Using Domain-Specific Modeling to Develop Software Defined Radio Components and Applications

Vikram Bhanot
Dominick Paniscotti
Angel Roman
Bruce Trask
Contact Author bt@prismtech.com
PrismTech Corporation
www.prismtech.com

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 and configurability of software versus that of hardware. As radio functionality continues to move into software, or looking at it another way, as that software moves "closer to the antenna", it becomes more commercially viable to maintain, configure, test and reuse communications algorithms and functionality as well as the hardware on which it runs. This evolution is very similar to that of the computer itself with today's PCs running applications, the bulk of which exist as software running on general purpose hardware.

The communications industry has coined a term for this type of communications equipment: the *Software Defined Radio*[14].

As the radio and communications domain moves into a software centric solution, it is only natural that it leverage advances in the software domain as part of its implementation. These advances include object orientation, frameworks, component based design, middleware, in addition to imperative and declarative languages. More recently, the rise in abstraction level of the radio platform in the form of operating systems and middleware in combination with advances in modeling tools has opened the door to allow the evolution of communications software to enter the realm of model-driven development. This is fortuitous as the complexity of these communications systems has increased so dramatically that the viability of these new systems now hinges on the increased productivity, correctness and robustness that model-driven development affords.

This paper details the application of model-driven development, and more specifically, domain-specific modeling to the software defined radio domain. This 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 embed-

ded 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 domain-specific modeling and model-driven development.

Detailed background

In 1999, a consortium of the leading U.S. military radio developer companies created the Software Communications Architecture (SCA)[1]. This is one of many possible software defined radio models that can be input into the Domain Specific Modeling process and techniques discussed in this paper. We chose this one as concrete example of one such architecture since it is an open standard freely available to all.

This SCA defines five primary aspects of next-generation communications equipment software

- A standard component object model
- A standard deployment and configuration 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 for intercomponent 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. However, the SCA specification does not solve all of the issues associated with implementing these complex systems. Some of the problems that remain 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 modelingl 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.

Enter Domain-Specific Modeling

In order to tackle and tame the complexity of these systems and of the new specification 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*. Couched in terms of Domain Specificity and at a finer granularity, these three elements map to:

- a *Domain-Specific Language* (DSL)
- a *Domain-Specific Graphical Language* and *Domain Specific Views* (DSGL, DSViews)
- a *Domain-Specific Constraint Language* (DSCL)
- a family of *Domain-Specific Code Generators* (DSG).

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	*The SCA*
Create a formalized grammar for these – DSL	*Create a formalize 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], Microsoft's Visual Studio Team System Domain Specific Language Tools (VSTS DSL)[5], and MetaCase MetaEdit+ [8].

To allow users to run on multiple host platforms most easily and to integrate with addition eclipse tools and frameworks, we chose to use the EMF/GEF solution.

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. One of the goals of domain specific modeling is ***simplified modeling and programming in the problem space*** vs. ***complex modeling and programming in the solution space***. Figure 1 below juxtaposes two possible ways to represent the same concept in the SCA Software Defined Radio Domain. The left side diagram shows a typical UML diagram for a trivial SCA Component with two ports and two properties. The C++ source code is even more complicated. The right side diagram shows the same entity in terms of a higher abstract concept, a component with two ports and two properties, that is much more readable and less complex

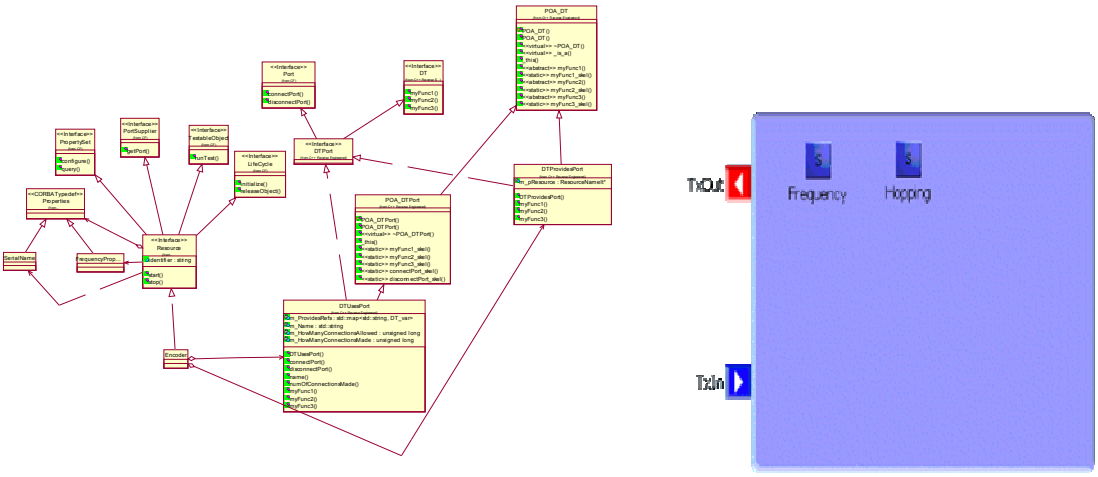

Figure 1

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. Fig 2 shows a greatly simplified metamodel for the SCA. Naturally, the full meta-model for the entire SCA is much more involved but for the purposes of demonstration and saving space we have presented a simplified version of it.

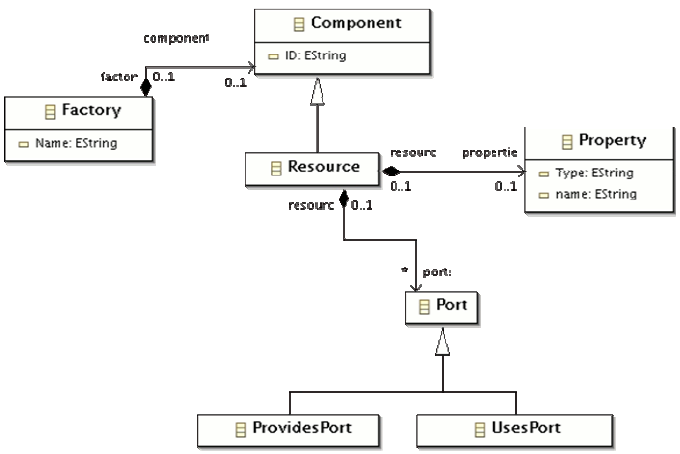

Figure 2

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 desribed 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. Listing 1 shows a simple example of the persistent form of the Domain Specific Language in accordance with the metamodel.

```
<?xml version="1.0" encoding="ASCII"?>
<com.prismtech.spectra.sdr.sca2_2.models:Assembly
    xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:com.prismtech.spectra.sdr.sca2_2.models="http://com.prismtech.spectra.sdr.sca2_2.mod
els">
  <components Name="BitFlipper" organization="PrismTech" id="DCE:8f647411-91a1-4295-bbc6-
6d3eff4982f7">
          <ports xsi:type="com.prismtech.spectra.sdr.sca2_2.models:UsesPort" instance-
Name="TX" name="Data"/>
       <ports xsi:type="com.prismtech.spectra.sdr.sca2_2.models:ProvidesPort"
instanceName="RX" name="Data"/>
  </components>
</com.prismtech.spectra.sdr.sca2_2.models:Assembly>
```

Listing 1

While providing a higher level of abstraction this text based language can still be labor intensive, error prone and hard to read. This leads directly into the next step of Domain-Specific Modeling.

Defining the Domain-Specific Graphical Language (DSGL) and Views (DSVs)

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.

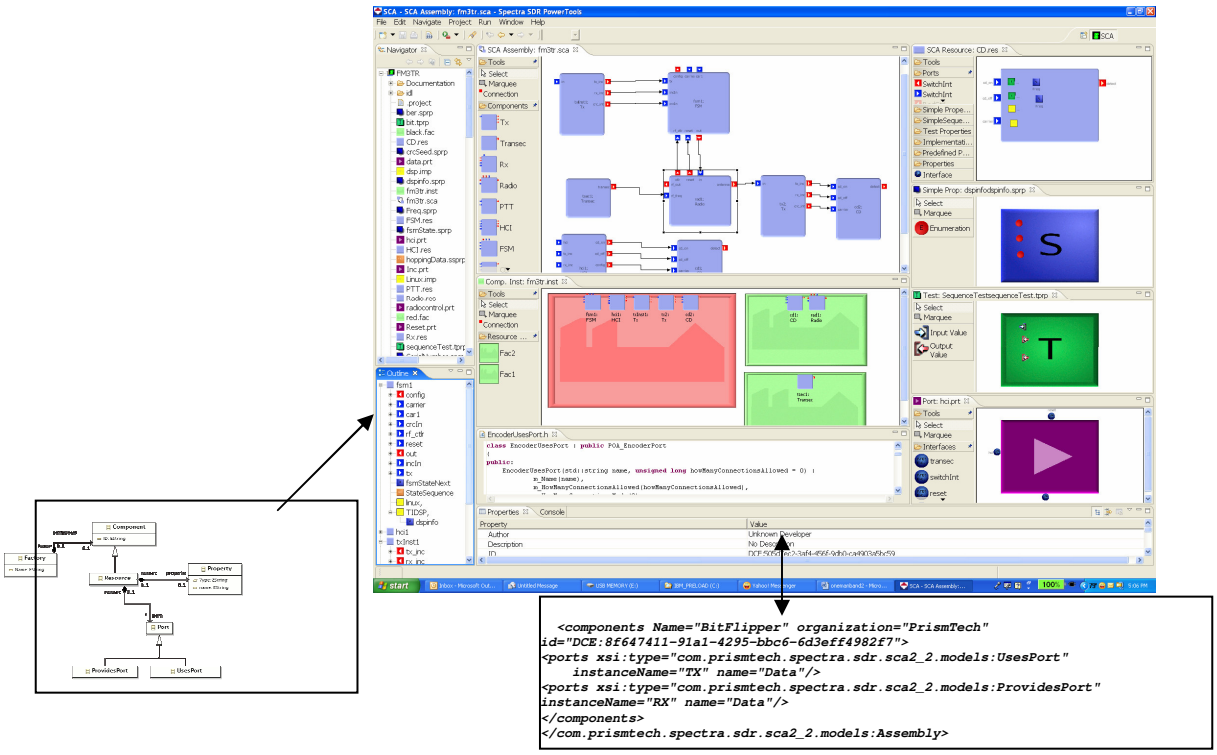

Figure 3

Figure 3 shows the PrismTech Spectra SDR PowerTool modeling tool. This modeling tool allows end users to quickly and acurrately 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. Additionally, the Domain-Specific Modeling tool provides the end user with ability to program in the declarative versus the imperative.

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.

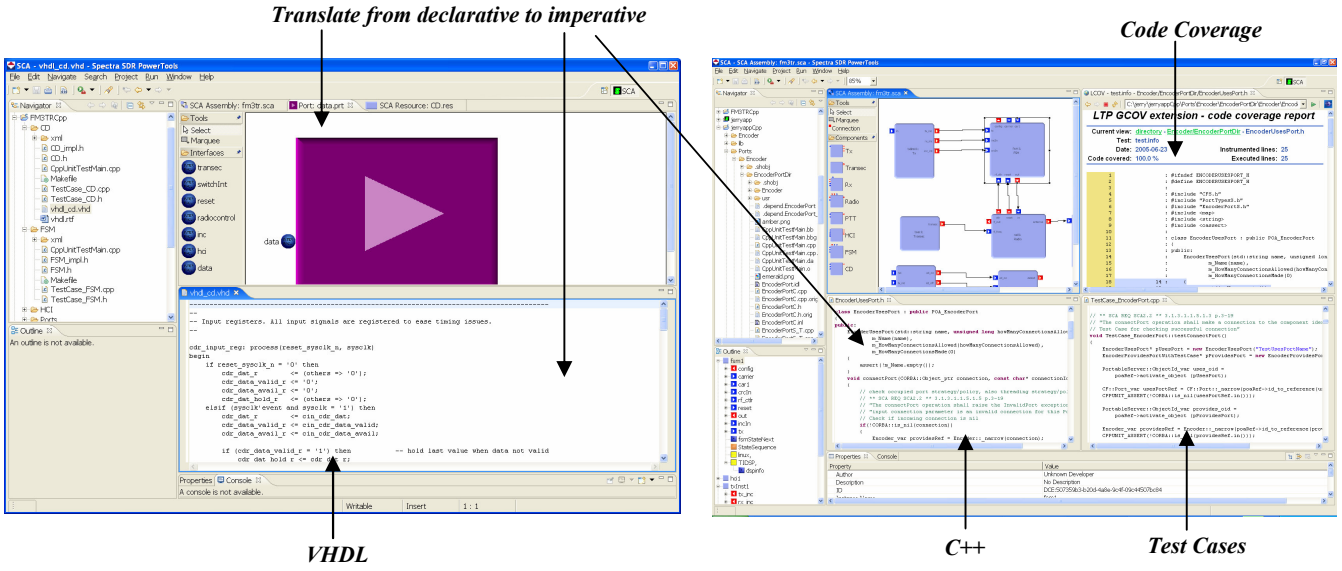

Figure 4

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 paramaterization 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.

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 – users can program at a much higher level of abstraction and use generators to automatically get to lower levels that can thereafter be transformed and executed. The increased level of abstraction is coupled with the fact that the DSL is much more declarative in nature and so the users become less concerned with how actions are done and more concerned with *that* they are done. Users of the tool report a minimum of 500% increases in productivity and compare the magnitude of gains to be analogous to using a compiler to generate assembly code from higher order languages.
- Increased correctness – the generators provides prevalidated logic and other artifacts
- Synchronization of software artifacts. Since the artifacts are generated directly from the model, the maintenance burden of maintaining them all is greatly reduced
- Involvement of lay programmers and increased communication amongst company teams. Since the model is expressed in problem domain terms and not solution domain terms, the communication of the model encompasses more disciplines beyond software engineering to include hardware and systems engineering and management teams.
- Lower cost of entry. As much of the infrastructure detail is captured in the meta-model, editor and generators, the learning curve of developing software defined radios for a particular domain is greatly reduced.
- Architectural consistency at the implementation level. While the SCA mandates architectural form at the interface level it does not at the implementation level. This opens the door to many different architectural implementations. While this is necessary in some uses cases, in many it is not and results in unnecessary complexity and maintenance burdens. The degree to which the applications have architectural consistency in their implementations determines the ease of maintenance by a central maintenance body.
- "Left shifting" of defects from runtime to modeling time. This provides orders of magnitude of cost savings across the development cycle

Dealing with Change

Model Driven Development as described above goes along way to handling many of the commonalities and variabilities in the software defined radio domain. The subsequent "closing" of the design to effects of movement in particular degrees of freedom and change must be strategic. No design/approach can close a software product to all degrees of freedom or variablities[9]. We have found particular techniques to be effective in handling changes in meta-models and domain-specific generators. Some of these include leveraging many of the techniques of the Agile Software Development world that enable one to "embrace change"[10] more easily than with older software methodolgies.

At the heart of these techniques are Test Driven Development[11], Refactoring[12], Refactoring to Patterns[13]. In addition to keeping the design of the tool as simple as possible, tests form suites that are useful in quickly isolating the exact areas where meta-model changes affect designs and thus provide targeted areas for refactoring. Refactoring towards new patterns that become applicable as new requirements enter the picture pro-

vides developers with very codified means of moving existing designs to new designs that more effectively handle new commonalities and variablities introduced by various changes in requirements.

In addition to agile software techniques, generative techniques can also be leveraged *within model driven development tools themselves.*

The most notable element of the model driven development that is affected by changes in the meta-model is the domain-specific editor that allows one to manipulate the domain-specific language via domain-specific graphical articfacts. Providing an additional generator framework in between the meta-model and editor that automatically generates a great deal of the editor is very effective in mitigating changes in the meta-model on model driven development tools.

Meta model changes in the meta-model cause unwanted effects to the DSL and DSGL code. When code generation comes to mind one usually thinks about the end product: the C++, Java, VHDL, documentation, and/or xml files that are generated. One reason why DSLs are favored is due to the decoupling of the model from the the generated files. If the user wants to create a change in his model he/she *effortlessly modifies the model and lets the generators/translators regenerate the output.*

Ideally, developers should take advantage of generators and translators when creating the DSGL as well. The goal would be to extract and generate as much of the DSGL as possible given the meta model. Several tools exists which allow the user to design the meta model and generate a DSGL editor. These tools are effective, however, they are usually lacking when it comes to domain specific visualizations. Some of them allow the user to specify bitmaps and connections anchors for any given model element. However, since these tools are generic it sometimes take great efforts to modify an editor that is generated (visual aspects) rather than designing one correctly from scratch making the correct visual abstractions for the look and feel desired.

When using a generic programming language one usually creates constructs that map directly to the problem domain. DSLs eleminate the need to specify unnecessary generic syntax/constructs in order to create a domain specific solution. Applying the same paradigm to the creation of a DSL editor can work as well. Instead of using a generic domain specific graphical editor generator (similar to using c++), tool developers can create a Domain Specific Domain Specific Language Editor Generator (DSDSL for a specific DSL). Once the look and feel is determined one can factor out the visual programming aspects and create generators that would interpret relationships between objects in the meta model and map them to a specific visual representations. Next time a change occurs in the meta-model the user can *effortlessly modify the metamodel and let the generators/translators regenerate the editor.*

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 effectively so that the end user can program more directly in the problem space and

not in the solution space. Using Model Driven Development and Domain-Specific Modeling via existing Language Workbenches is another effective step in this direction. 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.

References

[1] http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html

[2] http://www.martinfowler.com/articles/languageWorkbench.html

[3] http://www.eclipse.org

[4] http://www.isis.vanderbilt.edu/Projects/gme/

[5] http://msmvps.com/vstsblog/archive/2005/07/02/56408.aspx

[6] http://www.smallmemory.com/almanac/Gamma98.html

[7] http://www.cs.wustl.edu/~schmidt/POSA/

[8] http://www.metacase.com/

[9] http://www.objectmentor.com/resources/articles/ocp.pdf

[10] http://www.aw-bc.com/catalog/academic/product/0,1144,0321278658,00.html

[11] http://www.aw-bc.com/catalog/academic/product/0,1144,0321146530,00.html

[12] http://www.aw-bc.com/catalog/academic/product/0,1144,0201485672,00.html

[13] http://www.aw-bc.com/catalog/academic/product/0,1144,0321213351,00.html

[14] http://web.it.kth.se/~jmitola/