AVS5 Compatibility: A Case Study Of A Large Software System In AVS/Express

Gary Oberbrunner
Member of Technical Staff
Advanced Visual Systems Inc.
garyo@avs.com

Abstract

This paper describes the development of the AVS5 Compatibility Kit, or AC kit, in AVS/Express. Although we will delve into many of the details of the implementation of the AC kit, the paper's emphasis is primarily on lessons learned in importing a large essentially foreign system into AVS/Express. This process is quite similar to the one undergone by any AVS/Express developer customer with a large body of code, especially one which is not simply a library of functions.

1. History

1.1. AVS5

AVS5 is AVS Inc's existing visualization product; it has been successful in the market for over five years. It uses a data flow model with modules representing units of computation. Each module has inputs, parameters, and outputs. Whenever the inputs or parameters change, the module "fires" and (perhaps) generates a new output. Execution is controlled centrally by the flow executive, which sends messages to each module telling it when to fire.

1.2. AVS/Express

AVS/Express shares many surface similarities with AVS5, but internally its architecture is quite different. Instead of modules, AVS/Express is composed of hierarchical objects which consist of state (data) and code. Both modules *and* data sets are objects in AVS/Express. Unlike AVS5 modules which had only one compute function, an AVS/Express object may have many methods, each of which triggers on a different set of events. In AVS/Express, wiring two objects together does not necessarily imply data flow between them; it only indicates a *connection* between them, which may be data flow in either direction or simply a path for notification of change in state.

Instead of being dataflow driven, AVS/Express is event driven. A change in any value creates a notification of that change, which generates events to all objects which have indicated an interest in hearing about it. That event automatically triggers a method in the receiving object; see Figure 1.

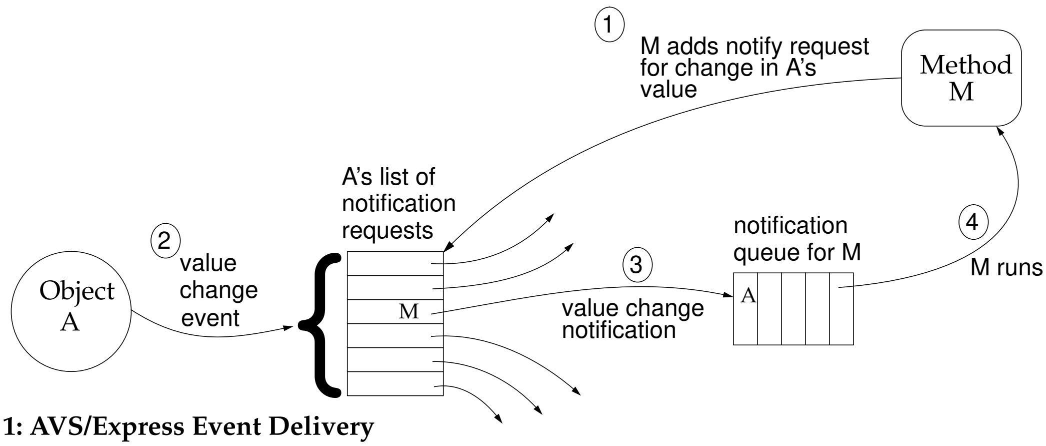

Figure 1: AVS/Express Event Delivery

2. Compatibility Architecture

The primary goal of the AVS5 Compatibility Kit, or *AC kit* for short, is to run AVS5 modules and networks in AVS/Express with as little change as possible. The architecture of the AC kit follows these basic principles:

- Use AVS/Express control structure
- Use AVS/Express execution model
- Reuse AVS kernel code where possible
- Run AVS5 protocol directly
- Run existing AVS5 binaries, do not require relinking with new libraries
- Port CLI calls and built-in modules to Express functionality

By reusing the original AVS5 source code to generate and parse the protocol, we maintain compatibility with existing module binaries and libraries all the way back to AVS2. Module binaries run as if they were talking to AVS5. The challenge then becomes to keep the internal AVS5 representation of the state of the world consistent with the AVS/Express representation, and translate events, connections and data as needed.

With this architecture, the AC kit actually runs existing AVS5 'mongo' and other module binaries, just like AVS5 did. They cannot tell via the protocol that they're not talking to AVS5. AVS/Express now reads the .avsrc file (ignoring certain configuration options which are overridden in AVS/Express), reads module library files, scripts, and networks. It runs the module description function and interprets the result. To achieve this level of compatibility was on one level very simple, and on another quite subtle and difficult.

It is simple to the extent that it works by running the same code that was in AVS5. The communication code is virtually unchanged, for instance. The startup code is largely intact. The module execution code is changed, but its structure is still the same (though now it is controlled from AVS/Express). This does not mean that we have enlarged the size of the AVS/Express binary by the size of the flow executive in AVS5. Many large parts of the flow executive are unneeded, the LUI support is replaced by Motif, built-in modules like the renderers and viewers are replaced by AVS/Express functionality, and many of the CLI calls simply call AVS/Express.

It is difficult because there are two competing representations of any given module, of the network, of the command-line interface, and of the data itself.

	AVS5	AVS/Express
module	`FLOW_MODULE`	`avsmod` (subclass of group)
network	Dataflow graph	Dependency graph
CLI	AVS CLI,.net,.scr	V and VCP
data	field, UCD, geom	AVS/Express field, GD object

Table 1: AVS5 and AVS/Express Compatibility Areas

The five major areas in which compatibility is required are modules, networks, CLI, data and widgets. Each of these is explored below.

2.1. Module Compatibility

In AVS5, a module is represented by a `FLOW_MODULE` structure. This describes the module's inputs and outputs, as well as connections and execution state.

To import it into AVS/Express, we create an object of type "`avsmod`", which is derived from the generic `group` object. It contains as subobjects all the inputs and outputs, parameters and their user interfaces, and a collection of methods invoked at various times (create, instance, invoke, destroy, save and restore, among others). A typical AVS/Express representation of an AVS5 module (in this case `clamp`) is shown inFigure 2. This object is automatically generated from the AVS description function's information.

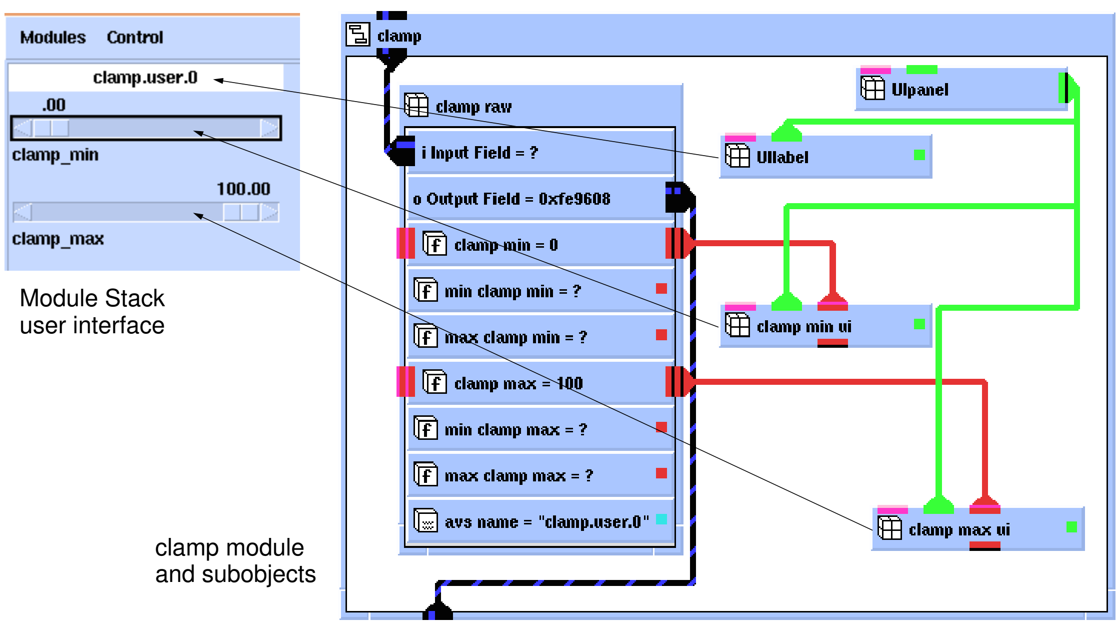

Figure 2: AVS5 `clamp` module imported into AVS/Express.

The `clamp_raw` submodule contains the actual interface to the AVS5 object, and the objects on the right implement the AVS/Express user interface for the min and max parameters. The input and output subobjects are preceded by "`i_`" and "`o_`" respectively to ensure that their names are distinct. In AVS5, port names were not required to be distinct because they were distinguished by number. In AVS/Express, each port name becomes a subobject which is referred to by name, so the names must be distinct.

Each parameter requires three subobjects in the raw submodule; one for the value (which is connected to a widget), and a pair for min and max. If the min and/or max values are unset (as they are in this example), the value is unbounded.

2.2. Network Compatibility

Although the AVS/Express network is fundamentally a dependency graph rather than a dataflow graph, for purposes of AVS5 compatibility they can be treated as equivalent, since AVS5 only deals with data sets flowing along its connections (it never exports entire objects). Update events from AVS/Express must be translated into activations of the AVS5 compute functions; this is described in Section 3.2.3 on page 6.

2.3. CLI Compatibility

The CLI protocol also uses the same parser and communication code as AVS5, with the functionality of each CLI command replaced in most cases with its equivalent in AVS/Express. CLI calls can be divided into several categories:

- Basic commands (`var_set`, `var_get`, `cat`, `help`, `sh`, `AVSmessage`, and script commands)
- Flow commands (`present`, network commands, `module` commands, `mod_lib`, port commands, parameter commands, `save` and `restore`)
- UI commands (lui commands, `manipulator`, `panel`, `reconfigure`, `window_mgr`)

  These are all stubs in the AC kit. To generate a UI in AVS/Express, wire up the corresponding UI modules.
- Application commands (`menu`, `choose_file`, `doc_browser`)

  These are superseded by UI functions in AVS/Express and by the AVS/Express online help system.
- Geometry and image viewer commands

  These commands currently do nothing in AVS/Express. The viewers can be controlled either directly, in V, or with the AVS/Express journaling mechanism.

2.4. Data Compatibility

Data conversion is a major part of the project, since each product supports many different data types. Converter modules need to be written to and from each major AVS5 type to ensure interoperability between AVS/Express and AVS5 modules (data is passed as usual with no conversion between AVS5 modules). Table 2 shows the correspondence. Pointer data types in AVS/Express are just passed as is, not interpreted.

AVS5	**AVS/Express**
field	field
UCD	field
geometry	GD object
colormap	datamap
molecule	pointer
pixmap	pointer
user-defined	pointer

Table 2: Data Types in AVS5 and AVS/Express

2.5. Module API Compatibility

Once the protocols are supported, API support is free. Users can run their existing binaries and continue to link against their existing AVS5 libraries. Modules running on other platforms (including Cray and CM-5, which use modified libraries) will continue to run normally.

3. Implementation

Following is a detailed look at integrating the AVS5 and CLI protocols into AVS/Express, with examples.

3.1. Techniques Used

In implementing the AC kit, we commonly employed several techniques to port a given AVS5 function:

1. Insert a call to AVS/Express in the middle of the AVS5 function.

   This is useful when AVS/Express functionality parallels AVS5 functionality, and AVS5 is "in control."

2. Have AVS/Express call the AVS5 function instead of its original caller.

   This is useful when the functionality is parallel but AVS/Express is in control.

3. Replace the AVS5 function altogether.

   In many CLI calls, for example, the AVS5 functionality is superseded by AVS/Express functionality.

4. Break up the function f into two parts, f' and f'', such that f' calls an AVS/Express function which then later calls back into AVS5 to f'' via an event or callback.

   Object instantiation, discussed below, requires a complex reorganization like this to ensure that all data structures are set up in the correct order.

3.2. Keeping Module Representations In Sync

To keep the two module representations in sync requires intervention at every phase of a module's existence. See Table 3 for a chart of the phases an object goes through in AVS5 and AVS/Express.

Phase	**AVS5**		**AVS/Express**
Creation	Read mod lib, create sys module	→	Make V template
Instantiation	Module command	→	NE or VCP, copy template
	create user module	←	
Connection	port_connect CLI command	→	NE or VCP connect
			make AVS/Express connection
	make AVS5 connection	←	
Execution			update method runs
			update `FLOW_MODULE`
	execute remote module	←	
		→	update `avsmod`
Deletion	delete CLI command	→	destroy object
	destroy AVS5 module	←	destroy callback

Table 3: Object Phases.

3.2.1. Object creation

Note that creation is distinct from instantiation: a module (either in AVS5 or AVS/Express) is created when it appears in the module palette. In both cases the template module is to be thought of as a class type rather than an instance. It is instantiated only when it is dragged down into the workspace.

AVS5 module objects are created in two ways: via AVS/Express when it reads in a module library (e.g. at startup), or via the CLI `read_module` command.

The basic AC kit mechanism is the same in either case. The template `FLOW_MODULE` is created first (by the standard AVS5 code), then later AVS/Express creates an `avsmod` corresponding to that `FLOW_MODULE`.

Via AVS/Express: When the AC kit starts up, it reads, via the "init_code" V properly on the AC kit library, the user's .avsrc file (using the same code as in AVS5) to determine the module libraries in use. It then reads those modules in into AVS5 template `FLOW_MODULE`s. Later, after AVS/Express is fully initialized, AVS/Express goes through that list and creates an `avsmod` for each one. These template `avsmod`s live in the AVS5 Modules palette page. However, to decrease startup time, the `avsmod` is created as a *buffered* object, which means that the subobjects (the raw submodule and all the UI objects) are not created until the module is instantiated. Only a pointer to the template `FLOW_MODULE` is stored at creation, to make instantiation faster.

Via AVS5: In AVS5, the user can also read modules dynamically via the CLI `read_module` command. This works the same way as above in AVS/Express; the template `FLOW_MODULE` is created by the `read_module` command as usual, then a corresponding buffered `avsmod` object is created by AVS/Express to live in the module palette.

3.2.2. Object instantiation

An object is instanced when it is dragged into the workspace or defined in a V or CLI script or network file.

Via AVS/Express: When a module is dragged from the palette into the workspace, the template object's copy method is run to create the module from the template. (This is true in general, not just for AVS5 compatibility modules.)

If the AC kit object was buffered (see above), it must be *opened* before the copy method runs. To open it, the `avsmod`'s `open` routine is called, which creates all of the module's subobjects, including the raw module and the user interface widgets. The list of inputs, outputs and parameters is determined by running the module's description function. Since these objects are created dynamically from the AVS5 description, if the user starts up AVS/Express with a different .avsrc, or different command-line module library args, the correct objects will always be available in the palette, with the correct ports.

The `avsmod` copy method then creates a `FLOW_MODULE` (not a template this time but an active one) from the template, with an appropriate unique name, inputs, outputs, and parameter values.

Via AVS5: The AVS5 `module` CLI command in the AC kit creates an AVS/Express avsmod object rather than a FLOW_MODULE. Once the object is copied, the avsmod copy method is automatically invoked, creating the FLOW_MODULE.

3.2.3. Object execution

The main problem here is keeping the two representations in sync with each other at the appropriate times. Whenever AVS5 code is running (e.g. running a compute function), the `FLOW_MODULE` must be up to date, and whenever AVS/Express is running, the `avsmod` must reflect the true state of the module.

To update the module, we copy the inputs and parameters from the `avsmod` to the `FLOW_MODULE`. If the input is one of the pointer types in Table 2 on page 4, we just copy the pointer. Outputs will be modified by the compute routine, so they don't need to be copied over. We set the sequence number of the `avsmod` to indicate when this module last ran, so next time we only copy parameters and inputs which changed since the module ran.

When the module finishes running, it may have changed its outputs and/or parameters. So we must extract the changed values from the `FLOW_MODULE` and insert them into the `avsmod`. Note that the output may live in the module running on another host; we do not send the data back to AVS/Express unless at least one downstream module is an AVS/Express module which will need that data anyway.

We must also update the `avsmod` when a module changes one of its (or some other module's) parameter values, so that change is reflected in the user interface immediately. This requires judicious use of `OMpush_ctx` and `OMpop_ctx` to control event delivery.

Use of `OMpush_ctx` and `OMpop_ctx` is critical here: for instance, when a module updates one of its own parameters with the CLI `parm_set` command or `AVSmodify_parameter`, we want that change to appear in the user interface immediately. However, we don't want that UI change to send an 'parameter changed' event to the module; that would make the module run again! The first argument to `OMpush_ctx` is the *context ID* object; in that context, no notifications will be generated for any events on that object. Judicious use of contexts this way can filter out undesired events while letting others through, in this case the events which update the UI, and potentially invoke other modules as a result. The additional benefit of wrapping the module execution in a context is that any events queued as a result of the module's execution (or any CLI calls that it might make) are queued until the matching `OMpop_ctx`. Those events are then evaluated for dependencies and executed in proper order. See Figure 3.

```
push ctx: ID = module object
run module compute function
push ctx: ID = parameter object
change parameter value
pop: user interface update events delivered
continue running module
pop: module update events delivered
```

Figure 3: Contexts in AVS5 module execution.

The last thing which must be up-to-date before running the module is the module's connections. If the module is connected to other AVS5 modules, the connection is made directly so those modules can communicate with each other as they did under AVS5. If the module is connected to an upstream AVS/Express module (i.e. the module's input comes from AVS/Express), we note this with a special flag in the `FLOW_MODULE`, and instead of getting the data from an AVS5 connection we pass it in directly from the AVS/Express element. If the module is connected to a downstream AVS/Express module, i.e. it has an output which is connected to an AVS/Express object (note that it might also be connected to an AVS5 module, since an output port may have many connections), when we run that module we force it to send its output to the kernel, so it will be available to the AVS/Express object(s).

3.2.3.1. Subroutines

To run the AVS5 module's compute routine it suffices to bring the AVS5 kernel's idea of the module and its connections up to date as described in the preceding paragraph. Then the kernel will have no idea that anything is different from AVS5, and will run the module (whether local or remote, subroutine or coroutine) normally.

In simplest terms, this means that just before running the module's compute function, we update the `FLOW_MODULE` and the connections from the `avsmod`, and when the module finishes running, we update the `avsmod` from the `FLOW_MODULE`.

3.2.3.2. Coroutines

AVS5 coroutines run differently from normal modules. They are, in a sense, "always running." They send their outputs whenever they want and get their inputs whenever they want. Synchronous coroutines call `AVScorout_wait` to wait until their inputs change, whereas asynchronous coroutines get their inputs and send new output values without regard to the state of other modules or the kernel.

Coroutine functions in AVS5 are handled via callbacks into the AVS5 kernel. There are callbacks for getting inputs, sending outputs, waiting for inputs to change (this is a two-phase callback), and marking the module as changed. These callbacks are executed in AVS5 via a `select` on the module's control file descriptor such that when the module makes a request, the select returns and processes the request.

The AVS5 select loop must be integrated into the existing AVS/Express select loop in order to process both AVS5 and AVS/Express requests whenever they occur. This is done by mapping the `AVSadd_select` and `AVSdelete_select` calls into the appropriate AVS/Express object manager calls, `EVadd_select` and `EVdel_select`, which add and delete descriptors to the AVS/Express select queue.

All that remains is to synchronize the module representations at the appropriate times, as discussed in the previous section, so that the module gets the correct inputs, and wake the module up when an AVS/Express event indicates that an input has changed (if the module is in `AVScorout_wait`).

3.2.3.3. Built-in Modules

The AVS5 kernel comes with a set of built-in modules including the geometry, image and graph viewers, display image, and generate colormap. These modules are not run out of mongo, but are compiled into the same binary with the kernel. Since each of these modules has an analog in AVS/Express, the built-in modules have been removed from the AC kit. The AC kit provides wrappers for the AVS/Express modules with similar but not identical interfaces to the AVS5 ones. The look and feel are entirely Express's.

Viewers: The geometry and image viewers are replaced with a simple V wrapper around AVS/Express 3D and 2D viewers, respectively. Both of these wrapper macros take an array of geometries or images respectively, to allow multiple input connections. Display image is similar, but like AVS5 display image, it takes only a single input. There is currently no replacement for the graph viewer, but the annotation and graphing kit will eventually be used to replace it.

Generate Colormap: The code for generate colormap in AVS5 is actually a trivial module with one colormap parameter which is copied directly to the output. All the work is done in the colormap editor widget. In the AVS/Express version, the corresponding Datamap editor widget is used, with a datamap-to-colormap converter module after it.

3.2.4. Object deletion

Object deletion is, like object creation, handled in AVS/Express; this time via the destroy callback. The AVS5 `module delete` command's guts are replaced by a call to `OMuser_destroy_obj`, which invokes the AVS/Express destroy callback, which in turn calls the AVS5 module destruction code, which was formerly called directly from AVS5.

In this way modules can be deleted from the AVS5 CLI or from AVS/Express (either from V or the network editor) following the same path. If the module is deleted via AVS/Express, the destroy callback directly deletes the AVS5 module. If AVS5 deletes the module, it invokes AVS/Express to do the work, with the same result.

3.3. Flow executive

AVS/Express is used as the flow executive for both its own modules and the AVS5 modules. It, rather than the AVS5 flow executive, oversees execution, and its underlying framework is used to connect modules together into networks. The AVS5 flow executive does not run at all. When an output changes, AVS/Express generates the event and notifies (runs) the appropriate methods, which may be AVS5 module methods or AVS/Express methods. Since AVS5 modules already have no knowledge of what's invoking them except via the protocol, making this change is not difficult, except for integrating the AVS coroutines into the execution framework: some care must be exercised to see that the coroutine gets its inputs only when they are ready, and doesn't send its outputs at a time when Express isn't ready.

3.4. Reading AVS5 Networks

When AVS5 executes a `read_net` or `read_script` command, each individual CLI command of the network or script is interpreted by the AVS5 CLI interpreter. In both scripts and networks, we use an `OMpush_ctx/OMpop_ctx` pair around each CLI command, to make the events generated by that command atomic (i.e. delay them until the command is finished). In the script case, like executing a live V command from the VCP, this also makes the results immediately visible. In the case of `read_net`, it is desirable to delay instantiation of all the widgets and execution of the modules until the network is fully in place. This greatly speeds up loading of the net. To accomplish this, we use an `OMpush_ctx/OMpop_ctx` pair with a `child_mode` of `OM_CTX_PARENT_NOTIFIES` around the whole network like so:

```
OMpush_ctx(OMroot_obj,OM_STATE_PROG,OM_CTX_ALL_NOTIFIES,
           OM_CTX_PARENT_NOTIFIES);
CLIsource(filename);
OMpop_ctx(OMroot_obj);
```

The third argument (`event mode`), `OM_CTX_ALL_NOTIFIES`, says not to throw away any events, even ones for the context ID (which in this case is the application object into which the network is being read). More significantly, the fourth argument (`child mode`) has the value `OM_CTX_PARENT_NOTIFIES`. This causes any sub-contexts of this context (such as the one surrounding each command) to act as if it had `OM_CTX_PARENT_NOTIFIES` ORed into its event mode. This means that when each command pops its context, the events queued by that command are not executed immediately, but continue to be held until the parent context is popped, i.e. when the network is fully read in. At that time, all pending events are sorted into dependency order and executed, displaying the results of the network and kicking off the module executions.

We do not *write* AVS networks from AVS/Express, because the network structures are quite different, and Express modules with no AVS5 equivalent have been added (e.g. the UI modules).

3.5. Command Interface

Each of AVS5 and AVS/Express expects to have access to a command line interpreter to accept interactive commands. In AVS/Express this is the VCP; in AVS5 it is the CLI. Obviously stdio can only be used for one at a time. Following the same philosophy as for the flow executive, in the AC kit, AVS/Express gets primary access to the command line.

Since the VCP can be extended to accept new **$**-prefix commands by calling `OMadd_command_parser`, the AC kit adds the **`$avscli`** command which sends its arguments to AVS5 to be parsed. And naturally the AC kit can read a .scr (script) or .net (network) file of CLI commands which are parsed by the AVS5 CLI parser, and the results converted to AVS/Express functions. This works as above, since it does not require stdio.

The porting of the individual CLI commands is done on a command-by-command basis. Commands like `module`, `port_connect` and `parm_set` are implemented by creating an AVS/Express object corresponding to the module, connecting two AVS5 modules together, and setting the value of the AVS/Express parameter subobject, respectively. Some CLI commands, such as those for the geometry and image viewer ones, simply call stub functions which do nothing since those functions are replaced with AVS/Express viewers which behave differently. Some of these may be implemented in future releases.

3.6. Data

AVS5 and AVS/Express also have different data models. Fortunately both models fundamentally rely on arrays of data, so converting AVS5 data to AVS/Express format is rarely difficult. Converting backwards from AVS/Express to AVS5 format can only be done in those restricted cases where the type maps naturally onto an AVS5 data type. The AC kit comes with a suite of converter modules to convert common formats directly; see Table 2 on page 4. Note that, following the AVS/Express data model, in most cases these modules do not have to copy the data; they simply generate a new header. This makes them quite fast. Of course if the data must be transferred from process to process, the network time will dominate rather than the format conversion time.

3.6.1. AVS5 Geometry

Converting geometry is a bit harder than converting fields, because geometry edit lists must be converted to AVS/Express graphics display (GD) objects, and the mapping is not 1-to-1. Also some edit list items like light sources and textures belong in the viewer in AVS/Express rather than in the geometry data type. Still, most edit lists (including all geometry and transform data) can be converted to AVS/Express format with little or no loss.

3.7. Widgets

AVS5 had its own proprietary widget set, not built on the X toolkit, called LUI widgets. AVS/Express has replaced these with standard and extended Motif widgets under X, and Windows widgets under Windows NT. Due to the difficulty of supporting the existing LUI calls with Motif widgets, there is no support for LUI in the AC kit. The LUI CLI calls are stubbed out so they don't cause errors, but they don't operate the user interface. AVS/Express methods must be used instead. This means that widget layouts specified under AVS5 are not used in AVS/Express; the widgets are laid out in the ModuleStack in a default order.

This is not a great problem however, since the widgets are all different sizes anyway so the old layout would not have worked in any case. You can, of course, use the AVS/Express widget editor to place the widgets in any desired configuration and save that as a V file, or even specify the layout programmatically.

LUI widgets also allowed the end user to click on them and change the widget type dynamically. Currently AVS/Express does not support any widgets of this type, so that functionality does not appear in the AC kit.

3.8. Network Editor

The AVS5 Network Editor is replaced by the AVS/Express one. Color usage for connections is different in AVS/Express, but for compatibility's sake AVS5 data types use the same colors they did in AVS, but preceded by black.

4. Conclusions: Lessons for importing large software systems

Though it appears similar, AVS5 is really not at all like AVS/Express in many deep ways, so it serves as a good test case. Other seemingly less related cases are easier, because they don't need to be wired in at so many levels. Some of the lessons learned are:

- It works! It can be done.
- Like any large project, careful planning is mandatory.
- OM architecture is flexible enough to accommodate a very different structure.

  It can map the data, execution paradigm, and control structure of many target applications. Adapts to a very different underlying scheme, different communication architecture, and different execution model.
- Push/pop is powerful for controlling execution and delivery of events
- It requires careful but not major surgery to the existing application, and a deep enough understanding of the application to know where to change it for minimal impact.
- Total time to design and implement the AC kit was approximately 9 staff-months.

Biographical Information

Name:	Gary Oberbrunner
Professional Title:	Member of Technical Staff
Company:	Advanced Visual Systems Inc.
Office Telephone:	(617) 890-4300
FAX:	(617)890-8287
email address:	garyo@avs.com

Paper Title:	AVS5 Compatibility: A Case Study of a Large Software System in AVS/Express
Educational Background:	BSEE 1984, MIT
Professional Background:	MASSCOMP 1984-1989 Thinking Machines Corp. 1989-1994 AVS Inc. 1994-present