OpenIGS 2:
A Non-intrusive Middleware Framework for the Integration of Application and Simulation Components.

C. Skluzacek[(1)], P. van der Plas[(2)]

[(1)]DutchSpace / Cesium Software
Postbus 59614
1040LC Amsterdam, The Netherlands
cesiumsw@gmail.com
c.skluzacek@dutchspace.nl

[(2)]ESA/ESTEC
Keplerlaan 1
2200 AG Noordwijk, The Netherlands
peter.van.der.plas@esa.int

INTRODUCTION

OpenIGS 2 is the continuation of the OpenIGS project started in 2002, which itself grew out of a number of projects related to the integration of Image Generation Systems with real-time simulation systems such as EuroSim and SimSat. The original Image Generation Software (IGS) developed in 1999 was intended to provide realistic 3D visualization for the European Robotic Arm (ERA) Mission Planning and Training Equipment (MPTE) which used EuroSim as its simulation platform running on Silicon Graphics Onyx computers. It was quickly realized however that this sort of visualization functionality would be useful on a number of other projects which used different hardware and software platforms. The original IGS system however was very monolithic and did not lend itself easily for porting to other platforms.

After a number of iterations, the OpenIGS project was started with the intention of providing an open architecture to address some of the issues of the previous generations. Among the most important of these issues were:

- Flexibility: application developers should be able to pick and choose what software components they wanted to use as well as which implementation of the component. Among other things, this was to avoid vendor "lock in". A prime example of this was the use of SGI Performer and Paradigm Vega in the original IGS which have since become defunct.
- Reusability: software components developed for one project should be, if designed generically enough, reusable in other contexts and projects. This is a basic tenet of software design and implies decoupling from other software components. OpenIGS promotes this even further.

While conceptually the OpenIGS architecture was effective, it did have shortcomings in its application interface as well as some performance issues. It also suffered from a lack of a formal support and maintenance path resulting in an ad hoc development environment.

One of the primary goals of OpenIGS2 is to create a sustainable and maintainable environment for users of OpenIGS. In addition to this goal, the architecture has been completely overhauled to take advantage of some of the new techniques and technologies in software development, to provide some more built in functionality, and to make the system more usable and flexible for developers and users, as well as to improve performance.

The OpenIGS2 activity is run as an ESA contract in the GSTP-4 programme, with DutchSpace as the prime and Cesium Software and Task24 as subcontractors. DutchSpace is responsible for the project management, testing and validation activities. Cesium Software is providing the design and implementation of the software system. Task 24 is developing the editor and is responsible for setting up the support, maintenance and delivery infrastructure which will be hosted at DutchSpace. This paper provides a high-level description of just the software architecture as well as some small examples for how to use the OpenIGS system.

MOTIVATION

Although the original primary goal for the OpenIGS system to provide 3D visualization capability for real-time simulators such as EuroSim is still valid, there are a number of technologies and tools available today which also provide visualization capabilities. The difficulty however usually comes when integrating these tools into an existing environment. This is especially true in the space industry where specialized or one-off applications are not uncommon. Inevitably, some combination of writing specialized "glue" code, scripts, and data filters must be employed as adaptor interfaces in order to get these tools to work with a system. Also inevitable is that this glue code is so specialized as to be non-reusable and difficult to maintain, especially as new releases come out, and interfaces or technologies change. While a certain amount of such development is unavoidable, the OpenIGS system can help mitigate and isolate these circumstances.

In this sense, the OpenIGS system functions as a middleware framework to encourage the independent development of reusable functionality while isolating and minimizing the amount of application specific elements. Ultimately, an application becomes simply a collection of independent functional modules which cooperate and interact with each other with OpenIGS acting as the catalyst. Even if there is a piece of application specific functionality, it is encouraged to develop these as an independent module primarily to avoid dependence on other components but also because it may become useful to someone else in the future.

The concept of middleware is also not new. There are also a number of technologies available that function as middleware, most notably: CORBA, COM, RPC, SOAP, etc. They all have their pros and cons, in the case of OpenIGS, the pros and cons could be considered:

Pros:

Non-intrusiveness: Unlike most other middleware, developing or using a software component with OpenIGS does not require any changes to the software component itself, nor does it require any special tools other than a standard compiler to develop this software. All interfaces with the component and the OpenIGS framework are developed external to the software component. A consequence of this is that in many cases, existing software components can be used with the system and developers of new software components need not have any specific knowledge of the OpenIGS system[1].

Lightweight: The OpenIGS Kernel library is fairly small, although it can attribute its compactness to the heavy use of C++ templated code. It consists of roughly 40 classes and

Patterns use: many common software design idioms and patterns are employed, supported, and encouraged for use. With user components, the Observer, Composite, Factory methods are encouraged and internally, the Strategy and Memento design patterns are used.

Cons:

Platform availability: At the moment only a C++ interface will be available. Experiments have been made for a Java binding although there has been no official request for this.

CONCEPTS

In object-oriented design (OOD), software applications consist essentially of a number of objects of different types that communicate and interact with each other. The mode of communication may come in several forms, but the most common is one object directly calling a method on another object passing any necessary information in the form of parameter data. Although this is pretty straightforward, it does require that the calling object have explicit knowledge of the called object. In other words, the objects are *tightly coupled*.

To achieve flexibility and reusability, heterogeneous software components need to be able to communicate with each other without explicit knowledge of one another. In other words, objects need to remain *decoupled*. Therefore, one of the primary functions of the OpenIGS system is the shielding of functional software components from anything that may promote coupling between two (unrelated) software components. The sources for tight coupling can occur at any of the levels involved with the communication between two objects, namely:

***Type/Object*:** in order for communication to occur, one object must know the type of object receiving/sending the communication and also the specific instance of that type.

***Method*:** Once an object instance is obtained, the appropriate method must be selected.

***Signature/Parameter*:** Once an object's method is identified, it must be known what the signature of that method is, which involves knowing the type, order and valid values of each of the parameters of the method.
OpenIGS employs the following techniques to relieve these sources of coupling:

1In practice however, this is not always the case and some adaptor interfaces may need to be developed. In addition, not all programming constructs are supported but the most common ones are.

Linking: also known as: “Observer Pattern”, “Signals/Slots”, “Delegates”, “Callbacks”, etc. This technique is fairly established and allows a software runtime object to send information to any other object when some event occurs, without the need for explicit knowledge of the receiving objects. A number of libraries provide such functionality, most notably: Qt's Signal/Slot framework and Boost's signal classes. While the Qt framework is arguably one of the more popular and has been around for quite awhile, it has quite a few shortcomings. The Boost signals family of classes is very generic and avoids almost all of Qt's Signal/Slot issues.

Binding: In traditional Observer/Signal/Slot implementations, the function signature of an Observer object must exactly match the function signature of the Source object. This in itself however is a source of coupling because the Observer object must in a sense “know” in detail what the Source object is communicating, otherwise they cannot be connected. Binding provides a greater deal of flexibility by allowing parameters to be rearranged, a value to be transformed or explicitly assigned to a parameter, or a parameter to be completely ignored altogether. The Boost.bind library is an excellent implementation of parameter binding and has been proposed for C++0x.

Conversion: Binding is just part of the story. While binding allows signature parameters to be manipulated, their *types* still need to be identical. This is also a form of coupling. A typical example might be a Graphical User Interface (GUI) Textfield configured to set a numerical parameter in a simulation. The output from the Signal from the TextField object is a string, but the simulation expects numerically typed data. OpenIGS provides an automatic conversion process to ensure that despite disparities in signature parameter types, two objects can still be linked.

In effect, the Binding and Conversion capabilities allows two objects to be linked together whose signatures are *conceptually* compatible. In the case of the GUI Textfield and the simulator parameter, the data is conceptually numeric, whether it comes as a string type or as a numeric type should be irrelevant.

Dynamic Loading: also known as “plug-in” is a process of loading functionality at runtime. OpenIGS not only enables the dynamic loading and creating of new objects at runtime, but also the definition of new *types* at runtime. This together with the Linking/Binding/Conversion concepts contributes greatly to flexibility and reusability. An ultimate example of this would be to substitute a completely different implementation of a GUI library or a 3D graphics library should another one be desired (due possibly to license issues or an old implementation become defunct), without affecting any of the other systems components.

HIGH LEVEL ORGANIZATION

The OpenIGS is divided into several subsystems and entities whose organization and interactions shown in Figure 1.

Kernel: provides the infrastructure to support the concepts described in the previous section

Functional Classes: provide the actual functionality needed by an application. These can be existing classes that are completely independent of and external to OpenIGS.

Modules: uses the facilities provided by the Kernel to act as a thin wrapper providing interface information about the Functional Classes to the rest of the OpenIGS system.

Configuration Files: define the behavior of an OpenIGS-based application. The Editor generates/modifies configuration files which in turn are loaded by the Kernel at runtime to define the Application.

Editor: provides an intuitive user interface to allow developers to create and correctly write configuration files quickly.

Application: a collection of Modules configured by xml files which provide specific large-scale functionality.

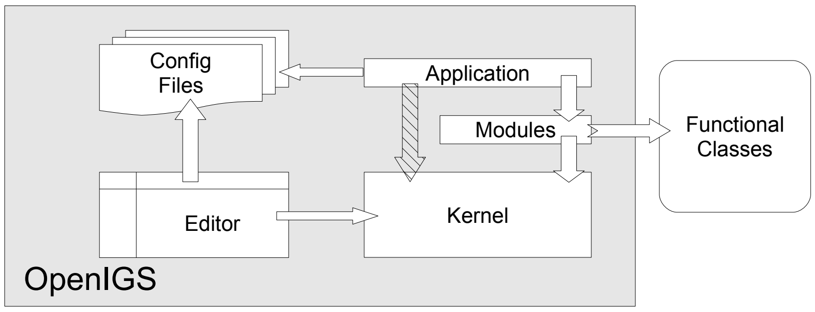

Figure 1: High Level Organization

KERNEL ARCHITECTURE

The Kernel provides the infrastructure and services for an OpenIGS Application. Its primary responsibilities are to maintain the various Module types and instances and to manage the communication between instances at runtime. It also dynamically loads the application's configuration files at start-up. The Kernel itself provides only minimal application level functionality. The primary components of the Kernel are: Reflection, Linking, Binding, Conversion, the Database and External Communication.

Reflection

The Reflection facility provides information about data types used in an application. All types that are to be used within an OpenIGS Application must be registered with the Reflection component. OpenIGS uses this information to determine what types of communication a component supports and to validate the information being passed to and from the component. It also defines a unique human-readable identifier with each type which can also be used when reading and writing to external data formats.

The main type that is reflected by the Reflection facility is the class definition. Classes are data structures which may contain data and operations on that data and is what is referred to as a software component. The Reflection facility maintains information about the static class hierarchy, i.e. the direct base classes of the class, as well as any pertinent methods available in the class. In addition, it also supports information about signals available in the class. Signals are a generic implementation of the Observer pattern and are means for a component to send information to other components without explicit knowledge of them. However, since Signals are not directly supported by the implementation language (C++), they must be provided by another software library. OpenIGS supports the boost::signal implementation although other signal libraries can be supported by providing an appropriate adaptor interface.[2]

Other data types supported by the Reflection facility are: the built-in numeric types (bool, integer and floating point), enumerated types, strings (std::string and std::wstring), smart pointers[3]. These types are used primarily to provide information about the parameters passed during a class method invocation.

Besides scalar types, the Reflection facility supports the concept of Containers which are types that maintain a collection of instances of another type. OpenIGS uses the Container concept to support the Composite pattern by allowing components to specify child objects. Composite structures are a common pattern employed for instance in 3D graphics scenegraph and 2D user interface toolkits to construct a hierarchy of objects with parent-child relationships. The Reflection facility provides support for the STL sequential container types (deque, list, multiset, set, and vector) and other container types can be supported by providing the appropriate adaptor interface.

To simplify communication between components, the Reflection facility also uses the concept of Properties. A Property is a related group of methods and signals of a class type that pertain to one particular aspect of that class. The methods and signals maintained relate to retrieving the value, setting the value, and signalling when the value is changed. One or all of these may be specified in the Property. The Property associates a single identifier with these related methods and signals so when the component is used, the appropriate method or signal is selected depending on the context.

As stated earlier, the Reflection information for a particular component is provided completely external from the component. The Reflection API is intended to make the specification of reflection information for a component as simple and intuitive as possible. Listing 1 shows an example of the declaration of a simple class. Listing 2 provides the corresponding declaration and definition for the reflection information for the simple class.

```cpp
namespace demo {
class BaseClass; // defined elsewhere

class DEMO_API SomeClass : public BaseClass {
public:
  SomeClass();
  int getValue() const;
  void setValue( int newVal );
  boost::signal<void (int)> valueChanged;
};
} // namespace demo
```

Listing 1: Class Example Declaration.

2 Libsigc++ comes to mind. The Qt signal/slot mechanism is not possible due to its use of non-standard syntax and the special moc compiler. OpenIGS does provide wrappers around the Qt signals to convert them to boost::signals.
3 Namely, boost::shared_ptr, which is a candidate for C++0x. Other smart pointers can be supported by providing the appropriate adaptor interface.

```
CLASSINFO( demo::SomeClass,                 - the class type
           “demo”, “SomeClass”, DEMO_API, - namespace, name, and linkage (for Windows DLLs)
           demo::BaseClass );              - a list of direct base classes

template<>
igs::ClassInfo const &
classInfo<demo::SomeClass>()
{
  static ClassInfo info = ClassInfo().init<demo::SomeClass>()      - initialize with base
                                                                     class information
      .addProperty(
          Property( "Value", Property::Persistent,                 - name and flags
              Operation( "getValue", &demo::SomeClass::getValue ), - getter Operation
              Operation( "setValue", &demo::SomeClass::setValue,   - setter Operation
                  Parameter( "value",                              - Parameter name
                             0,                                    - default value
                             Parameter::In                         - Parameter mode
                  )

              ),
              makeSignal( “valueChanged”, &demo::SomeClass::valueChanged,  - Signal
                  Parameter( “value”, 0, Parameter::Out )                  - Signal Parameter
             )
          )
      )
  ;
  return info;
}
```

Listing 2: Class Reflection Information Declaration and Definition.

By using direct pointers to the class method, type information can be automatically extracted using template specialization techniques. But this also serves other purposes, namely that the objects with a Reflection definition can be maintained and executed anonymously. It also provides the opportunity for optimization as will be discussed later as well as compile-time error checking to ensure that the arguments are appropriate.

Database and Modules

The Database is the repository for all the classes registered in an application and all instances of those classes. Using a dynamic loading mechanism, new classes can be loaded and registered at runtime. From here, class information for each of the loaded classes can be obtained using the Reflection ClassInfo API. Using Factory methods, new instances of a class can be created after it is registered.

After a new instance of a class is constructed, a Module interface to the instance can be retrieved from the Database using the class name specified in the ClassInfo object and the name of the instance. A Module object is simply a thin wrapper around the actual class instance object and provides methods for retrieving the ClassInfo information. The ClassInfo can then be used to actually execute operations on the underlying object by simply using the Operation's name. In this manner, Module objects can be passed around the system and methods called on the underlying object without actually having to know the specific type of the underlying object as is shown in Listing 3.

```
igs::Database database; // previously populated with classes
igs::ModulePtr someModule = database.getInstance( "demo", "SomeClass", "someName" );

igs::ClassInfo someClassInfo = someModule->getClassInfo();
igs::Operation setValueOp = someClassInfo.getOperation("setValue");
setValueOp( 5 ); // calls SomeClass::setValue(5)

// alternatively, can execute the operation on the Module itself
// using the operator[] syntax:
(*someModule)["setValue"]("5"); // automatically converts to int before passed to setValue.
```

Listing 3: Retrieving Module instance and Executing an Operation.

As can be seen in Listing 3, there is no explicit knowledge of the SomeClass type as declared in Listing 1. This capability is essential in order to be able to dynamically load new Module classes without recompiling the whole system. This is also essential for configuring Links which is the subject of the next section.

Links

The Link component of the Kernel sets up communication between two objects using information from the Reflection API. Specifically, using the ClassInfo information discussed above, Linking associates an Operation or Signal from a source object to an Operation or Signal on one or more target objects. When a Link is executed, the Parameter values from the source's Signal or Operation are transferred to the target's Operation and the target's Operation is executed.

Binding

When linking the Signals and Operations of two components together, it is often the case that the signatures (type and order of parameters) of the transmitting and receiving Signal and Operations do not always match exactly. This is where Binding and Conversion come in to play.

Binding enables the reordering, mapping and transformation of Parameters coming from the source object to the target object. Using the Parameter names, a mapping from the source Parameter to the target Parameter can be made to change the order. In addition, the following operations can be be applied to the Parameter value as it is transferred from the source to the target:

- The value can be modified/transformed. This is useful for example in cases where the source provides values in radians and the target expects them in degrees. The appropriate scaling factor can be applied before it reaches the target.
- The value can be set to a specific value.
- The default value as defined by the target object can be used for the value.
- The current value as defined by the target object can be used for the value. In this case, a method must be specified from the target object to retrieve this value.

As a possibility for future work, dimensional and unit analysis could also be incorporated into the binding process. Dimensional and unit compatibility is another contributor of coupling between components. For example, if a simulator provides length data in meters, and a visualization system expects it in feet, the appropriate conversion factor needs to be applied. The Binding facility could be used to automatically apply the appropriate factor so different components would not need to explicitly synchronize their units. This would also allow uses of different scales among heterogeneous components and also ensure that correct dimensions are being used.

Conversion

Similar to the way type information is stripped from the interface of a Module, in order for data to be passed around from a source to a target object, the type information for Parameter values need to be stripped so it can pass through the system without any dependencies on its actual type. This is not to say that the type information is lost however, just that it isn't immediately accessible. The Kernel uses a type-neutral Value class to hide the type information of data and only expose it when necessary. The Value class is based on the Boost.Any class but provides some additional enhancements.

The main additional enhancement that the OpenIGS Value class provides over Boost.Any is the ability to automatically convert the type of a Value. This eases the dependence of the actual type of the data being passed between two components and puts the focus more on what is being represented by the data. The source object can then transmit its data in whatever format it deems appropriate and the target can receive in its expected format, just as long the concepts are compatible then they can communicate directly. For example, numeric data can be represented by an actual numeric type (i.e. int32or float) or come in string form (i.e std::string). The Conversion facility automatically converts between the two. The other two types of concepts are enumerated data, which can exist as either an enum type or in string form, and class data, where the Conversion facility can automatically convert between two related class types for example the base and subclasses in a class hierarchy.

External Communication

Using the information provided by the Reflection facility about a particular class type, the state of a class instance can be sent to an external source (i.e. writing to a file) by using the retrieval method of the class' Properties. Conversely, the state of a class instance can be restored altered by using the set method of the class' Properties. This is the mechanism by which the Memento pattern is implemented.

The Parser class of the OpenIGS Kernel is responsible for translating between the internal Reflection data structure and an external format. OpenIGS provides a translation to and from XML files and other formats can be supported by providing a different Parser implementation. The IO class provides the actual data transmission mechanism for transferring information to/from an external source. OpenIGS provides IO module implementations for file-based IO as well as network-based IO using the TCP/IP and UDP protocols.

The primarily use of the External Communication facility by the Kernel is to “bootstrap” the application by reading information from configuration files. The configuration files contain information about what Modules to load, how to configure them, and for setting up Links between Module instances.

Other common uses of the External Communication facilities would be:

- use a network module (TCP or UDP) as the IO object to allow sending and receiving application configuration data at runtime from a remote application thereby providing RPC type functionality.
- Use the File based IO and a Link configuration to write data to a file for recording/playback/logging facilities.
- Set up an IO object to read data from a telemetry link combined with a custom parser to translate the information into a visualization application.

MODULES AND APPLICATIONS

Using the infrastructure provided by the OpenIGS architecture, combined with the ability to link Modules together, complex and varied application behaviour can be developed. In essence, the Modules used and how they interact with each other define an application. Using the configuration mechanism, complete applications can be created from just a set of configuration files. Consequently, the OpenIGS distribution provides a simple generic driver program that loads the configuration files thereby making it possible to create completely new applications without any programming.

Along with the basic infrastructure provided by the Kernel, the OpenIGS distribution includes Module classes needed for creating basic 3D visualization applications as well as distributed applications.

- 3D Graphics based on OpenSceneGraph
- 2D Graphical User Interface (GUI) based on Qt
- EuroSim external simulation access interface Module
- Network IO Modules using TCP and UDP protocols
- Space-industry related Modules

These Modules can be used to create a variety of applications. Using the Network and Graphics Modules in particular, a variety of types of applications (Figure 2) can be developed that fall under the following general categories:

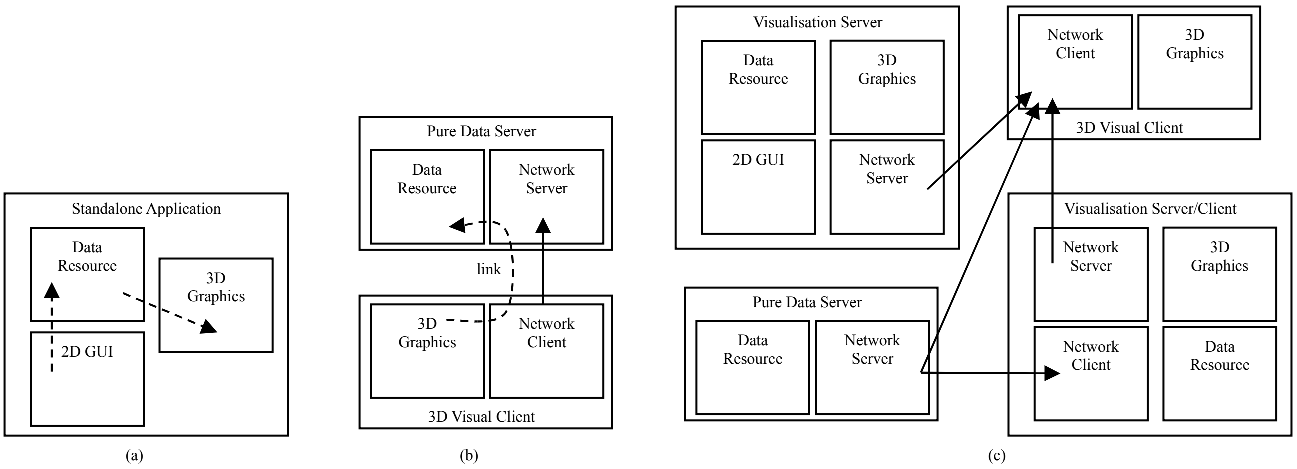

Figure 2: Example Application Configurations

Standalone: The complete application exists in a single process on a single machine

Data Server/Visual Client: A server application contains purely data resources. Other visual clients, which contain no data resources can connect to the data server and display the data. Using a distributed link, 3D objects in the visual client are updated when changes occur in data resource of the data server. This situation may be useful when a single dedicated machine with a fast processor and a software license for the simulation is connected to another machine that has enhanced graphics capabilities and does not have access to the software license.

Distributed Visualisation: A network of applications connected together can share the data processing and visualisation responsibilities. Some may have data resources, some may have visualisation capabilities, and some may have both. This represents the most general application configuration.

Despite the variety of types of applications in the previous example, the only thing that distinguishes the standalone application from a data server type application is the presence of a few lines in a configuration file. Thus in principle any standalone application can become a data server, client, or include graphics simply by including and configuring the appropriate modules.

To expedite the development of applications, a custom XML Editor is being developed by Task24. Using the Reflection API, the editor can ensure that not only is the syntax for the configuration files is correct, but also that they contain well-formed and valid elements. In addition, sample template configuration files for typical usage scenarios will be provided. These can then be tailored for a specific situation.

PERFORMANCE

As is most often the case in software, especially in real time simulation software, performance is an issue. It is also often the case that there must also be a tradeoff between performance and generality. Performance then becomes more prevalent because OpenIGS is designed to be a general framework. Despite this trend towards generality and flexibility, there is also opportunity for performance optimization.

Because most of the business part of an OpenIGS application is provided by external functional software components, the primary potential source for performance bottlenecks caused by the OpenIGS Kernel occur during a Link execution, namely the Binding and Conversion facilities. We can consider two main use cases for a Link:

- high frequency data update, i.e. from a simulator variable to a visualization object
- low frequency or single shot updates, i.e. from a GUI component to some target object.

Performance is most critical particularly in the high frequency use case. In such a situation, we do not want any bind or conversion operations to be occurring. This implies then that the source and target Signal/Operation signatures must be identical. If this is indeed the case, then no intermediate processing needs to occur whatsoever. Since the Reflection API stores pointers to actual class methods and signals, these can be linked together directly resulting in almost zero overhead. But this is of course at the loss of generality, the source and target signatures *must* be identical in parameter type, count, and order in order to make the direct link.

In the low frequency situation, Links are not executed very often and it really doesn't matter if they perform within one millisecond or 100 milliseconds for example. In this case, flexibility is a priority, A GUI component should not have to provide a signal interface for every type of data and a target object should be able to receive data in its expected form.

Other situations can be imagined that lie somewhere in between these two extremes. The OpenIGS architecture can automatically identify the appropriate optimization for each of these extreme situations and situations in between. During the Link configuration, it can be recognized if there is a conversion step necessary and if binding information has been provided. If not, then a direct link can be set up. If a conversion is necessary, the conversion function can be looked up and cached ahead of time. The Binding procedure can also be cached in a similar way. In the case where performance is priority but binding or conversion is also necessary, adaptor functions and classes can be created to bypass the binding and conversion steps at the cost of having to explicitly write extra code.

CONCLUSION

OpenIGS 2 builds upon the concepts introduced by OpenIGS 1 through its redesigned implementation and the addition of new functionality. The components for Reflection, Linking, Binding, Conversion, Dynamic Loading, and External Communication provide for a highly flexible, loosely coupled, application development environment. The direct support for Signals and Composite structures automate commonly used Design Patterns. This generic infrastructure, together with the provided functional modules, editor, and template configurations provide the tools to address typical visual simulation requirements in the space industry with the flexibility to incorporate new requirements. The architecture also provides room for automatic optimisation to trade-off between performance and flexibility.

AVAILABILITY

The development of OpenIGS 2 is nearing the CDR milestone and the expected initial release will be available in Q4 of 2010. For more information about OpenIGS and about obtaining the software, please send an email to one of the contacts in the title.