped(DtFireInteraction());
}
```

Below, we describe two more examples of how VR-Link minimizes the amount of code you need to write, as compared to using the RTI's API directly:

Keeping track of remote objects

Most applications need to maintain a database containing current information about the state of all remotely simulated entities (or other types of objects). For example, if you have a 3D-graphics-based application, you might want to query this entity database each frame to determine where to draw each entity. The RTI does not maintain this database for you. In fact, it doesn't "remember" the state of any objects at all. The RTI just passes update messages among federates.

Using the RTI API alone, you would need to explicitly subscribe to each object class you are interested in, then subclass `RTI::FederateAmbassador` and implement callbacks (by overriding virtual functions) for the `discoverObjectInstance`, `removeObjectInstance`, and `reflectAttributeValues` services. You would need to implement an object list, and add objects to it as a result of `discoverObjectInstance` calls, and remove objects from it as a result of `removeObjectInstance` calls. In addition, you'd probably want to call `RTI::RTIambassador`'s `requestObjectAttributeUpdate` after you discover an object, to make sure you get updates for all needed attributes. (If you don't do this, and attribute values haven't changed since the last time they were sent, they will not be sent to you). When you receive a `reflectAttributeValues` call, you would need to find the appropriate object in your list, decode the update message, and place the updated values into the structure you are using to store the current state.

(Decoding an update message means converting attribute values from opaque streams of bytes to more useful structures or C++ classes. For example, you may need to deal with byte-alignment issues, little-endian/big-endian conversions, doing byte arithmetic to iterate through variable length fields in complex attributes, etc.)

Using VR-Link, you just create an instance of the `DtReflectedEntityList` class. It subscribes to object classes, requests attribute updates, and does all the bookkeeping above. It automatically creates and deletes instances of `DtReflectedEntity` based on discover and remove service calls. It automatically receives and decodes update messages into the appropriate `DtReflectedEntity`'s `DtEntityStateRepository`. It also performs dead-reckoning and (optionally) smoothing. It means that you never even have to worry about when individual attribute update messages are received from the RTI. You can just ask the `DtReflectedEntityList` for the current dead-reckoned position of a particular entity (identified by handle or by name), and it will give it to you. The complete VR-Link-based application below prints the entire current state of the first discovered entity each frame, including dead-reckoned position and orientation. Of course, it also handles the mechanics of properly creating (if necessary), joining, resigning and destroying (if necessary) the federation execution as well.

```
int main()
{
    DtExerciseConn exConn("VR-Link", "MyFederate");
    DtReflectedEntityList rel(&exConn);

    while(1)
    {
        exConn.clock()->setSimTime(exConn.clock()->absRealTime());
        exConn.drainInput();
        if (rel.first())
        {
            rel.first()->entityStateRep()->printData();
        }
    }
}
```

```

    }
}
}

```

Simulating Entities Locally

On the sending side, VR-Link will save you just as much bookkeeping as on the receiving side, perhaps more.

Using the RTI's API alone, you would need to query the RTI for the object class handle you want to use, and for the attribute handles of any attributes you want to publish. You would then need to publish the object class. You would subclass RTI::FederateAmbassador, and write callbacks for start/StopRegistrationForObjectClass and turnOn/OffUpdatesForObjectInstance services in order to be notified about whether any remote federates are interested in the classes and attributes you are publishing. You would register your object instance. You would need to maintain a data structure containing the current state of the object you are simulating, and a similar data structure containing the attribute values you have last sent through the RTI. Each frame, you would need to determine whether any values have changed (perhaps by enough to exceed a threshold) since the last time it was sent. Once you've determined which attributes need to be sent this frame, you would need to create an RTI::AttributeHandleValuePairSet, encode values into it for the relevant attributes (all the same issues apply as in decoding), and send the update using updateAttributeValues. In addition, you'd need to write a federateAmbassador callback for provideAttributeUpdate. You need to maintain a list for each object you are simulating of the attributes that have been requested since the last time they were sent. You need to add attributes to this list when you get a provideAttributeUpdate callback, and remove attributes from the list when you send updates.

Using VR-Link, you just create an instance of the DtEntityPublisher class, set current values for attributes each frame, and call tick(). DtEntityPublisher publishes the class, keeps a record of the values that have been sent through the RTI, performs the comparisons to determine which attributes need to be sent because they have changed, keeps track of which attributes are actually needed by remote federates, processes provideAttributeUpdate callbacks and insures that attributes that have been requested in this way are sent even if their values have not changed, encodes updates when necessary and sends them. Application code looks like this:

```

int main()
{
    DtExerciseConn exConn("VR-Link", "MyFederate");
    DtEntityPublisher pub(DtEntityType("1:1:225:1:1:0:0"), &exConn);

    while (1)
    {
        exConn.drainInput();
        exConn.clock()->setSimTime(exConn->clock()->absRealTime());
        // Set current attribute values using typesafe interface
        pub.esr()->setLocation(DtVector(10.0, 20.0, 30.0));
        ...
        pub.tick();
    }
}

```

What if I want to directly access the RTI, or tune protocol-specific parameters?

No problem. Our goal is to insulate you from the networking details you don't want to see, not to restrict you from getting to the ones you do.

For example, in HLA, VR-Link typically sets up your RTI connection for you. But we provide access to the RTI::RTIambassador object (just call DtExerciseConn's rtiAmb() function), so that you can make some RTI calls directly. While VR-Link satisfies the RTI requirement of providing a subclass of RTI::FederateAmbassador, you can further derive from our implementation, overriding the way we handle RTI callbacks, or adding your own functions to handle RTI services that we ignore. Just tell VR-Link about your subclass using DtExerciseConn's setFedAmbCreator() function.

Below the top-level protocol-independent API are a wealth of just-as-well-documented protocol-specific classes and interfaces. For example, DtHlaObject gives you access to all of the details of the HLA object

that your DtEntityPublisher or DtReflectedEntity is managing. You can view or change the set of attributes you are publishing, make ownership-management calls, force updates of particular attributes, register for callbacks each time an update is received, and much more.

On the DIS side, you can set or inspect particular fields of each PDU, configure networking parameters such as multicast addresses or UDP ports, set heartbeat and timeout periods, etc.

Doesn't this extra layer of software hurt performance?

Performance is always a great concern of ours, because after all, real-time simulation is our focus. The thing to remember is that the functionality that we have implemented in VR-Link is not extra code. It's code that you would have had to write yourself if you were not using VR-Link; we've just written it for you. You would have needed to receive updates and decode them into a state repository. You would have had to package up HLA updates and pass them to the RTI. Just about the only avoidable overhead associated with using VR-Link are additional function calls that you could have avoided by writing large monolithic functions, rather than breaking things up the way we have. However, since most well-written applications tend to break tasks down into functions anyway, the typical call-stack depth in VR-Link-based applications probably does not differ much from non-VR-Link-based applications in practice. In tests that we have done, the additional processing that VR-Link adds compared to directly making RTI calls or implementing DIS and making direct socket calls is negligible.

In addition, there are many features in VR-Link that directly help you to build efficient applications. Some examples: VR-Link respects HLA advisories, so it will not send updates for attributes that no one is subscribed to. Reflected object lists are based on hash lists, so that lookups are efficient. Decoding of incoming HLA updates is table driven, so that we don't have to "switch" on attribute handles. VR-Link performs dead-reckoning for only those entities whose locations you actually ask for. Further, we remember that we've already done dead-reckoning for a particular frame, so that if you ask for the dead-reckoned location for an entity three times in a single frame, the dead-reckoning code is only executed the first time. Excess copies are avoided wherever possible. For example, when you set a parameter value in a DIS PDU, the data is immediately stored in the buffer that will eventually be used to send the complete PDU.

Does VR-Link work with new or custom FOMs?

Absolutely! VR-Link was designed with FOM agility in mind. VR-Link comes with built-in support for several different versions of the RPR FOM, but it can be extended or configured in many different ways to work with changes or extensions to the FOM, or to work with entirely new FOMs.

There are two broad categories of custom FOMs that you might need to work with:

1. FOMs that contain concepts that are already present in VR-Link's top-level API, but which may represent the concepts differently from the way they are represented in the VR-Link API.
2. FOMs that include concepts that are not present in VR-Link's API.

VR-Link includes powerful tools to help you in both situations. For FOMs that contain concepts that are covered by the existing VR-Link API, VR-Link's FOM Mapper architecture allows you to "map" from the VR-Link API to the data representations dictated by your FOM. When FOM Mapping is possible, application code that is written to the top-level VR-Link API does not need to change in order to switch from FOM to FOM. In other words, our FOM Mapper architecture allows you to create federates that are FOM-Agile.

For FOMs that contains new concepts, or concepts not already covered by VR-Link, the VR-Link Code Generator can be used to automatically extend the VR-Link API and libraries to support your FOM's new concepts. Your application can call the extended VR-Link API, just as it calls the built-in portions of the VR-Link API. The two techniques can also be used together for FOMs that have *some* overlap with the VR-Link API, but that also have some new concepts. So you don't lose the benefits of FOM Mapping just because you must also extend the VR-Link API.

How does FOM Mapping work?

VR-Link's top-level API is FOM-independent, in that it allows you to set and inspect the state of local and remote objects, and the data contained in locally generated or received interactions, without regard to how the data is actually represented in a FOM. This top-level API includes the object publishers, reflected objects, reflected object lists, state repositories and interaction classes. The implementations of these classes consult a VR-Link object called a FOM Mapper for help in encoding and decoding data as dictated by your current exercise's FOM. A FOM Mapper for a particular FOM is built by writing encoding and decoding functions, and registering them with a DtFomMapper object.

When we say that VR-Link comes with built-in support for the RPR FOM, what we mean is that we have implemented a FOM Mapper for the RPR FOM, and that that FOM Mapper is distributed with VR-Link. This means that you have out-of-the-box interoperability with RPR-FOM-based federations without having to implement a FOM Mapper yourself. However, the techniques that we used for implementing the RPR FOM Mapper are no different than the ones a customer would use to create a FOM Mapper for their new FOM. In other words, user-defined FOM Mappers are treated no differently than the built-in RPR FOM Mapper.

When working with FOMs that have the same concepts as the VR-Link API, you can switch from FOM to FOM just by modifying the FOM Mapper, which you can do without affecting the top-level API. This means that application code written to the top-level API does not need to change when switching from FOM to FOM. When a FOM Mapper is implemented in a DLL, applications do not even need to be recompiled in order to switch to a new FOM.

For example, in one FOM (such as the RPR FOM), position data may be represented in geocentric coordinates in an attribute named "Position". In another, it might be represented in topographic coordinates in an attribute named "Location". In either case, position data is available to application code through VR-Link in the manner indicated by its FOM-independent API - as geocentric data obtainable using *DtEntity-StateRepository*'s `location()` member function. FOM Mapping code takes care of converting from the various FOMs' representations to the representation chosen by VR-Link.

Can I really tailor my application to a new FOM without recompiling?

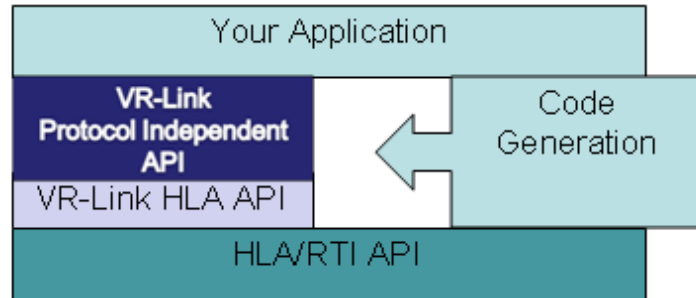
Yes. Because FOM-Mapping code can be dynamically loaded by VR-Link as a plug-in, applications do not even need to be recompiled when switching among such FOMs, as long as the target FOM includes the same concepts as VR-Link's API. When you build a FOM Mapper as a DLL, you can even use it with applications for which you don't have source code. For example, you can tell the MÄK Stealth, VR-Forces, or MÄK Plan View Display to load your custom FOM Mappers, if you want to use those tools in federations that do not use the default RPR FOM.

What about a FOM that includes new concepts?

If you are using FOMs that contain concepts not found in the VR-Link API, you must extend the VR-Link API with new classes to represent your FOM's new objects and interactions. Specifically, you need to create new classes derived from our base state repository, publisher, reflected object, and reflected object list classes to handle your new FOM elements. Although you can do this by hand-coding or cut and paste, we strongly recommend using the VR-Link Code Generator to do the work of extending VR-Link automatically.

What is the VR-Link Code Generator?

Whether your FOM just adds a few classes to the RPR FOM, or represents entirely different simulation concepts, the VR-Link Code Generator can help. The VR-Link Code Generator reads any HLA FOM, and automatically generates VR-Link extensions for that FOM. The tool produces fully-formed C++ source and header files, along with the Microsoft Visual C++ solution files and the UNIX® Makefiles required to compile them into a VR-Link extension library. The generated classes are ready to use for publishing and tracking new classes of HLA objects, and for sending and receiving custom interactions. The generated code will have the same "look and feel" as built-in VR-Link classes.



The VR-Link Code Generator works with both HLA-1.3-style OMT files and IEEE-1516-style XML-based FOMs.

What if I am just extending the RPR FOM?

This is a very common case among VR-Link users. Although you could use the VR-Link Code Generator to generate state repositories, publishers, etc. for your *entire* RPR-based FOM, it doesn't really make sense to do that, since VR-Link already has built-in support for the underlying RPR FOM. Instead, you can instruct the VR-Link Code Generator to generate VR-Link extensions just for the classes that are not covered by VR-Link already. The Code Generator's GUI allows you to select which classes to generate code for.

For cases where your FOM *changes* the name or data representation of some *existing* RPR FOM classes or attributes, you don't need to generate API extensions at all. Just start with the built-in RPR FOM Mapper, and register a few new mapping functions to handle the changes.

Does the VR-Link Code Generator help me build FOM Mappers?

Unfortunately, no. The process of generating code to exactly match a FOM specification is a fairly mechanical one that lends itself easily to code generation. And that is the problem that the VR-Link Code Generator solves. The process of mapping from a fixed FOM-independent API to a specific FOM requires a semantic understanding of the FOM, the VR-Link API, and how they relate. It's easy for the Code Generator to look at a FOM attribute called "MyLoc", and generate functions called `getMyLoc()` and `setMyLoc()`. But it would be quite difficult for an automated tool to "know" that the attribute was meant to represent the same concept that VR-Link refers to as "Position". And it would be impossible for an automated tool to understand the free-text comments and documentation associated with a FOM, that gives a human programmer the information he needs to implement mappings between the VR-Link API and that FOM.

In the past, VR-Link included a simple tool called the FOM Mapper Builder that helped you get started in writing a FOM Mapper, by handling some of the mechanics of setting up your header and source files. But the actually mapping and translation functions still had to be hand-coded. Because of the need to heavily edit the files created by the FOM Mapper Builder, most customers found it easier to use our FOM Mapper source code examples as a starting point, than to use the output of the FOM Mapper Builder. As a result, although the FOM Mapping concept itself is as central a part of VR-Link as ever, the FOM Mapper Builder tool is no longer supported.

How extensible is VR-Link?

VR-Link's C++ API and implementation allow you to override almost all of its default functionality through subclassing. You can extend the toolkit to work with new experimental DIS PDUs, or with new FOM interaction or object classes. The FOM Agile architecture that we described above is just one of the ways that VR-Link supports extensibility. User-defined kinds of PDUs, interactions, objects, sockets, state repositories, etc. have the same first-class-citizen status within VR-Link as our built-in types.

VR-Link comes with several source-code examples demonstrating the procedure for creating subclasses of DtPdu, DtInteraction, DtObjectPublisher, DtReflectedObject and DtStateRepository, and for telling VR-Link about your new subclasses.

How stable is the API?

Maintaining backward compatibility with older releases is a key design philosophy at MAK. Our goal each release is that customers should not have to change any application code in order to upgrade to the latest version of VR-Link, unless they specifically want to take advantage of new features. Sometimes, this is not quite possible, but in general, we are quite successful in meeting this goal. The only major API redesign we have done was in 1997, when we converted VR-Link from a DIS toolkit, to a package that abstracts away from protocol details, allowing us to support both DIS and HLA with largely the same API.

All of the other products that MAK has developed, as well as most of our contracts and R&D projects rely on VR-Link. This alone gives us a large incentive to keep VR-Link's API stable from release to release.

Do I get source code with VR-Link?

While we provide source to lots of examples, and several utilities, we do not provide source for the VR-Link libraries. However, because of the extensibility we described above, this actually rarely matters to users. Ask yourself this: Is it the source code per se that is important, or is it just that you want to know you have the power to modify the behavior of the toolkit, extend it with your own functionality, and work around the (rare) bugs that you find. Most people have told us that it is the latter.

For many toolkits, you need source code in order to have this kind of control, however, we have gone to great lengths to insure that you will have the power to modify, extend, and fix *without* VR-Link source code. Almost everything in VR-Link can be overridden: Don't like the way we create our sockets? Create a subclass of DtSocket that makes the calls you want. Want to add a print statement every time an update is sent? Override DtExerciseConn::send(). Don't want to wait till next release for a fix to a problem with encoding a particular attribute? Write a new encoding function, and register it with the FOM Mapper. Besides, when someone does report a bug, our engineers will be quick to provide a patch or workaround, meaning you will not just have to wait for the next release to have your problem addressed.

You only support the MAK RTI, right?

Certainly not. While the MAK RTI is probably the most popular RTI used by our VR-Link customers, we know that many are required to use other RTI implementations. VR-Link has been used successfully with DMSO RTI NG, MATREX RTI, JNTC RTI, RTI NG Pro, Gertico, RTI-s, pRTI 1.3, and others, in addition to the MAK RTI. In fact, VR-Link should be able to work with any RTI that is built against the standard RTI header files associated with the HLA 1.3 Specification, or against the SISO Standard Dynamic Link Compatible C++ API for the IEEE 1516 Interface Specification.

Does VR-Link support the IEEE 1516 version of HLA?

Yes. In fact, most VR-Link customers can seamlessly transition their federates from HLA 1.3 to IEEE 1516 - without changing any code! The same top-level protocol-independent API that has always supported both DIS and HLA 1.3 supports the IEEE 1516 version of HLA as well. Customers write federate code once, and choose a protocol version at compile time.

If you use the MAK RTI, you will probably even be able to mix and match HLA 1.3 and IEEE 1516 federates in the same federation. (For most services, we were able to use the same on-the-wire message format for our HLA 1.3 version and IEEE 1516 version).

What about HLA Evolved?

HLA Evolved is the next version of the IEEE HLA 1516 standard which was finalized in April 2010. VR-Link 4.0 will be released in July 2010 and fully support HLA Evolved.

Does VR-Link support IPv6?

Yes, VR-Link supports IPv6 for DIS. For other protocols VR-Link communicates with a middleware or RTI layer and therefore does not make any direct network calls. For example, if you have an IPv6 requirement you will need to use an RTI which supports IPv6.

What platforms can VR-Link support?

Just about whatever platform you *want* us to support. Extreme portability is high on the list of VR-Link's design criteria. In fact, we have often chosen not to use certain C++ features internally if they are not supported uniformly across a wide variety of platforms. Our standard releases support Windows VISTA, Windows XP, and Linux, but we have done demand-based porting of various releases to IRIX, Solaris, DEC Alpha, PC-Solaris, HPUX, AIX, PowerUX, VxWorks, and OpenVMS! Let us know what you need.

What about technical support and upgrades?

At MÄK, technical support is not just an afterthought. Our reputation for supporting our customers is one of the key reasons that people choose our products. When you call or email us with questions, you speak directly to our product developers who know the software inside and out. When you buy MÄK's products, you can be sure that MÄK will be in your corner as you work towards successful completion of your HLA/DIS project. We've even been known to be on the phone with customers during their HLA certification process, or during key events.

If you need assistance that goes beyond the scope of technical support, our engineering services group is available to do VR-Link customization, extension or integration on demand. We have had many customers not only buy VR-Link, but also buy our engineers' time to fully manage their transition to HLA or DIS compliance.

With maintenance, you are entitled to upgrades when they are released. Typically, new releases not only add support for the latest versions of RTIs, the RPR FOM, HLA Specifications, etc, but also try to maintain compatibility with older versions as well. For example, our current release supports many versions of the MÄK RTI, includes FOM Mappers for RPR FOMs 0.5, 0.7, 0.8, 1.0, and 2.0, and continues to support DIS 2.0.4, IEEE 1278.1, and IEEE1278.1a.

What are the future plans for VR-Link?

The world of modeling and simulation is always changing; VR-Link strives to keep pace. A new version of DIS (DIS Evolved) is in the final stages of development. Future versions of VR-Link will likely support it.

As always, we are working with various partners to discuss incorporating their commonly-used DIS or RPR FOM extensions into VR-Link. Please let us know if there are any functional or API improvements that you would like to see in VR-Link or any of our products.