- SCA ADVANCED FEATURES - OPTIMIZING BOOT TIME, MEMORY USAGE, AND MIDDLEWARE COMMUNICATIONS

Steve Bernier (CRC Ottawa, Canada; steve.bernier@crc.gc.ca);
Charles Auger (CRC Ottawa, Canada; charles.auger@crc.gc.ca);
Juan Pablo Zamora Zapata (CRC Ottawa, Canada; juan.zamora@crc.gc.ca);
Hugues Latour (CRC Ottawa, Canada; hugues.latour@crc.gc.ca);
Mathieu Michaud-Rancourt (CRC Ottawa, Canada; mmrancou@crc.gc.ca)
Communications Research Centre Canada (CRC)
Ottawa, Ontario, Government of Canada.

ABSTRACT

This paper describes a list of advanced features that can be used to optimize boot time, memory footprint and communication speed for SCA Radios. The paper first describes the most common performance issues and then presents a number of advanced SCA features that can be used to address each issue. Each feature is discussed in details to exhibit under which condition they can best perform. Performance metrics are presented for each feature. Finally, the paper concludes with an outlook on the next wave of advanced optimization features.

1. INTRODUCTION

The Joint Tactical Radio Systems (JTRS) is a US DoD program. Its objective is to provide the DoD with needed communication capabilities through a family of affordable, interoperable radios. The JTRS radio sets are Software Defined Radios (SDRs) that can be used to quickly field new communications capabilities. The JTRS program achieves its goal by maximizing the reusability of common software and hardware for different radios.

The software implemented for JTR radio sets must comply with the Software Communications Architecture (SCA) specification [1]. The SCA specification describes a Core Framework (CF) which provides the infrastructure to allow software components to plug-and-play. The SCA Core Framework specification also describes set of rules and APIs that must be used by developers of SCA-compliant software components.

The term “SCA Radio” is often used as a synonym for “Software Defined Radio”. However, the later have been around for much longer than SCA Radios. Software has been used in communication devices for decades. SCA Radios only started to be deployed a few years ago. However, due to more stringent portability and interoperability requirements, SCA radios generally contain much more software than previous generations of SDRs. To be more specific, SCA radios use some software to process the input/output signals. Most SCA Radios contain several million lines of source code. And as a result, SCA radios generally take longer to boot and use more memory than older SDRs.

The first generation of SCA Radios was considerably slower to boot and required a large amount of runtime memory. Unfortunately, that generation of SCA radios is at the source of most of the bad press the SCA has suffered from. However, many years of research and development have enabled the SCA community to address those issues and deploy SCA radios that meet the most stringent requirements [2, 3, 4, 5].

This paper describes several optimizations and system design approaches that have been proven to address some of the most serious performance issues of SCA radios. This paper introduces novel features and revisits features introduced in previous papers [6, 7, 8]. Section 1 of this paper addresses boot time optimizations, section 2 discusses memory footprint optimizations and section 3 presents communication speed optimizations. Section 4 takes a look at the future of SCA Core Frameworks and section 5 concludes the paper

2. BOOT TIME OPTIMIZATIONS

The SCA relies on software components that must be able to plug’n’play with each other. In addition, the SCA supports plug’n’play of components that might run on different processing devices. SCA components are very modular and

are described using a number of different files containing meta-data. An SCA waveform application is essentially an assembly of components. Assembly descriptors identify which components need to be instantiated, how they need to be interconnected, and what should be the default values for each of their configurable properties. All that assembly information is stored in a number of files. The same is true for node assemblies, which describe Devices and Services that need to be launched upon power-up.

In order to run the different components of an assembly, the SCA CF must read a number of files and parse the metadata they contain. For each component to be deployed, it must also download the binary code to the selected target Device for execution. This means reading, transferring, and writing many files.

The above discussion shows that the SCA is file centric; it relies on the use of a file system for flexibility and modularity purposes. But the fact that file system access is typically slow on embedded systems has the consequence of introducing delays that can significantly affect the boot time of a platform. The sub-sections below present some of the most common issues along with optimizations that can help minimize their effects.

2.1 File System Integrity

Given the fact that the SCA file system contains critical information, the integrity of it is fundamental to the proper functioning of an SCA radio. There are many ways to ensure file system integrity. One common technique is to perform an integrity check every time the SCA Radio is powered. Depending on the type and size of file system being used, this check can take anywhere from a few seconds to over a minute.

One way to minimize the impact of a file system check on the boot time is to make most of the file system read-only. The integrity check only needs to be performed on the portion of the file system that is being used for writing. The SCA Core Framework needs very little storage for writing files as it mostly reads them. Only the SCA Devices being used as targets for deployment of software components need significant storage for writing. And this can actually be alleviated with a caching feature (see section *2.2.4 ExecutableDevice File Caching*).

The choice of the file system driver has also been proven to make a substantial difference. Some file systems have been able to perform in a few seconds what other file systems take tens of seconds. Another way of avoiding a lengthy file system check is to use a journaling file system, which safeguards the integrity in real-time as files are being written and deleted.

2.2 Native File Access

Most embedded systems use flash memory which can be very slow compared to run-time memory. SCA Radios are no exception. It is therefore crucial to use fast flash memory. But in any case, using a Core Framework that is capable of avoiding or optimizing file access will directly translate into time savings for Radio boot up.

2.2.1 DomainManager Native File Access

The DomainManager accepts registration of all DeviceManagers. A DeviceManager is one of the first SCA components to be started upon power up. In fact, a single SCA radio might start several DeviceManagers to take care of several boards. The DeviceManager does a fair amount of file reading to learn which components (SCA Devices and Services) need to be launched. Unless the DeviceManager is hard-coded to launch all the right components and configure them appropriately, there is no way to avoid reading the node assembly metadata files.

Once the DeviceManager has read the metadata and launched the SCA Devices and Services, it must register with a DomainManager. The DomainManager needs to collect the full list of Devices and Services registered by all DeviceManagers running in the SCA radio. This is necessary for application deployment.

However, when the DeviceManager registers with a DomainManager, because of the lack of a standard API, it cannot provide all the necessary deployment information about its Devices and Services. Therefore, the DomainManager must download and read the Metadata for each registered SCA Device.

When the registering DeviceManager is running on the same processor as the DomainManager, the metadata for Devices and Services is almost assured to be accessible natively by the DomainManager. Using native file access, the DomainManager can read up to 40% faster, which can translate into the saving of several seconds. **Table 1** shows a performance comparison regarding how much time it takes to read a file natively compared to via the SCA file system layer. The tests were executed in two different operating environments as illustrated in **Table 1**.

Table 1 - File Access Time Metrics

Test System	File Size	SCA File System	Native File System	Gain
Linux, Pentium 3GHz	4MB	355ms	20ms	94%
INTEGRITY, PPC 405, 400 MHz	1.5MB	2500ms	1500ms	40%

2.2.2 DeviceManager Optimized Registration

When a registering DeviceManager is running on a different processor from the DomainManager, native file access acceleration can't be used. One workaround is to actually add a specialized registration API on the DomainManager, which can be used by a DeviceManager to provide deployment information (gathered by the DeviceManager during the node boot) regarding the registering Devices and Services.

The regular registration process for an SCA Device uses a minimum of 19 CORBA calls. Some of those calls are very slow as they are used to copy metadata files. This optimized registration feature can provide significant performance gains and speed up the boot up sequence by avoiding the majority of those calls. **Table 2** shows metrics comparing the performance of optimized registration versus standard registration. The tests used to produce those metrics employed two different operating environments with variations of 1 Device and 4 Devices in each. The tests were used to measure how long it took the DeviceManager to register its Devices with a DomainManager running on a different processor.

Table 2 - DeviceManager Registration Time Metrics

Configuration	Standard Registration	Optimized Registration	Gain
Linux Pentium, 2 GHz, 1 Device	0.56 sec	0.19 sec	66%
Linux Pentium, 2 GHz, 4 Devices	1.53 sec	0.24 sec	84%
LynxOS PPC 405, 400 MHz, 1 Device	0.86 sec	0.13 sec	85%
LynxOS PPC 405, 400 MHz, 4 Devices	2.33 sec	0.22 sec	91%

2.2.3 ExecutableDevice Native file access

Since the SCA is a distributed system, it is assumed that SCA components need to be downloaded to target SCA Devices (i.e. a LoadableDevice or an ExecutableDevice). However, target SCA Devices can also be implemented to determine if the files being downloaded are natively accessible. When that is the case, the deployment of an application can benefit from the performance improvements described in table **Table 1**.

Ultimately, all file copies, even native files copies can be avoided by allowing target SCA Devices to natively access the files where they have originally been installed. With this feature, time consuming copies can be avoided. Saving copies of each component artefact can be very significant; the binary code alone can be several megabytes. But this feature may not be useful if the file system where the files are located is set to read only.

2.2.4 ExecutableDevice File Caching

Another way to avoid file copies is to make target SCA Devices use a file cache. Target Devices can locally save the files they are sent. With this feature, a copy is only made the first time a same file is sent to a same target Device, which can be useful for SCA radios that always run the same applications, deployed using the same target Devices.

Such a feature requires that target SCA Devices have access to a permanent file system (e.g. flash). Since, the first deployment usually happens during manufacturing, fielded radios benefit from this feature. For SCA Devices using volatile memory, benefits are obtained when the same application is repeatedly launched/shutdown without powering down the Radio. **Table 1** provides estimates of the time it takes to copy files and thus how much time can be saved with this caching feature, which is quite significant given the fact that the deployment of an SCA application typically involves the copy of several binaries. This feature can also be used in combination with native file access.

3. MEMORY FOOTPRINT OPTIMIZATIONS

3.1 XML Parsing

All of the SCA metadata which contains information about SCA software components and assemblies is described in XML. CRC measured how much time and dynamic memory is required to parse the metadata information using three different approaches. The first approach uses a standard COTS parser called Xerces-C++ [9]. The second approach consists in reading a digested XML format instead of the standard text format. The third approach consists in using a specialized parser that can read the standard SCA XML files. The tests were executed on a Linux Desktop 3 GHz Pentium processor using a metadata file containing the description of 50 component properties. The results are shown in **Table 3** and described in the next three sections. Each test was executed 10 times and the metrics were averaged out.

Table 3 - Parsers Performance Metrics

Parsers	Static Memory	Dynamic Memory	Parsing Speed
Xerces-C++	3,000 KB	66 KB	6.7 ms
Digested	300 KB	8 KB	1.1 ms
Specialized	420 KB	10 KB	1.7 ms

3.1.1 Xerces-C++ Parsers

Xerces-C++ is a very popular open-source XML parser that has been compiled for a few embedded system operating environments, but it is mainly used for desktop applications. CRC used Xerces-C++ and implemented a number of parsers to read the different XML files of the SCA. Xerces-C++ is available as a library of several mega bytes in size.

The SCA parsers were built using static linking which required 2.6MB of the Xerces-C++ library. The total static size of the parsers was of 3MB. **Table 3** presents the performance metrics for execution of the parsers.

3.1.2 Digested XML Parsers

A digested XML parser is a parser that reads XML files in an intermediate binary format. The idea behind this approach is that computers can process digested binary files much faster than plain text files that need to be parsed character by character.

The digested file format is produced off-line by reading the standard XML and saving only the relevant information in a binary format. CRC created a few variations of the binary format along with binary readers that could read the same XML files as the Xerces-C++ parsers. The advantage of this approach is performance, but the disadvantage is the proprietary file formats used for the digested information.

The total static footprint needed for these parsers is 300 KB which represents a major improvement over the parsers based on Xerces-C++. As shown in **Table 3**, the digested XML parsers also used much less runtime memory and performed much faster.

3.1.3 Specialized XML Parsers

A specialized XML parser is a hand-crafted parser that knows specifically how to read the SCA XML files. The CRC specialized parsers are based on state machines inferred from the SCA XML file formats. The state machines have been optimized and only the relevant information is parsed.

The total static footprint for the specialized XML parsers is of 420 KB which is still a very impressive improvement over the parsers based on Xerces-C++. As show in **Table 3**, this approach is virtually as good as the digested XML parser approach without relying on a proprietary file format.

3.2 Address Space Collocation

The SCA specification employs many concepts from the Component Based Development (CBD) paradigm [10, 11], which promotes reusability and adds flexibility to system designs by creating applications from an aggregation of independent components. For an SCA application, this translates into multiple Resources, each performing a subset of the waveform's functionality (e.g. FFT, encoding, filtering).

While having many advantages at the design level, this approach, compared to having a single Resource that implements the full functionality of an application, may increase the static footprint of the application, especially in environments that do not support shared libraries. This is due to the basic OS/CORBA/SCA functionality that each Resource must have, independently of their application-specific functionality. For example, a typical size for a fully implemented, statically linked, SCA Resource that uses several properties and ports and implements complex signal processing algorithms is 1 MB. Approximatly 50% of that footprint (500 KB) is associated to the fixed-cost infrastructure for a component.

In memory-constrained environments, one way to alleviate the cost of extra footprint for multiple components is to use the concept of ResourceFactory. An SCA ResourceFactory can be used to launch many Resources inside a single address space. Therefore, at the OS level, instead of having one process (i.e. one address space) for each Resource, a single process is created for the ResourceFactory which then creates different tasks (threads) for the different Resources to be launched. This allows the fixed-cost for basic OS/CORBA/SCA functionality to be paid only once for the address space, and re-used by the different tasks. And logically, the overall footprint improvement increases proportionally to the number of Resources being created inside a ResourceFactory rather than as separate address spaces. For example, using the above numbers of 1 MB per Resource, 50% of which is the fixed cost, an application composed of ten Resources may save approximately 5 MB of static footprint when launched by a ResourceFactory.

Furthermore, using appropriate SCA tools, 100% of the source code for a ResourceFactory can be generated and does not require any modification to previously developed components for them to run into a ResourceFactory address space.

The concept of co-localizing multiple components in a single address space can also be applied to other SCA entities. For example, on single node systems, the DomainManager and DeviceManager can be co-located in the same address space, yielding similar improvements (25% for two components) as for Resources. Note that co-localizing components inside a same address space may also yield savings in terms of dynamic memory (e.g. heap) usage. Co-location can also be used for SCA Devices and services.

4. COMMUNICATION SPEED OPTIMIZATIONS

4.1 The Need for Middleware

The SCA relies on software components that must be able to plug'n'play with each other. Software components built from third party organizations may be implemented using different programming languages. Interoperability between software components can't rely on source code artefacts (e.g. header files or libraries), it must be achieved through middleware. For that purpose, the SCA has chosen the well-established standard for middleware called CORBA. The use of middleware is at the heart of the component based development paradigm. This fairly recent programming paradigm is widely used in other markets and tries to mimic the hardware component paradigm.

The hardware component industry has been very successful at producing reusable components. Hardware components can easily be assembled together even if they come from different manufacturers. And this can be done without the implicit knowledge of how components have been created. The assembly of components only relies on their behavior specification (e.g. pinout, protocol) and their requirements (e.g. voltage). The software component paradigm strives to achieve the same success with software.

CORBA allows components to be connected to each other during runtime. Connections are not achieved using source code, a compiler, and a linker. SCA components are independent of one another and they provide interfaces used through connections. This is akin to hardware components connected by traces between pins on a printed circuit board. CORBA has been used for real time embedded systems for a long time. During the late 1990s, CORBA evolved and became more appropriate for smaller and faster embedded systems. One key CORBA feature for realtime systems is the use of specialized "pluggable transports" [12, 13]. This feature allows CORBA objects (e.g. SCA Components) to communicate using specialized, fast, communication transports. Using pluggable transports can yield impressive performance [7, 14].

Unfortunately, early SCA radios did not make use of pluggable transports; they relied on standard TCP/IP as a transport for CORBA. The use of TCP/IP is fine for communications between components located across the Internet, but it is very slow for communications between components within a same embedded platform.

4.1 CORBA Performances

The performance of CORBA in the context of the SCA is nearly absent from the research community literature. It has only been covered by a few papers [7, 14, 15, 16]. The goal of this section of the paper is to provide one more contribution. CRC has created a simple SCA application to measure how long it takes to transfer a buffer of data between SCA components. The application is called "PerformanceApp" and is made of three SCA Resource components. The first Resource is called the PerformanceAnalyser. It is used to produce variable size sequences of octets or doubles and to send the sequences to another Resource. Each sequence produced is time-stamped just before being sent. The PerformanceAnalyser can also receive the time-stamped sequences and compute the time it took for a sequence to travel back to the PerformanceAnalyser.

The second SCA Resource is called PassThrough. This Resource receives data sequences and simply retransmits them. In the application used for this paper, the PassThrough Resource retransmits all sequences back to the PerformanceAnalyser. The third and last component of the application is the AssemblyController Resource used to coordinate the application functionality.

CRC executed the PerformanceApp in three different application deployment configurations and measured the average time it takes for a data sequence to travel from the PerformanceAnalyser component to the PassThrough component and back. For each deployment configuration, two types of data sequence were transmitted: sequence of octets (8 bits) and sequences of doubles (64 bits). In addition, each type of sequence was transmitted in two different sizes: 1024 or 2048 elements. For each test, a thousand sequences were transmitted. Each test scenario was executed three times and the measurements were averaged.

The first configuration had both the PerformanceAnalyser and the PassThrough components deployed on the same processor. In this configuration, the

transport used was TCP/IP. The tests were performed using two different Operating Environments (OE) running the SCARI++ GT Core Framework and ORBexpress RT. The first OE consisted of a laptop running Linux, using an Intel Centrino processor clocked at 2 Ghz. The second OE consisted of a single board computer running the Green Hills INTEGRITY operating system on a AMCC 405GPr PowerPC processor clocked at 400 Mhz. **Table 4** and **Table 5** present the average round-trip times for each test in micro-seconds.

Table 4 – Intel/X86 OE Round Trip Metrics using TCP/IP

Sequence type:	**Double**		**Octet**	
Number of Elements:	1024	2048	1024	2048
Average round trip time in usecs:	80	129	66	74

Table 5 – INTEGRITY/PPC OE Round Trip Metrics using TCP/IP

Sequence type:	**Double**		**Octet**	
Number of Elements:	1024	2048	1024	2048
Average round trip time in usecs:	3334	7272	1428	1767

The second configuration is exactly the same as the first configuration with the INTEGRITY/PPC OE, except for the CORBA transport that was used. In this configuration, TCP/IP was replaced by a pluggable transport called INTCONN. The ORBexpress RT ORB makes use of a this high-speed proprietary transport available for the INTEGRITY RTOS. It is important to note that this test was done without changing any source code in any of the SCA components. The test was conducted by telling the CORBA middleware to first use INTCONN if possible. And this configuration can actually be done at the command line for properly implemented SCA components. Table **Table 6** presents the average round-trip time for each test in micro-seconds and the percentage of improvement over the first test configuration.

Table 6 - INTEGRITY/PPC OE Round Trip Metrics using INTCONN

Sequence type:	**Double**		**Octet**	
Number of Elements:	1024	2048	1024	2048
Average round trip time in usecs:	2215	4768	1042	1273
% Improvement over TCP/IP	33.56	34.43	27.02	27.98

The third configuration had both the PerformanceAnalyser and the PassThrough components deployed on the same processor and in the same address space. It was tested with the same PowerPC OE used in the previous configurations. This last configuration relies on the support of ResourceFactory by an SCA Core Framework. In this case, with proper SCA modeling tools, an SCA ResourceFactory can be automatically generated to allow the instantiation of pre-existing SCA Resource into a single address space. When both CORBA objects (e.g. the PerformanceAnalyser and the PassThrough) are located in a same address space, clever ORBexpress middleware can bypass any transports and use function invocations for communication between objects. And as for the previous pluggable transport scenario, this CORBA enhancement is completely transparent to users of the transport. This configuration was tested using the same two OEs as for the first configuration. As you can see, **Table 7** shows a tremendous performance improvement over the previous configurations.

Table 7 - INTEGRITY/PPC OE Round Trip Metrics using a ResourceFactory

Sequence type:	**Double**		**Octet**	
Number of Elements:	1024	2048	1024	2048
Average round trip time in usecs:	244	492	155	231
% Improvement over TCP/IP	92.7	93.2	89.2	86.9
% Improvement over INTCONN	89	89.7	85.1	81.2

5. FUTURE SCA CORE FRAMEWORKS

The next generation of SCA Core Frameworks will provide even more optimizations features. New features will most likely provide static deployment optimizations which consist in trying to avoid performing the same task more than once during the deployment of components [3]. Both the ExecutableDevice file caching feature and the optimized registration feature described in this paper are static deployment optimizations. Static deployment features can be used to get a deterministic behavior from an SCA Core Framework. Determinism is a crucial first step towards certifications such DO-178B and Evaluation Assurance Level (EAL).

In the future, CRC will work on features such as source code generation of a DeviceManager and an ApplicationFactory. These two components embody the component deployment engine of the SCA and therefore read large quantities of metadata files to dynamically find suitable target SCA Devices for the deployment of different SCA components. For some platforms, the dynamic decision making that takes place to choose SCA target Devices is not required. Being able to generate a DeviceManager and a ApplicationFactory that follow a fix deployment strategy will help further reduce footprint requirements and boot up times. as well as obtaining a more deterministic behavior.

CRC's goal is to be able to offer radio manufacturers the possibility of modeling and implementing their components for a standard SCA Core Framework and to use a modeling tool to generate the missing pieces for static deployment when needed. This approach most likely won't require any changes in the source code of existing SCA components. Standard SCA component assemblies will be able to benefit from these new static deployment optimizations without having to be modified. It will be very similar to what can already be done for applications with a ResourceFactory.

6. CONCLUSION

This paper started by explaining that in order to support the plug'n'play of reusable software components, the SCA relies on large quantities of files. It was also established that the boot up time of an SCA radio is directly related to the speed of its file system. The choice of a file system technology is amongst the most important system design decisions for performances.

And since the SCA is very file-intensive, the paper introduced a number of features used to optimize file access. It was explained that reading a file natively is faster than reading through the SCA file system layer. Performance metrics describing the speed of file access for typical configurations were presented. It was also made clear that the best possible way to shorten the boot up time is to avoid reading files whenever possible. Features that help avoid reading files like ExecutableDevice file caching and the optimized DeviceManager registration were presented along with performance metrics.

The paper also described a number of features used to reduce the memory footprint requirements of an SCA operating environment. It was shown that the choice of the XML parser can make a huge difference in static and runtime memory requirements. Specialized XML parsers uses as little as 15% of the dynamic memory required for the most popular COTS XML parser and the same is true for static memory. This can easily translate into megabytes of memory savings. The paper also introduced techniques transparent to the SCA component developer that can save significant amount of memory by exploiting the concept of address space collocation.

The paper also covered one aspect of middleware performances. The use of pluggable transports to get better performances was described. Metrics for different deployment configurations clearly demonstrated that CORBA performances can be made more than acceptable with a minimum of SCA expertise. When CORBA middleware is given the opportunity to dynamically choose the most appropriate communication path, significant performance gains can be obtained. And for performance, the ultimate SCA deployment configuration involves the use of a ResourceFactory.

Finally, the paper introduced the concept of static deployment optimizations and explained that it is expected that those features can provide more deterministic behavior and further reduce static and dynamic memory requirements.

10. REFERENCES

[1]] JTRS Standards, http://sca.jpeojtrs.mil/.

[2] Bernier S., "Evolution of the SCA", *Proceedings of the International Software Radio Conference*, London, June 2007.

[3] Bernier S., Bélisle C., "Taking the SCA to New Frontiers", *Proceedings of the SDR'06 Technical Conference*, November 2006

[4] S. Leblanc, "Case Study: SCA Software-Defined Radio, The AN/GRC-245 Radio", *International Software Radio Conference*, June 2009, London, UK.

[5] Mark Turner, "Harris SDR Solutions – Scalable, Reusable, and Secure", *International Software Radio*, London, UK, June, 2004.

[6] Bernier S., Zamora Zapata JP., "The Deployment of Software Components Into Heterogeneous SCA Platforms", *Proceedings of the SDR'08 Technical Conference*, October 2008.

[7] J. Belzile, "Putting it all together – Objectives and Challenges", *SDRF'05 Technical Conference*, 2005.

[8] C. Linn, "Designing JTRS Core Frameworks For Battery-Powered Platforms: 10 Techniques For Success", *SDR'04 Technical Conference*, 2004

[9] Xerces C++ Parser, http://xml.apache.org/Xerces-C++/

[10] B.J. Cox, "Planning the Software Industrial Revolution", *IEEE Software magazine*, November, 1990.

[11] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Professional, Boston, 2002.

[12] C. Hrustich, "CORBA for Real-Time, High Performance and Embedded Systems", *Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, isorc, p. 345, 2001.

[13] D.C. Schmidt, C. O'Ryan, O. Othman, F. Kuhns, J. Parsons, "Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware", *Design Patterns in Communications*, Cambridge University Press, 2001.

[14]G. Middioni, "CORBA over VMEbus Transport for Software Defined Radios", www.motorola.com, 2005.

[15] P. Balister, M. Robert, J. Reed, "Impact of the use of CORBA for Inter-Component Communication in SCA Based Radio", *SDR'06 Technical Conference*, Orlando, Fl, November, 2006.

[16] Sarvpreet Singh, M. Adrat, S. Couturier, M. Antweiler, M. Phisel, S. Bernier, "SCA-Based Implementation of Stanag 4285 in a Joint Effort Under the NATO RTO/IST Panel". *SDR'08 Technical Conference*, 2008

Copyright Transfer Agreement: The following Copyright Transfer Agreement must be included on the cover sheet for the paper (either email or fax)—not on the paper itself.

"The authors represent that the work is original and they are the author or authors of the work, except for material quoted and referenced as text passages. Authors acknowledge that they are willing to transfer the copyright of the abstract and the completed paper to the SDR Forum for purposes of publication in the SDR Forum Conference Proceedings, on associated CD ROMS, on SDR Forum Web pages, and compilations and derivative works related to this conference, should the paper be accepted for the conference. Authors are permitted to reproduce their work, and to reuse material in whole or in part from their work; for derivative works, however, such authors may not grant third party requests for reprints or republishing."

Government employees whose work is not subject to copyright should so certify. For work performed under a U.S. Government contract, the U.S. Government has royalty-free permission to reproduce the author's work for official U.S. Government purposes.