OSSIE: OPEN SOURCE SCA FOR RESEARCHERS

Max Robert (Virginia Tech, Blacksburg, VA, USA; probert@vt.edu); Shereef Sayed (Virginia Tech); and Carlos Aguayo (Virginia Tech), Rekha Menon (Virginia Tech), Karthik Channak (Virginia Tech), Chris Vander Valk (Virginia Tech), Craig Neely (Virginia Tech), Tom Tsou (Virginia Tech), Jay Mandeville (Virginia Tech), Jeffrey H. Reed (Virginia Tech)

ABSTRACT

The Software-Defined Radio (SDR) research community currently needs an implementation of the SCA core framework (CF) that is open to modifications, free, simple, and in C++. Recognizing this need Virginia Tech has developed and released OSSIE (Open Source SCA Implementation::Embedded). This paper describes the underlying philosophy for the development of OSSIE, the basic structure of the released framework, shortcomings to the current implementation, available sample waveforms, and a research path for the implementation.

1. INTRODUCTION

The Software Communications Architecture (SCA) is at the core of the Joint Tactical Radio System (JTRS) family of radio systems [1], and is likely to form the core of future military and, through the efforts of organizations such as the OMG [2], the SDR Forum [3], commercial systems. One of the primary challenges for universities today is educating the graduating engineer on the fundamental choices that are required for the development of SDR. Traditional education in radio systems has focused on aspects such as classical communications, such as modulation, RF circuit design, DSP, and information theory. Graduating engineers may have a background in software development, usually in C++, but little or no background in structured programming and middleware, two crucial aspects of SDR design.

The SCA offers a powerful architecture that covers the essential aspects associated with waveform design in SDR. Given that alternative architectures are unlikely to differ much from the SCA, the SCA provides a solid foundation for students to understand SDR development. The incentive to use the SCA as an educational example lies in the fact that there is a growing need in the community for engineers that are familiar with this architecture. However, two significant problems arise from the use of the SCA in an educational environment. First, the SCA is a relatively complex specification, and a simple sample implementation can be of substantial help in increasing the level of understanding on the part of the student. Second, while the Communications Research Centre (CRC) has released a very useful open-source implementation of the SCA [4], this implementation is in Java, and most electrical engineers, the typical communications system designers, are generally not familiar with this language. Therefore, there is no simple-to-use core framework that is freely available in a language that is well known to most electrical engineers like C++.

These problems are not limited to the educational community; they extend to the research community as a whole. Researchers face many problems that are similar to those encountered by students. While a researcher may be already well aware of the SCA specifications, he may not have available a simple-to-use framework. Such a framework provides an aid to understanding to the researcher. The researcher has the opportunity to not only see how specific issues were resolved, but he also has the ability to test proof-of-concept implementations with relative ease. While Java is widely used in the computer science community, it is not prevalent in the electrical engineering community. Therefore, just like electrical engineering students, communications engineers are more likely to be familiar with C++ than Java, making the availability of such a framework in C++ an asset.

To resolve this set of problems, researchers at Virginia Tech have developed OSSIE (Open-Source SCA Implementation::Embedded). This paper describes the structure of OSSIE, code-simplification strategies that reduce the background needed by the student or researcher, such as a CORBA wrapper, and limits on the framework implementation imposed by the simplicity of the implementation.

2. DEVELOPMENT PHILOSOPHY

The target developer for OSSIE is a typical Electrical Engineering Master's student. This student is typically fresh out of the undergraduate program. In the undergraduate program such a student is likely to have had some exposure to object-oriented programming in general, especially in C++, but it is unlikely that he is familiar with advanced object oriented programming (OOP) concepts like polymorphism or to any kind of middleware, especially CORBA.

An entry-level Master's student also has some significant issues with respect to time and learning curves. A typical Master's student spends a total of 2 years in a graduate program. The first year is generally constrained to taking classes, leaving only the second year for thesis work and the remainder of the classes, typically only one or two. In that final year of the program, the student needs to find a research topic, assemble the research tools necessary to execute the research, perform the actual research, write the thesis, and defend.

Assuming an academic year, until May of year two (commencement) to finish the research. It generally takes approximately three months to write a thesis, with an additional month to defend the thesis and implement the necessary corrections to the work. Therefore, the student needs to finish his research and begin writing by January of the final year of the program at the latest. If the student begins his research in the beginning of the academic year (August), this means that the available time for assembling the tools and perform the research is between August and December, or five months.

The problem that arises in SDR research is that specifications in general, and the SCA in particular, are fairly sophisticated, requiring a fairly steep learning curve. Therefore, if OSSIE is to be used in an academic environment, ideally the student should be able to get to the point where he can perform SDR-related research on the platform within two months of beginning the work. In that span of time, not only does the student need to familiarize himself with the specifications, but he needs to familiarize himself with the existing framework.

Given these constraints, one the key attributes of OSSIE is that it must be very simple to use. Readability and simplicity are actually more important than attributes that are more critical for production implementations, such as exception handling. While good exception handling can accelerate development, it was deemed that for the first version it would add significant amounts of code that may reduce readability. To decrease the level of confusion, it is also imperative that the implementation match the specification layout as much as possible. This means that the methods and attributes included in the framework implementation should match as closely as possible the specifications. This concept extends to helper classes; if additional classes are needed that are not described in the specifications, such as XML parsers, they should be included in a separate software package (in the case of OSSIE, framework packages are released as libraries).

Furthermore, the typical Master's student is unlikely to be familiar with CORBA. The implementation should be aware of this limitation, and where possible, should isolate the developer from the idiosyncratic semantics associated with CORBA. The basic concept behind CORBA is fairly simple and can be quickly understood. However, the problem with CORBA arises from two basic problems.

First, additional steps are necessary than would at first seem unnecessary, but upon further inspection are important given the way that CORBA works. For example, passing a string is a common step that the framework needs to support. However, if one were to pass just a reference to a string, this is likely to cause problems, especially when the called method leaves scope, and the memory is deallocated. To resolve this problem, CORBA::string_dup should be used when a reference to a string is passed. Using the string duplicate, the scoping problem disappears. Implementations using CORBA are rife with problems along these lines, and it is desirable to isolate, at least initially, the beginner as much as possible from these problems.

Second, CORBA semantics can be overwhelming to the developer, yet only a handful of tasks need to be supported by CORBA, most of which can be considered "cut & paste" code. A simpler semantic structure would reduce the apparent complexity of the code, thus reducing the learning curve. It should be noted that while CORBA is an integral part of SCA version 2.2, the actual calls performed are not part of the specifications. Therefore, if the set of calls necessary to perform certain operations are collected into a smaller subset that is easier for the developer to relate to, then the implementation becomes easier to navigate while staying strictly within the bounds of the specifications.

Finally, the implementation should be fairly open, allowing for significant additions. For example, while exception handling is not included in the current version of OSSIE, it should allow the developer to add it if need be, since that may be an integral part of the student's research agenda.

One of the interesting aspects of the needs of a Master's student is that they match the needs of the typical engineer that wants to investigate aspects or specific scenarios in the SCA. The relative simplicity makes it easy for the researcher, especially one that is already familiar with the specifications, to investigate specific research topics with little regard for other aspects which may not be directly relevant. For example, if one were to investigate power management algorithms at the framework level, OSSIE allows the researcher to implement these concepts either as part of the framework by modifying the code directly or as an additional service. Finding the entry point into the software for this type of investigation should be relatively easy. This type of simple, open structure allows the researcher to invest a relatively little amount of time to increase his understanding of the topic while maintaining the freedom to add as many features to the framework as he wants.

3. IMPLEMENTATION OVERVIEW

OSSIE is an implementation that follows the SCA 2.2 specifications. While this implementation does not yet implement all the requirements in the specifications, it does attempt to follow the specifications. With the exception of Aggregate Device, the implementation contains implementations of all the classes seen in Figure 1.

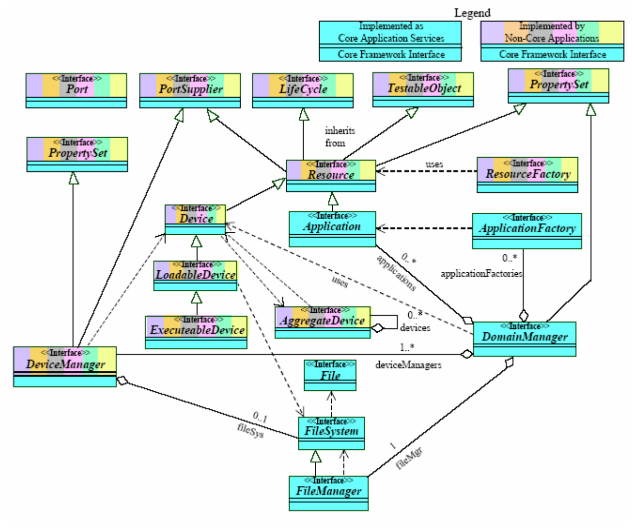
Legend
Implemented as Core Application Services
Core Framework Interface
Implemented by Non-Core Applications
Core Framework Interface
<<Interface>> Port
<<Interface>> PortSupplier
<<Interface>> LifeCycle
<<Interface>> TestableObject
<<Interface>> PropertySet
<<Interface>> PropertySet
inherits from
<<Interface>> Resource
uses
<<Interface>> ResourceFactory
<<Interface>> Device
<<Interface>> Application
<<Interface>> ApplicationFactory
0..*
applications
0..*
applicationFactories
<<Interface>> LoadableDevice
uses
<<Interface>> ExecuteableDevice
<<Interface>> AggregateDevice
0..*
devices
<<Interface>> DomainManager
<<Interface>> DeviceManager
1..*
deviceManagers
<<Interface>> File
0..1
fileSys
<<Interface>> FileSystem
1
fileMgr
<<Interface>> FileManager

Figure 1 – CF Classes

The implemented classes comprise all the relevant classes necessary to support waveforms in a wide variety of configurations. One of the key early questions that needed to be resolved was what platform to support. The first version of OSSIE was written for Windows 2000 using Visual C++ 6.0. While this operating system is neither real-time nor POSIX-compliant, it is an operating system that is widely available. Therefore, the implementation was made such that calls are made to look like POSIX through an additional layer where it was deemed necessary.

With the selection of an operating system (in the first release version) allowed the selection of an appropriate middleware. It was expected that eventually a new operating system would be used, therefore The ACE ORB (TAO) was selected as the CORBA version used in this implementation. One of the additional benefits of using TAO is that ACE is a required component. ACE (ADAPTIVE Communications Environment) provides what can best be described as an operating system abstraction. Furthermore, both ACE and TAO are open-source, thus keeping with the overall spirit of the OSSIE implementation. Furthermore, by using ACE, system calls can be made that are ACE-specific yet portable, thus increasing the flexibility of the implementation.

With the selection of an operating system also allowed for the selection of an XML parser. From a practical perspective, parsing in the SCA is broken down into two principal pieces, XML parsing and SCA-specific parsing. XML parsing involves general navigation issues in XML like identifying tags and creating a way to manage the content within these tags. SCA-specific parsing involves understanding the relevant files and tags so that the correct information from the correct file is sent to the correct component. It is impractical to develop a new XML parser since several exists today that are free and reliable. The Xerces C++ parser, available under the Apache Software License, was selected as the XML parser for the OSSIE project. The SCA-specific parsing was performed using specialized code that was written for OSSIE. The SCA-specific parser is one of two libraries in the OSSIE framework release.

3.1. Additional Operating System Support

Windows is limited in its ability to support SDR applications, primarily in terms of access to some low-level system functionality, and as mentioned above, it was expected from early on in the project that alternate platforms would eventually need to be supported. Therefore, OSSIE was ported to Linux (Fedora with kernel version 2.6). Furthermore, to extend the support within Windows, project files that are used with VC++ 7.1 were included in this port. Version B, released in October of 2004, supports Windows 2000/XP and Linux. It should be noted that, while some OS-specific functionality was included in the framework through the use of preprocessor directives (i.e.: #ifdef), functionality that could be implemented through ACE was implemented that way, thus increasing the portability of the code.

4. RELEASE STRUCTURE

As mentioned before, the OSSIE implementation is released as a set of two libraries, a parsing library and a framework library. The parsing library contains the SCA-specific parsing calls. A structure similar to that used in the CRC's SCA Reference Implementation (SCARI) [Ref:SCARI] was used for the parsers used in OSSIE. It was found that the approach used by the CRC team is an efficient and clean way of achieving the parsing goals.

The library containing the SCA-specific parser was also selected as the site for locating the only other class used in OSSIE as a helper class, the ORB wrapper. The ORB wrapper will be discussed in more detail in the Code Simplification Strategies section of this paper.

The second library released under OSSIE is the Core Framework library. This library contains an implementation of all the CF classes, except Aggregate Device, which was considered unnecessary for the scale of projects that have been attempted to date. It should be noted that later versions are likely to include the Aggregate Device class, since research at Virginia Tech is expected to move in the direction of multi-processor boards.

The CF library includes the core application services as well as pieces of the non-core applications. The non-core applications include classes such as Device and Resource, which are implementation-specific. However, there are some common pieces exist in these different classes, and to minimize the amount of work required on the part of the developer, these pieces were implemented. The virtual methods that cannot be populated because the specifics of the application are unknown, then they were implemented as empty methods. The use of empty methods means that the developer implements the required functionality only if needed to investigate the specific behavior, otherwise it is left empty and the resulting code still compiles.

An example of this selective population of methods is the Resource class. The Resource class needs to implement just four methods: the constructor, start(), stop(), and identifier(). The constructor resource just associates a UUID (Universal Unique Identifier) and the name with the Resource. Therefore, if the developer provides a non-NULL UUID and a non-NULL name, the Resource's UUID and name are set to the given values. If the developer does not provide this information into the constructor, then the Resource reads the information from the configuration file; this behavior is standard and a developer is unlikely to want to re-define it, so there was no sense in not including it in the implementation. The same concept applies to the method identifier(), where the identifier value is returned. The methods start() and stop(), however, are implementation-specific. The developer is expected to overload these methods in the implementation with the appropriate code so that the start and stop commands implement the desired functionality.

In order to reduce the amount of code that is directly visible to the developer, OSSIE is released as a set of libraries. A shared library provides significant benefits in the management of code, since it can significantly reduce the amount of visible code that the developer needs to deal with. In order to reduce the footprint of the implementation, dynamic libraries were used. With static libraries, the whole library would have been included in each component, thus leading to large executable sizes. The use of dynamic libraries means that the component loads only the code that is necessary to execute the required functions, thus significantly reducing the required memory. While OSSIE is not designed with a small footprint in mind, such an approach was considered a no-cost improvement on the implementation.

5. CODING STRATEGIES AND SHORTCUTS

As mentioned above, an ORB wrapper was implemented in OSSIE. The goal of the ORB wrapper is to reduce the amount of exposure that a developer has to the CORBA interface. While the ORB wrapper is a work in progress and is expected to further isolate the developer from CORBA, there are some calls that it now contains that perform some tasks. Sample methods in the ORB wrapper class include: lookup (get an object reference by name), bindobj (bind a name to an object reference), and getNamingContext (return the current naming context used).

The ORB wrapper coupled with a project-wide ORB reference means that the developer is isolated from some of the CORBA interface. It should be noted that the developer still needs to perform actions such as narrow the object reference and activate the object, but where possible those will be abstracted in future versions.

3.1. Limits on Implementation

The current implementation of OSSIE follows a basic philosophy of simplicity and readability. To achieve this goal, the implementation is missing some pieces that are considered important in other types of implementation, such as commercial implementations. The two principal aspects that were not implemented in this version are the Aggregate Device class and exception handling.

Aggregate Devices were considered unnecessary given the types of implementations that the OSSIE is intended to support. However, it should be noted that Aggregate

Device is considered a non-core application component. Therefore, it should be fairly straightforward for a developer to add this functionality into the framework. Given the progress of ongoing research at Virginia Tech based on OSSIE, it is likely that this class will be added in later versions.

The other missing aspect of the implementation, exception handling, is not as straightforward or as easy to implement. The SCA specifications outline what exceptions to catch, but leaves the developer to determine what to do when exceptions are caught. In general, the path that the point to which the framework should go to when an exception occurs is fairly obvious. While this is the case in most instances, it is not consistently so.

Even though the framework is intended for relatively inexperienced developers, it was deemed that the developer should aware of what he is supposed to implement, and experiments would be executed in a controlled environment. Given these base assumptions, the OSSIE implementation catches some, though not all, exceptions, but performs no exception handling. Thus, if an exception is thrown program execution halts.

It should be noted that the addition of exception handling, while laborious, would not be conceptually difficult. Furthermore, the elimination of exception management from the framework implementation has lead to significant reductions in the amount of code.

6. SAMPLE WAVEFORMS

One of the crucial aspects in the release of OSSIE is the application or set of applications that are provided as sample implementations. These samples should be sufficiently simple for the beginner to understand, but sufficiently sophisticated for the examples to be meaningful.

Version A of OSSIE, released in July of 2004, included a sample waveform that followed the structure seen in Figure 2.

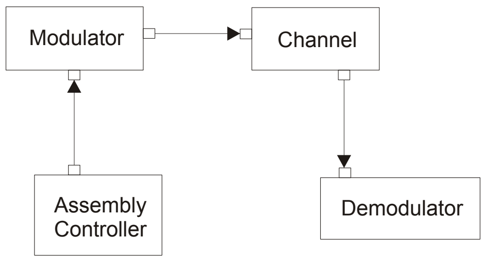

Figure 2 – Diagram of sample application

For the sake of simplicity, the Port and Resource functionality was combined into a single class in this application, as seen in Figure 3.

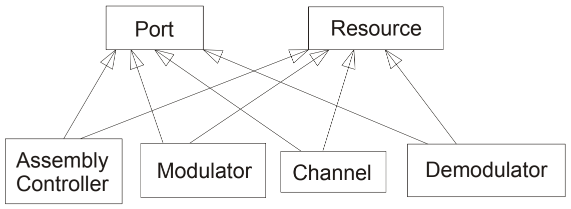

Figure 3 – Inheritance for components in sample application

The reason for this choice is that this approach simplifies the implementation. Conceptually, using this approach, the GetPort() call to PortSupplier returns a pointer to the Resource itself. This is a relatively simple concept, and it was considered to be a good starting point for an entry-level developer. However, there are some issues drawbacks associated with this approach, which are best illustrated by investigating alternate ways to implement components.

The above example is not the only way to implement a waveform; Port and Resource can be implemented as separate classes, as is implied in the specifications. If Port and Resource were implemented as separate classes in this application, then it would increase the complexity of the sample waveform. To basic approaches could have been implemented, a single-thread case and a multi-thread case. In the case of the single thread, the Resource class would have created an instance of the Port within the Resource. In this approach, the two classes would have been kept separate. Functionally speaking, this approach would have yielded the same type of behavior.

In the case of a multi-threaded implementation, the Port would be instantiated as a separate thread of execution. This approach has significant benefits. The primary benefit is that separate event reactors could have been created. In implementations using CORBA clients, the simplest implementation is to place the client in a blocking loop that waits for CORBA events to arrive. When a CORBA event is serviced and the loop returned to a blocked wait. Generally, this blocking call is combined with a timeout to allow for the Resource to gracefully exit execution. If the Resource were to need to service a separate event source, like a GUI, then a separate loop is necessary to service those events. If the process has a single thread, then it is geared for the management of a single source of events. Of course, this is not necessary if non-blocking calls are used, but those types of implementation can be more complicated and for the purposes of this sample are assumed to be outside the

scope of the developer. For the process to be able to manage multiple types of events that are services through different blocking calls, then separate threads of execution are necessary.
The basic signal that was passed between the components in the simulated link is a BPSK signal with an arbitrary signal bandwidth. The nature of the components, inheriting from both Port and Resource, and operating within a single thread means that the whole waveform behaved as though all components operated under a single thread. The best way to understand the concept is to step through the signal flow. The modulator generates a symbol stream, in this case 1000 symbols. The modulator component now pushes the information onto the channel component. For the reader that is not familiar with CORBA, this is essentially like calling a public method on an object. Since the object making the call is essentially calling a method on another object, execution does not return to the calling object until the called method returns.

This process is cascaded from object to object in the application. In this case, the channel object then pushed the data to the demodulator object using semantics that look just like calling a method on an object. The channel object's execution is now blocked until the execution of the demodulator object's called method is complete. Thus, we have a cascade effect; the modulator object cannot continue until the channel object is complete, and the channel object cannot complete until the demodulator execution is complete. Hence, the modulator execution is blocked until the demodulator execution is complete. This effect is common in distributed processing, and sometimes it is a desired effect, such as when all execution is dependent on a single system component or resource that cannot be concurrently shared. However, in the case of a radio system, this is a program flow that is not desirable, since system resources are likely to be unnecessarily idle, thus leading to a (possibly) suboptimal use of system resources. Ideally, the thread of execution should be non-blocking and concurrent. However, the initial example presented to the beginning SCA developer had to strike a balance between functionality and simplicity. In order to strike this balance, the first released sample version followed the simple program flow described above.

One of the problems encountered in the development was to find a good way of presenting data to the developer such that it had some visual impact. Given the cross-platform nature of OSSIE, a platform-specific graphical environment was deemed an inadequate solution. Therefore, for the Version A release, MATLAB was selected for graphing the received information. To implement this, the Demodulator component was developed using the C libraries provided by the MATLAB 6.5 release 13. When the waveform is installed, the Demodulator component begins the MATLAB process automatically. When the demodulator component receives the data signal from the channel component, it is then passed to the MATLAB environment, and the Demodulator component executes a series of commands to graph the data onto a window. Not all developers have access to MATLAB. To resolve this issue, Version B of OSSIE, released in October of 2004, provides the developer with the option of using MATLAB or wxWindows [3], a cross-platform graphing library. It was decided to keep the MATLAB option because it provides the developer with an example of how to interface MATLAB and the SCA. The combination of the SCA with MATLAB provides the developer with a powerful platform that allows rapid prototyping.

7. THE DEVELOPMENT PATH FOR OSSIE

The OSSIE development team considers the development of OSSIE to be an asset to the Virginia Tech wireless research community as well as the SDR research and development community as a whole, and hence is dedicated to the vision of an open-source C++ implementation of the SCA CF that is true to the initial release philosophy. Planned improvements for the future include a more complete framework as well as more advanced waveforms. The eventual goal is to receive certification for OSSIE under JTEL, and thus enhancements and corrections are expected on the framework as time progresses. Given that OSSIE is a research platform, additions to the framework are expected such as power management. Waveforms that are expected for future releases are ones that support concurrent processing, waveforms that integrate test equipment into the SCA, SCA 3.0, and other outgrowths from ongoing research. Visit http://www.mprg.org/research/ossie/ to download OSSIE.

8. ACKNOWLEDGEMENT

The first year of the OSSIE development effort has been largely unpaid, volunteer effort by a group of dedicated graduate and undergraduate students, all of whom are listed in the authors list for this paper. Direct and/or indirect funding for this effort was received for some members of the team through the DCI Postdoctoral Research Fellowship, the Office of Naval Research, Tektronix, and the MPRG Affiliates Program.

9. REFERENCES

[1] – http://jtrs.army.mil

[2] – http://sbc.omg.org/
[3] – http://www.sdrforum.org
[4] – http://www.freiburg.linux.de/~wxxt/