

# Using Domain Specific Modeling in Developing Software Defined Radio Components and Applications

Bruce Trask

Contact Author [bt@prismtech.com](mailto:bt@prismtech.com)

Angel Roman

PrismTech Corporation

## Abstract

*This paper details the application of Domain Specific Modeling and Model-Driven Engineering (MDE)<sup>16</sup> to the software defined radio (SDR) domain. The software defined radio domain has very unique characteristics as its systems typically are a confluence of a number of typically challenging aspects of software development. To name a few, these systems are usually described by modifiers such as, embedded, real-time, distributed, object-oriented, portable, heterogeneous, multithreaded, high performance, dynamic, resource-constrained, safety-critical, secure, networked, component based and fault-tolerant. Each one of these modifiers by themselves carries with it a set of unique challenges but building systems characterized by all of these modifiers all at the same time makes for quite an adventure in software development. In addition to all of these, it is quite common in these embedded systems for components to have multiple implementations that must run on disparate processing elements. With all of this taken into account, it stands to reason that these systems could and should benefit greatly from advances in software technology such as product line engineering, domain-specific modeling and model-driven engineering.*

## 1. General Background

For the past twenty years, there has been a continuous evolution in electronic communications equipment. The evolution can be described as one of moving the radio functionality from being located in the hardware platform running with proprietary processors and circuitry to being located in firmware running on programmable logic and then to being located in software running on general purpose processors. The driving force behind this evolution has been the need to leverage the inherent greater malleability of software versus that of hardware. As radio functionality continues to move into software it becomes more commercially viable to maintain, configure, test and reuse communications algorithms and functionality as well as the hardware on which it runs. The communications industry has coined a term for this type of communications equipment: the *Software Defined Radio*[14].

The conventional radio development paradigm during the 1980s and 1990s involved make one-off systems that had to be redesigned, and recoded as new hardware platforms evolved. To solve this, a full Commonality Variability Analysis (CVA) was done across the entire family of existing radios. The result was the Software Communications Architecture (SCA)[1]. It was the key production asset released broadly to both the manufactures of the entire family of military radios as well as the public domain. This SCA defines five primary aspects of next-generation communications equipment software

- A standard component object model
- A standard deployment and configuration component framework
- A standard declarative programming format for describing software components and how they are connected together
- A standard portability layer upon which component run
- A standard messaging format/middleware for inter-component communication

As a result, the SCA significantly furthers standardization of the software radio domain and thus brings many benefits to the domain such as interoperability, portability, reuse, and a level of

architecture consistency. As is the case with many new platform technologies, the SCA specification does a good job of solving many hard problems, but leaves some unsolved while simultaneously introducing new problems. Some of the problems that remain or are introduced include:

- Labor intensive implementations of the SCA object model in 3GL languages
- Lack of architectural consistency at various levels of implementations
- The learning curve of the specification and lack of effective training materials
- The technology gaps between software developers and radio domain experts
- Ensuring correctness of implemented systems
- The dynamic nature of the SCA, which opens the door to a host of runtime errors that would best be “left shifted” out of runtime into either modeling or compile time.
- A complex set of XML descriptor files which are difficult to get correct by hand as there are many rules that govern them above and beyond being well formed
- No formal meta-model or UML profile exists for the SCA
- While the SCA definitely raises the level of abstraction with regard to radio component development, it does not inherently provide an automatic and configurable means to get back to the lower, executable levels of abstraction or to its declarative languages.

We feel that essence of the problem can be boiled down to: *an advancement of platform technology without a commensurate increase in language technology*[22]. The language most used in radios today for the entire system is C and C++. These two languages were invented over twenty years ago. In that intervening time, there have been many advances in platform technology that have outpaced the ability of these third generations languages to suffice as the only real tool in the hands of the software developer tasked with implementing these new complex systems.

## 2. Enter Domain-Specific Modeling Techniques

In order to tackle and tame the complexity of these systems, the new specification and the platform technologies resulting from the product line analysis, new language technology was required to allow the developers tools to catch up to the platform technology. As such it was necessary to provide:

- effective support under the SCA that allows users to program directly in the terms of the language of the domain and specification, ideally in graphical and declarative form to the greatest extent possible
- means to ensure that the programming is correct
- means to automatically generate executable 3GL programming language implementations from these models
- means to automatically generate additional software artifacts that are synchronized with the model
- Those familiar with Domain-Specific Modeling will recognize the above bullets as part of the sacred triad of Domain-Specific Modeling: *Language, Editor, and Generator*[2].

Table 4 lists the activities used in tackling the complexity in domain and then leveraging Domain Specific Modeling techniques to it

General Approach	Radio Domain
Isolate the abstractions and how they work together, including commonalities and variabilities	The SCA
Create a formalized grammar for these – DSL	Create a formalized SCA meta-model
Create a graphical representation of the grammar – GDSL	Create a SCA specific graphical tool
Provide domain-specific constraints – GDSCL, DSCL	Program into the tool the constraints
Attach generators for necessary transformations	C++, C, Ada and VHDL generators

Table 1

One type of tool that can be used to develop the above software artifacts are what some refer to as Language Workbenches[2]; i.e. tools that allow a developer to define a domain-specific language and its graphical counter part, the editor, as well as a domain-specific generators that can iterate over the domain-specific model to produce executable artifacts. Some language workbenches available today include the Eclipse Modeling Framework and the Eclipse Graphical Editor Framework (EMF/GEF)[3], the Generic Modeling Environment (GME)[4], and Microsoft’s Visual Studio Team System Domain Specific Language Tools (VSTS DSL)[5].

### Defining the Domain-Specific Language (DSL)

The goal here is to provide a domain-specific higher level of abstraction with which both software and lay developers can program. Key to this is not only raising the level of abstraction but also providing domain-specific abstractions. Developers of SCA applications typically program in 3GL languages such as C, C++ and Ada.

The raising of the level of abstraction is made possible through the creation of a formalized metamodel expressed in terms of the particular language workbench. In this case this involves creating a metamodel that the Eclipse Modeling Framework can understand.

As stated before, the SCA provides a general architecture and UML diagrams as well as text-based behavioral descriptions and requirements and annotated XML DTD documents. While these are very detailed they are not formalized sufficiently to serve as a useful meta-model by themselves. The meta-model created and described here involved building upon the structure of the SCA and culling from the rest of the specification requirements, constraints and behaviors that together make up a complete and comprehensive meta-model characterizing the entire specification. As is usual, the group of developers building the meta-model are experienced SCA and software defined radio developers as well as experienced modelers.

It is from this meta-model that one provides the end user with the ability to program more directly in the domain. Additionally, end users are able to program more in the declarative than in the imperative; i.e. saying what they want to have, not specifying how it is to be done.

### Defining the Domain-Specific Graphical Language (DSGL)

What is needed next is a way to express the Domain Specific Language graphically or visually. This involves working within your Language Workbench of choice to adorn the Domain-Specific Language with graphical and visual artifacts that allow the user to program quickly and correctly and in a way that communicates correctly the essence of the architecture and design.

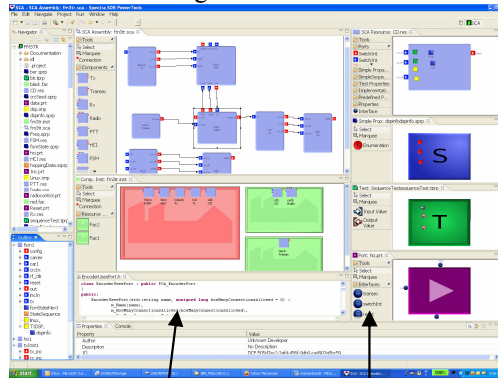


Figure 3

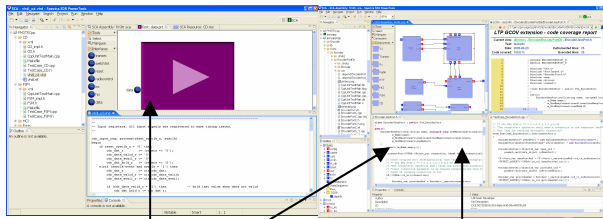
Figure 3 above shows the PrismTech Spectra SDR PowerTool modeling tool. This modeling tool allows end users to quickly and accurately build software defined radio components and connect them together. The DSGL is built and based on the underlying meta-model described earlier and can be persisted in textual form for processing by other programs. It is through this DSGL that end users program with very intuitive icons, images, tools, artifacts and property sheets. Just as UML provides different views to describe various aspects of object-oriented systems so to does this tool provide Domain Specific Views that allow users to design, express and communicate domain specific aspects of their designs.

### The Domain-Specific Constraint Language (DSCL)

Almost as important as what you see in the graphical tool illustrated in Figure 3 is what you don’t see. The very fact that the DSGL is based on the meta-model means that it restricts programming to within the bounds of the meta-model. In other words, the tool is metamodel-centric as opposed to GUI-centric. In this case, the GUI itself forces the user to abide by the structural and creational aspects of the meta-model. This goes extremely far in allowing the developer to program quickly and correctly in terms of their domain. Additional constraints can be added via various programming facilities of the language workbench being used. Concrete SCA-unique examples of these types of constraints include not being able to connect ports that support different interfaces or not exceeding connection thresholds of output ports. These are errors that are typically allowed to creep into the runtime system which lead to expensive integration and support problems. By “left shifting” these potential defects into the modeling/compilation phase, we can simultaneously harness the dynamic nature of the SCA runtime component deployment, configuration and connection paradigm and do so in a correct and robust fashion. The DSCL enforces structural compositional, directional, etc. constraints, pre-conditions, post-conditions and invariants

## Domain-Specific Generators (DSG)

Ultimately, the tool must be able to transform the domain specific language into an executable or imperative format, or to a form that can be transform easily by other compilers into an executable form. This is achieved through the connection of *Domain Specific Generators* to the Domain Specific Editors. Embedded systems are frequently targeted at disparate processing elements (e.g. general-purpose processors, digital signal processors, field programmable gate arrays (FPGA)) and as such the tool needs to be able plug in multiple domain specific code generators that can iterate over the model and produce multiple types of executable code.



Translate from declarative to imperative C++

Figure 4 shows examples of the software artifacts coming from the domain-specific generators. Having the key information captured in the model, changes in the model are instantly reflected in the generated code.

The SCA architecture is most effectively implemented using a number of industry standard Design Patterns. Most notably are the Extension Object Pattern[6], Extension Interface Pattern[7] and the Component Configurator Pattern[7]. These patterns are typically repeated over and over again in an SCA implementation with minor parameterization to account for the context in which they are used. The pre-validated implementations of these patterns can be generated directly from the domain specific generators. Many of these patterns capture infrastructure scaffolding, behavior required by the SCA specification as well as middleware concerns that can be difficult for radio developers to understand and get correct. Additional artifacts are generated from the model including, the XML descriptors, Unit Test Cases, documentation etc. The constraints of the tool straddle the editor and the generators. By using the generated code, the users can rely on prevalidated logic and patterns written by experts in the domain and thus they are “constrained”, if you will, to being correct in their implementation.

### 3. Benefits of Domain-Specific Modeling as applied to Software Defined Radios

A number of notable benefits become extremely apparent as a result of providing a domain modeling tool and all its constituent parts to the software defined radio domain.

- Increased productivity
- Increased correctness
- Synchronization of software artifacts.
- Involvement of lay programmers and increased communication amongst company teams.
- Lower cost of entry.
- Architectural consistency at the implementation level.
- “Left shifting” of defect detection from runtime to modeling time.

### 4. Lessons Learned

A. Creating Domain Specific Graphics - One aspect of domain specific tools that we found to be labor intensive, difficult to get

correct *and* user friendly is the creation and implementation of domain specific graphics, views, editors and layouts. General purpose GUI widgets, frameworks and tools abound but they are usually insufficient to express domain specific concepts clearly and accurately. Additionally, the graphical user interface frameworks available in the industry are quite complex and have steep learning curves. This is a potential area of difficulty for those developing domain specific tools. The Graphical Editor Framework (GEF) from Eclipse, for example, is quite large and takes a significant amount of time to become familiar with. To alleviate this problem, the industry is currently working on MDE tools to make the creation of domain specific graphics, views, editors and layouts much easier.[21]

B. Validating the Generated Code - The sheer volume of code that is generated from the MDE tool described above warrants a precise and scalable means with which to validate that the code emanating from the generators is correct. Applying once again the principles of Agile Software Development, we found that making heavy use of test cases to validate the continued correctness of the generated code was essential. Also important is the automation of the execution of these tests and the reporting of the results.

C. The Vendor Lock-in problem - While users of domain specific tools definitely need and want the increase productivity and correctness that tools such as these afford, they are simultaneously concerned with being locked in to the specific vendor’s tools and models. This is a potential serious pitfall that domain specific tool vendors must address from the start. We addressed the issue directly by choosing the Eclipse platform as the application framework upon which the tool is built.[3] This Eclipse platform is not only a Java IDE but is actually more so a malleable and extensible *application framework* for which to develop domain specific tools. One key characteristic of Eclipse is that it has a large degree of platform independence and thus frees our customers up from having to run on particular host platforms and are thus free to use Windows, Linux, Mac, Solaris etc. Eclipse does this while transparently providing a native look and feel for each particular host operating system.

The next step to address the potential vendor lock in problem is to use a standard model serialization syntax. The Eclipse Modeling Framework provides capabilities to serialize its models in XML Metadata Interchange (XMI) version 2.1 format. This goes a long way in allowing our users to access and transform their models as well as providing the capability to import/export them into other tools (such as popular Unified Model Language, UML, tools) as they see fit.

With regards to the metamodel itself, we are lead members of groups within standards bodies such as the Object Management Group (<http://sbc.omg.org>) that are leading efforts to standardize these meta models so as to make them open to the industry.

D. Tool interchange - Exchanging information between tools continues to be an issue for tool users. About the best one can do now is to provide standard serialization formats, such as XMI, to enable systematic interchange of data between tools. There are not now, however, any standard interchange protocols that exists between tools. We are also active participants in groups such as

the Model Integrated Computing PSIG at the OMG to assist in standardizing such protocols (<http://mic.omg.org/>).

E. Cross Tool Integration - Developers of complex software defined radio systems and more generally of complex distributed real-time embedded systems use myriad tools to get their job done. They need and want as much seamless an integration and look and feel across these tools as possible. Fortunately for our team, Eclipse was the correct choice to deal with this problem as well. Most Realtime Operating System (RTOS) vendors, for example, are migrating their C++ development environments to Eclipse and as such our tool integrates very well with them. Our tool can take advantage of the C/C++ Development Tools (CDT) project in Eclipse and the RTOS vendors' migration towards it. A similar evolution is occurring with UML tool vendors.

F. The Economics of Going Domain Specific - Weiss, Lai and Coplien [18][19] discuss the economics of software product line development in great detail. While there has been a "perfect storm" of recent critical innovations in the software industry that go a long way towards increasing the efficiency and viability of making domain specific tools for software product lines [20], developing such tools is still non trivial and quite complex. Whether it is economically feasible of course depends on the scope and size of the family of systems to which one is targeting the tool. For our domain of Software Defined Radios, the scope and family is very well defined and the commonalities and variabilities have been to a large degree isolated.

### **Summary and Conclusion**

The history of software has seen the continued process of raising the level of programming abstraction while simultaneously providing an automatic and configurable means to traverse to lower levels of more executable forms of programs. Additionally, this evolution has included the continued introduction of ways and means to express domain concepts and design intent effectively so that the end user can program more directly in the problem space and not in the solution space.<sup>15</sup> Using Model Driven Engineering and Domain-Specific Modeling via existing Language Workbenches in combination with Software Product Line Engineering is another effective step in this direction and one towards a viable commoditization of the software industry. Application of these techniques to the Software Radio Domain has yielded orders of magnitude of increase in productivity, correctness and robustness of these systems and can serve as the foundation for a graceful evolution of its products. The Eclipse Platform and Frameworks provide an effective "Language Workbench" with which to implement the Language, Editors and Generators of the full Domain Specific Modeling tool.

### **References**

- [1] Software Communications Architecture Specification, MSRC-5000A V2.2 November 17<sup>th</sup> 2001
- [2] Language Workbenches: The Killer-App for Domain Specific Languages?, Martin Fowler, 12 June 2005
- [3] The Eclipse Foundation
- [4] The Generic Modeling Environment, Institute for Software Integrated Systems, Vanderbilt University
- [5] Visual Studio Team System DSL, Visual Studio 2005 Microsoft

- [6] Extension Object, Erich Gamma, Pattern Language of Program Design 3., R.C. Martin, D. Riehle, F. Buschmann eds. Addison Wesley 1998
- [7] Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Schmidt et. al. Wiley 2000
- [9] Agile Software Development, Patterns Principles and Practices, Robert C. Martin, Prentice Hall 2002
- [10] Extreme Programming Explained: Embrace Change 2/E, Kent Beck et. al. Addison Wesley 2005
- [11] Test Driven Development: By Example Kent Beck, Addison Wesley 2003
- [12] Refactoring: Improving the Design of Existing Code, Martin Fowler et. al. Addison Wesley 1999
- [13] Refactoring to Patterns, Joshua Kerievsky Addison Wesley 2005
- [14] Joseph Mitola home webpage
- [15] Model Driven Engineering IEEE Computer Feb 2006 Douglas C Schmidt
- [16] Carnegie Mellon Software Engineering Institute Product Line webpage
- [17] Doug Schmidt personal email communication
- [18] Software Product-Line Engineering: A Family-Based Software Development Process, David Weiss et. al. Addison Wesley 1999
- [19] Commonality and Variability in Software Engineering, Coplien et. al. IEEE Software 1998
- [20] Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley 2004
- [21] Eclipse Graphical Modeling Environment
- [22] Model Driven Engineering, Douglas C. Schmidt IEEE Computer Feb 2006