

# EXPERIENCES IMPLEMENTING GSM IN RDL (THE VANU RADIO DESCRIPTION LANGUAGE™)

John Chapin, Victor Lum, Steve Muir  
Vanu, Inc.  
Cambridge, Massachusetts  
{jchapin,viclum,steve}@vanu.com

## ABSTRACT

*RDL is the Radio Description Language being developed at Vanu, Inc. It is a language specialized to the needs of software radios. One important goal for software radio languages is portability, which is the capability to execute a waveform application on hardware platforms from different manufacturers. RDL is designed to improve the portability of waveforms compared to existing approaches such as the JTRS SCA. This paper describes RDL and reports the experience of a team of engineers who used it to implement the physical and link layers of the GSM cellular telephone waveform.*

## INTRODUCTION

Waveform portability is an important goal for potential users of software radios, who seek to reduce the traditionally high cost of achieving interoperability among radio systems acquired from multiple vendors. For example, a major goal of the Software Communications Architecture (SCA) of the US DOD Joint Tactical Radio System program is waveform portability. [1]

Waveform implementations have two parts that face significantly different portability challenges. In this paper we use the following terms for these parts. The *application* part configures and controls the system, and implements higher level functions such as protocol state machines and network routing. The *signal processing* part implements the transforms between user data and a sampled representation of a RF waveform.

Although the SCA does not state so explicitly, it focuses on improving the portability of the application part, not the signal processing part. The SCA standardizes the interfaces between the application part and the other components of the system. [2] The SCA envisions the

---

Supported under Award number 2000-MU-CX-K024 from the Office of Justice Programs, National Institute of Justice, Department of Justice. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Justice.

signal processing function as part of the platform, which will be built by the platform vendor with sufficient flexibility to support a variety of waveforms.

The separation between portable application part and platform-provided signal processing part will result in a level of waveform portability that falls somewhere between two extremes. In *known-waveform portability* only the waveforms considered by the platform vendor during platform development can execute correctly. This type of portability reduces the cost to develop a new platform that supports a range of preexisting waveforms. In *new-waveform portability* new waveforms can be developed to run on the platform, including waveforms radically different from those considered by the platform vendor, assuming the hardware has sufficient computational capability. This type of portability additionally reduces the cost to field a new waveform to a variety of preexisting platforms.

The extreme of new-waveform portability is an unapproachable ideal. However, software radios should be designed as far along the range towards new-waveform portability as possible, to maximize the flexibility and minimize the upgrade cost experienced by users.

Two factors limit the amount of new-waveform portability achieved. First is the flexibility of the signal processing subsystem. At present most radios use dedicated hardware to perform some signal processing functions, limiting them to the waveforms known at design time. However, in the future platforms will continue to increase in flexibility, as processors improve in performance and move more of the signal processing task from hardware to software.

The second portability limitation is the interface used by the application part of the waveform to configure the signal processing subsystem. If the platform offers flexibility along an axis not exposed through the interface, new waveforms cannot take advantage of it. It is therefore important to design current standards such as the SCA with an eye towards future highly flexible platforms.

The SCA attempts to provide some level of new-waveform portability by defining a set of functions giving the

application part control over a great variety of signal processing parameters. This API was developed by studying the signal processing requirements of a designated basis set of waveforms. [3] Such an API is likely to support only future waveforms closely related to the basis set, for two reasons. First, waveforms exhibit great variation in signal processing requirements. There is a high probability that the interface will lack the ability to express some aspect of the processing needed for a new waveform. Second, attempting to support a wide variety of possible signal processing functionality will make the API extremely complex. This increases the chance that only the particular configurations considered by the platform vendor will function correctly. *Therefore, we believe the approach used in the current SCA will limit SCA-compliant systems to known-waveform portability.*

This paper describes a different approach to the interface between the application part and the signal processing subsystem. RDL is a language rather than a list of functions. This enables the application part of the waveform to express a much larger range of desired signal processing functionality, in a way that is less complex for the signal processing subsystem to implement. RDL shows how this interface can be structured to enable platforms to support a significantly greater range of new waveforms, thereby significantly reducing user costs.

### CONCEPTS BEHIND RDL

The signal processing part of most waveforms can be expressed as a pipeline or graph whose nodes are primitive signal processing functions chosen from a standard library. Examples of primitives include various types of modulators, convolutional encoders, Viterbi decoders, and filters. This fact is widely exploited by rapid prototyping and analysis tools.

RDL extends this approach to the fielded system. The waveform application installed on a platform builds a graph of software objects. Each object denotes a particular signal processing function, and has member data fields for the various parameters of the function, but does not contain the code that implements that function. The resulting graph *describes* the physical layer of the desired waveform.

The waveform application delivers this graph to the RDL runtime system, which uses the hardware and software resources available on the platform to *implement* the specified processing. The runtime system can, for example, examine the graph and decide whether it (or some subgraph) matches the capabilities of a legacy hardware modem that is present. If so, the runtime system

would initialize the hardware parameters of the modem based on the parameters set in the graph. If no hardware assist is available, the runtime system can use the information in the graph to configure a software signal processing system.

In the Vanu software signal processing system, there is a library of processing modules corresponding to the objects that may be in the RDL graph. The runtime system instantiates an implementation object from the library for each node in the application graph, sets its parameters as specified by the application, and makes the appropriate high-speed data connections between the objects and lower-speed control connections to the application part. Some of the library objects are fully generic, such as filters and decoders, while others provide specialized services for different waveforms.

### RDL LANGUAGE

RDL offers two primary constructs.

- *Modules* are the nodes in the processing graph. Each corresponds to a signal processing function.
- *Assemblies* are graphs. An assembly contains modules or subassemblies, connected into a particular topology.

Additionally the language contains the notion of *streams* that express the graph connections along which data flows, *ports* that are attachment points for streams on modules, and *channel* types describing the messages that can flow along a stream.

Assemblies serve to simplify the expression of complex processing graphs that have meaningful internal structure. An example assembly would be the receive chain that monitors GSM control channel traffic. In a test device which monitors multiple control channels simultaneously, the control channel receive assembly could be instantiated multiple times in just a few lines of code, with the center frequency and other relevant parameters set differently in each one. Assemblies are not visible to the signal processing subsystem. They are expanded to lists of modules and connections before execution.

A key design decision is that an assembly is treated like a module. Assemblies have parameters, control and data ports, and can be instantiated, connected and disconnected from each other. More precisely, each assembly *implements* a module, just as a class implements an interface in an object-oriented language like Java. All uses of the assembly refer solely to the module, hiding the existence of the assembly. This approach improves

```

GsmFrame extends frame<boolean>;

module BurstInterpreter {
    // parameters
    parameter GsmBurstFormatType format;
    parameter int32 training;
    parameter int32 inputLength;
    // data ports
    datain GsmFrame input;
    dataout GsmFrame output;
    // control ports
    controlout BiToSyncChannel SyncResponse;
    controlin SyncToBiChannel SyncCommand;
}

channel BiToSyncChannel {
    void ReportOffsetAverage(
        int32 offset,
        uint32 frameStamp,
        uint32 numElements);
    void NoSequenceFound(uint32 frameStamp);
}

```

**Figure 1: RDL Module**

modularity, since assembly internals can be freely changed, and allows the runtime system to substitute an optimized implementation for the assembly.

**Modules:** Figure 1 shows part of a RDL module description file. This describes the module that searches for the known training sequence in the middle of a GSM time slot and returns the data bits surrounding it which contain the information transmitted in that slot. Comments specifying its behavior have been omitted from this listing.

In the module declaration, the parameters are the values that the application can specify to control the behavior of the signal processing function. The data ports can be connected to data streams enabling exchange of data values with other modules. The control ports can be connected to control streams enabling exchange of out-of-band control signals with other modules.

Each port is labeled with a type which indicates the values that may be sent or received through that port. Typing has turned out to be very useful to avoid errors in large processing graphs.

Data ports are limited to scalar and array types to enable a high-speed streaming implementation. The GsmFrame type shown in the example is an array of bits.

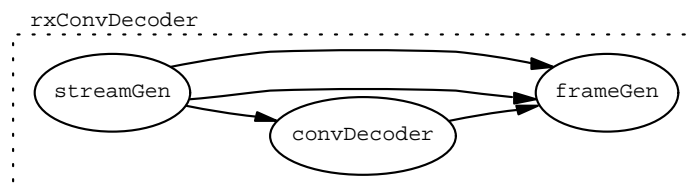
The channel construct declares a type that may be used for a control port. In the example shown, control ports of type BiToSyncChannel can carry two messages, one that delivers three integer data values and one that delivers one.

```

module RxConvDecoder {
    parameter EncoderFormat format;
    dataout GsmFrame output;
    datain GsmFrame input;
}
assembly RxConvDecoderDef
    implements RxConvDecoder {
    module ConvDecoder convDecoder;
    module DecoderFrameGenerator frameGen;
    module DecoderStreamGenerator streamGen;
    // data flows
    streamGen.mProtectedOutput ->
        convDecoder.input;
    streamGen.mUnprotectedOutput ->
        frameGen.mUnprotectedInput;
    streamGen.mHeaderOutput ->
        frameGen.mHeaderInput;
    convDecoder.output ->
        frameGen.mProtectedInput;
    // link ports
    frameGen.mOutput -> output;
    input -> streamGen.mInput;
}

```

**Figure 2: RDL Assembly**



**Figure 3: Data flows of assembly in Figure 2**

Although function declaration syntax is used in channel declarations for convenience, control streams are message pipes with asynchronous queued message delivery.

**Assemblies:** Figure 2 shows an assembly consisting of three component modules, each of which might be implemented as an assembly or as a primitive module. The function of this assembly is to separate its input into encoded and unencoded parts, decode the encoded part, and recombine the parts to produce a fully decoded output stream.

The first part of the assembly declares the modules that are parts of the assembly. The second describes the desired connections among the modules. The syntax X.Y refers to the port named Y of the module named X. The data flows lines in this assembly describe the graph shown in Figure 3. The link port lines of the assembly indicate which ports on the subcomponents act as the ports of the module that the assembly implements. The code that initializes the parameters of the parts of the assembly based on the format parameter of the assembly is not shown.

## INTERFACE TO HIGHER LAYERS

The processing graph described by a collection of RDL modules and assemblies is one part of a larger waveform application. Other parts of the application, such as protocol state machines, need to interact with the processing graph. For example, a processing module that monitors received signal strength must notify the GSM application to begin a handoff when signal strength drops below some threshold.

RDL reuses the control channel construct described in the previous section for connections between the processing graph and higher layers. A module which has control ports but no data ports may be declared a *control module*. Control modules represent portions of the application rather than signal processing components.

When an application delivers a processing graph to the RDL runtime for execution, it provides a reference to an object in its address space for each control module in the graph. Control messages sent to controlin ports of that module by other modules in the graph are delivered as procedure calls to the application object. For the reverse path, there is an API by which the application can send messages through the controlout ports of the control module.

## RDL IMPLEMENTATION

The current implementation uses Java class files as the download format for RDL descriptions, C++ as the signal processing implementation language, and CORBA to connect the two. RDL modules are compiled to a collection of CORBA IDL, Java, and C++. RDL assemblies are compiled to a Java class, which the waveform developer subclasses and extends with a function that sets the parameters of the components based on the parameters of the assembly.

The waveform developer writes Java for the application part of the waveform and links that code together with the Java produced from the RDL assemblies that describe the signal processing part of the waveform. This produces a portable download image as a Java jar file. Much of the application part of waveforms such as GSM consists of state machines, so a commercial statechart tool that generates Java code is integrated with the system. [4]

## STATUS OF THE GSM IMPLEMENTATION

The implementation of GSM in RDL has reached a laboratory demonstration level of functionality but is not yet ready for field trials. For example, the current demodulator does not handle noise robustly. Such

limitations do not hinder the evaluation of RDL. On the other hand, runtime changes to the RDL graph are not yet supported by the runtime system. The GSM engineering team plans to use this feature extensively but no information is yet available on the effectiveness of the feature's design.

The RDL description of GSM layer 1 is built from 36 different primitive modules and 28 different assemblies, totalling 3190 lines of RDL code including comments. In one execution configuration, when a GSM mobile is camping on a control channel waiting for a call, the processing graph contains 20 signal processing modules and 7 control modules, connected by 25 data streams and 16 control streams.

## LESSONS LEARNED

Implementing GSM enabled evaluation of a variety of aspects of RDL. Three are described here.

**Assemblies:** The assembly construct proved extremely successful. Writing assemblies rather than flat graphs enables the waveform programmer to encapsulate knowledge of a part of a waveform in a reusable component and hide its internal details. This contributed to the small size of the RDL description and its readability.

In one example, the lead GSM developer decided to test whether the system could receive two channels simultaneously. Given an assembly that implemented a single receive channel, it took just a few lines of code to configure the signal processing subsystem to implement two at the same time, assign them to the appropriate channels and time slots, and execute the test. Performing the same configuration operation without assemblies, or with a parameter-controlling API such as that used in the SCA, would have been much more expensive.

**Streams:** Developers reported that several aspects of the stream design were successful. First, the type system worked well to catch simple programming errors. Ports are typed and their connections are checked for type matches. Second, developers appreciated that a single stream construct is used for both data and control streams, despite the significant semantic differences between them, so the developer can just connect A to B without worrying about what A and B are. The use of function call syntax for channel types, which do not have function call semantics, caused initial confusion but was regarded as the correct approach once the construct became familiar.

**Expressiveness:** The language appears to provide as clear a description of a signal processing graph as can be

achieved textually. The decomposition into assemblies keeps the number of connections in each group down to a reasonable size. Still, graphical representations are easier to work with, so we wrote tools to produce useful visualizations (using the *dot* tool from AT&T [5]). Despite the importance of visualization, textual source code is better than graphical source for a language like RDL because of the availability of editing, source code management, compiler and other tools.

The lead GSM developer found himself having to write a nontrivial amount of boilerplate for the assemblies. We wrote a simple graph-editing tool with a RDL code generation backend that handles most of the common cases. This proved to be successful at reducing the coding effort. It is not clear how to make the language significantly more concise without giving up types or the ability to handle certain uncommon but important cases, so this approach appears a reasonable compromise.

### PORTABILITY OF RDL WAVEFORMS

RDL as a language appears to be completely flexible: any desired waveform can be expressed in RDL. That does not mean, however, that any desired waveform can execute on a given platform. The set of modules in the runtime system determines which waveforms can execute on that platform.

An ideal runtime system will support a flexible set of simple primitive modules that can be combined to implement any desired waveform. It is difficult to approach the ideal, for two reasons:

- Waveforms exhibit great variation in signal processing requirements. To make it possible to implement any desired waveform, the individual modules need to implement the smallest unit of functionality possible. This exposes the performance overhead of the module interconnect mechanism of the signal processing subsystem. Practical implementations need to choose a module size that balances the desire for flexibility against the need to amortize these overheads.
- The implementation of individual functions can be significantly optimized if the parameters to those functions are known. Therefore there is a tension between performance and generality of modules. A module which implements a particular linear feedback shift register, for example, is substantially faster than a generic register instructed to generate that particular polynomial. Practical implementations need to choose a module generality that balances the desire for flexibility against the need to optimize performance.

Examining the GSM implementation, we find that the modules fall into two categories, those that are generic signal processing functions, and those whose functions appear specific to GSM. Six modules were reused unchanged from previous waveforms. Ten were newly developed but appear widely applicable to 2G and later cellular systems. Twenty appear unlikely to have any utility except for waveforms similar to GSM.

These numbers show that the current system falls somewhere between known-waveform and new-waveform portability. The GSM waveform could not have executed on a preexisting fielded system, developed without knowledge of GSM, because of the twenty modules used that perform GSM-specific tasks.

However, if a software signal processing system like that built by Vanu is used, the necessary modules could be added to the fielded system through an over-the-air platform-specific software upgrade. Once these modules are present, any of a wide variety of new waveforms that reuse components of the GSM design can execute without platform-specific software development.

### CONCLUSIONS

The language approach of RDL allows the signal processing capabilities of a software radio platform to be combined together and reused in ways not considered by the platform designer. Therefore a platform providing an RDL interface to its signal processing functionality will support a much wider range of new waveforms than a platform whose interface is a set of signal processing parameter control functions. RDL will thus improve the flexibility and reduce the costs experienced by end users who seek to upgrade fielded software radio systems.

A language like RDL should be considered as a potential extension to the JTRS SCA, to ensure the SCA properly supports future software radios with highly flexible signal processing subsystems.

### REFERENCES

- [1] Joint Tactical Radio System Request for Information (RFI), JTRS2000-RFI-01, July 2000.
- [2] Joint Tactical Radio System Software Communications Architecture, Version 2.0. <http://www.jtrs.saalt.army.mil/docs/documents/sca.html>
- [3] Application Program Interface Supplement to the SCA Specification, Appendix B. See [2].
- [4] <http://www.wrs.com/products/html/betterstate.html>
- [5] <http://www.research.att.com/sw/tools/graphviz>