EXPERIENCE REPORT: RAPID MODEL-DRIVEN WAVEFORM DEVELOPMENT WITH UML

Shaw-Ping Lee
(Thales Technology Centre Singapore, Singapore, shawping.lee@asia.thalesgroup.com);
Mark Hermeling
(Zeligsoft Inc., Quebec, Canada, mark@zeligsoft.com);
Chueh-Min Poh
(Thales Technology Centre Singapore, Singapore, steven.poh@asia.thalesgroup.com)

ABSTRACT

Of great value to the Software Defined Radio (SDR) application developer is an integrated tool chain that combines a domain-specific SCA tool for the development of SCA artifacts with a general-purpose UML-based tool for the development of component artifacts. We have defined a process which integrates Zeligsoft CE™ with Telelogic Rhapsody to achieve waveform modeling and component development in tandem in three independent and repeatable steps. The three steps involve reverse-engineering a component C++ worker class in Rhapsody, modeling the component function in UML, and configuring the Operating Environment (OE) to compile the framework code with the component code. This process has been applied to develop successfully two waveforms from a single architecture model for two OEs. Engineers truly benefit from using this process in term of time saving, shorter learning curve, and reduced porting effort.

1. INTRODUCTION

In 2005, Thales Technology Centre Singapore (TTCS) collaborated with Thales Defense Deutschland (TDD) to design and develop a SCA SDR WNW waveform. This R&D project aims to create competency in SDR and SCA in the Asia region. TTCS was the lead on the project and combined TDD's expertise in Physical layer protocols with TTCS's own experience in building upper level protocols. Zeligsoft was brought in to the project to provide tools, training and consulting to TTCS in the use of the Software Communications Architecture (SCA) standard for SDR deployment and configuration.

As often with embedded software development projects, hardware was not available at the beginning of the project. Hence, TTCS decided to start development using the HARRIS dmTK SCA Core Framework (CF) running on an MS-Windows PC and then transfer the project to the actual radio hardware once it became available. As a result, the solution that TTCS adopted had to consider portability and reusability. The TTCS engineers are most comfortable with the model driven approach for code development and hence one of the requirements for the development solution was that it incorporate a UML modeling tool as well as SCA domain knowledge.

Together with Zeligsoft, TTCS defined a development process that integrates Zeligsoft Component Enabler (CE™) with Telelogic Rhapsody to achieve waveform modeling and component development in tandem. Transformation of SCA models from Zeligsoft CE to Rhapsody's UML environment was greatly simplified as Zeligsoft CE's SCA models are also UML-based.

The approach comprises a three-step process with the ability to repeat each step independently in order to provide iterative workflows. Step one involves transforming an SCA component's functional code, produced by Zeligsoft Code Generator, to Rhapsody through the use of the reverse engineering capability in Rhapsody. Step two involves developing and generating component functional code in Rhapsody. Finally, step three involves configuring the operating environment in Zeligsoft CE to compile the framework code with the functional code generated by Rhapsody. Additional details about these three steps will be described in section 2

This process was applied to develop a WNW waveform for two hardware platforms: HARRIS dmTK CF running on a MS-Windows PC and SCARI++ CF running on Spectrum Signal Processing's SDR-4000. This process simplified component behavior modeling in Rhapsody, with the SCA framework code handled by Zeligsoft's UCG. In addition, it significantly reduced the time and effort required to port the application from the PC environment to the SDR-4000 environment to roughly 1 week per component. This flexibility is a testament to both tools' breadth of support for

multiple operating environments. Engineers truly benefit from the integrated tool chain as they can quickly develop their components, even with limited SCA knowledge. This ultimately translates to spending less time tackling the intricacies of framework code and more time focusing on the value provided by the component's function. In addition, less time spent on the low level details of component structure means less chance of bugs slipping into the project, reducing project risk.

The Zeligsoft SCA development solution comprises a modeling environment and a code generator. The Zeligsoft code generator strictly separates Common Object Request Broker Architecture (CORBA) and SCA infrastructure code from the functional behavior. The functional behavior is implemented by the engineer using his or her favorite Integrated Development Environment (IDE), or a UML tool such as Telelogic Rhapsody. The functional behavior is typically referred to as "the Worker". The Worker class can easily be reverse-engineered into a UML tool and extended through that means. The separation between CORBA and SCA code and the Worker allows the engineer to focus on implementing behavior rather then implementing SCA. TTCS adopted Zeligsoft's solution [1] and used this as part of the development process for the protocol components on the General Purpose Processor (GPP).

1.1. Existing Tools and Methodology

The Software Communications Architecture (SCA) improves portability and re-usability through the use of components: independently deployable artifacts with strong encapsulation. A component in the SCA is defined by well-defined and strongly typed interfaces and is realized by one or more implementations: binary files that are compiled for a specific OE comprising Real-Time Operating System (RTOS), processors, CORBA ORB and SCA CF.

An SDR adhering to the SCA consists of a component-based platform and a component-based application (waveform). The latter is deployed (instantiated) on the former. For this to work a component needs to implement certain infrastructure code as documented by the SCA standard. The easiest way to think about a component is that it is an autonomous binary that can be started and then connected with other components to form an application.

Thus to implement a component, one must implement the infrastructure code as well as the actual functional behavior of the component. The interface code is not difficult to write, but it requires attention to detail and precise understanding of the standard. This makes it an error-prone activity with little added value. As with any piece of code, there are multiple ways to implement the infrastructure code (as for example, there can be many implementations found for the Queens [2] problem).
There are a number of SCA development tools available in the market for both research and commercial applications. Some tools are open-source and they all vary in terms of features and capabilities. These tools are domain-specific, that is, they understand the SCA domain. They understand SCA concepts such as components and ports and have built-in knowledge of how to translate that component and that port into source code. Each tool has its own way of implementing the infrastructure code. The infrastructure code usually has a placeholder for the component behavior code that the user can either author in his/her favorite IDE, in the SCA tool itself or in a general UML tool.

The true challenge is integrating the domain-specific tool with a general UML tool like Rhapsody. Rhapsody does not have knowledge about the SCA and does not require that knowledge as the goal is to abstract that detail from the user. The ideal solution is a model-to-model transformation from SCA to general UML and tight linking between the two tools, if possible using same user interface. Ideally the engineer's changes to the SCA diagram will translate into the model in the UML tool directly and vice-versa.

As mentioned before, the code representation of a component contains functional and infrastructure code. This infrastructure code can be completely handled by the SCA tool, the UML tool is used to model the functional parts of the component. Hence the model-to-model transformation should take an SCA model and create a UML model of the functional code only, abstracting away (or hiding) the infrastructure code.

This way of working can be extrapolated to extend not just general UML tools, but also signal processing tools like The MathWorks/Simulink. Again, with this example there is infrastructure code and functional code (in this case signal processing). The latter is handled in The MathWorks Simulink.

For now, the available SCA tools generate source code that implements the infrastructure parts that can be combined with the functional code that is implemented in the generic UML tool (or in a signal processing tool). SCA tool vendors can be contacted for more details on this.

2. DEVELOPMENT PROCESS

In this section, we shall explain the development process in detail. First we explain which tools were used and how they are related. We talk about modeling, generation and how to iteratively model-a-little, code-a-little and test-a-little.

Finally, we touch on CORBA and SCA as well as porting and re-usability.

2.1. Integration and Flow of Artifacts

The waveform development process followed by TTCS is a 3-step process as shown in figure 1. Zeligsoft CE is used to model the SCA waveform architecture consisting of a number of SCA components. Each of the components is then generated into a Worker class (the skeleton for the functional code) and infrastructure code (the implementation of the SCA-required behavior).

The Worker class is then reverse-engineered into Rhapsody using Rhapsody's reverse engineering functionality. Each component will have its own Rhapsody model. After functional code has been inserted, C++ code can be generated in Rhapsody. Finally these codes are compiled against a specific OE (RTOS and CORBA ORB) by Zeligsoft Code Generator to create the executable for the component.

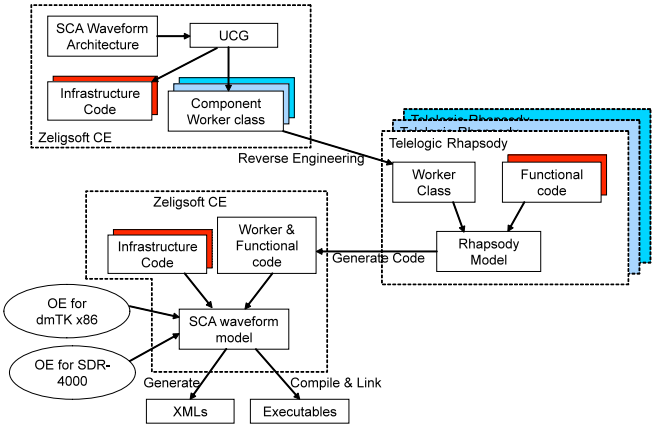

Figure 1: Tool Chain Integration Process

2.1.1. Step 1: Waveform Architecture

Waveform modeling is done in Zeligsoft CE. In this stage, the waveform components are defined and assembled into an application. Components have ports to communicate with other components. The WNW contains ten components performing functions such as ad-hoc routing, IP, MAC, PHY adapter, and so forth.

Figure 2 shows the representation of a set of collaborating components in Zeligsoft CE. Each black filled square box represents an SCA uses port and each white square box represent an SCA provides port. Both ports are based on an interface that indicates the messages that flow through it. The lollipop symbol at the top of the component indicates the SCA CF::Resource interface that represents the infrastructure interfaces. Ports are connected by connectors, shown as the black edges between ports in the diagram. The connections are owned by the encompassing application, not by the individual components.

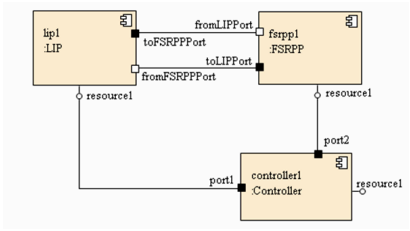

Figure 2: Waveform Model in Zeligsoft CE (Simplified Model)

The component definition is translated into source code by Zeligsoft Code Generator. The source codes include both the worker and infra-structure code to handle the components' properties configuration, ports functions, and SCA life-cycle functions. These codes are independent of the OE. An SCA component can have numerous implementations, one each for a different OE. The generator also generates makefiles for each of these implementations to compile and link these codes into executables.

2.1.2. Step 2: Functional Behavior In Rhapsody

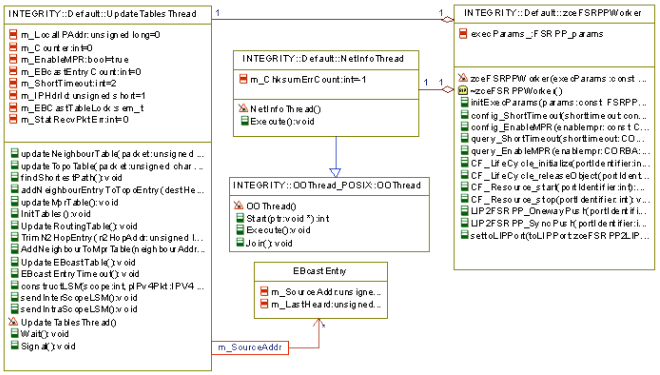

Figure 3: Rhapsody Model of a Component (Simplified Model)

Telelogic Rhapsody was used to develop the functional behavior through UML diagrams.. The Worker class of a component is reverse-engineered to create a class in Rhapsody. In this project, we developed the functional code initially for MS-Windows XP running on an x86 PC. This code was then ported to INTEGRITY (running on the SDR-4000) with slight modifications (implement macros, abstraction functions/class) to handle the differences between Windows XP and INTEGRITY.

As shown in figure 4, the uses port is represented in Rhapsody as a private association with the name of the port suffixed with an underscore (*toLIPPort_*). The association is to a standard C++ class that implements the uses port. This class abstracts the worker from the CORBA implementation of the uses port. The initiation of connections between ports

is also not handled in the worker. Rather it is handled in the infrastructure code that implements CF::Port, which is not represented in the model.

The functions for the provides port are represented in figure 4 as a function with the port IDL module and interface name prefixing the function name (such as *LIP2FSRPP_OnewayPush* function). The worker inherits from an abstract interface that represents the functions in the interfaces on the provides ports.

As mentioned, this pattern provides for complete separation of the functional code from the CORBA and the SCA infrastructure code, thereby lessening the cognitive load on the engineer. Further detail about the code pattern is beyond the scope of this paper and can be obtained directly from Zeligsoft.

Some of the infrastructure behavior is terminated in the worker. For example, the CF::Resource interface has start, stop, configure and query functions. These functions are as much as possible implemented in the infrastructure code and then forwarded to the worker so that the user can add behavior (to start signal processing for example). After which the user can now add detailed level code and additional contents to the UML model in Rhapsody and generate it into source code.

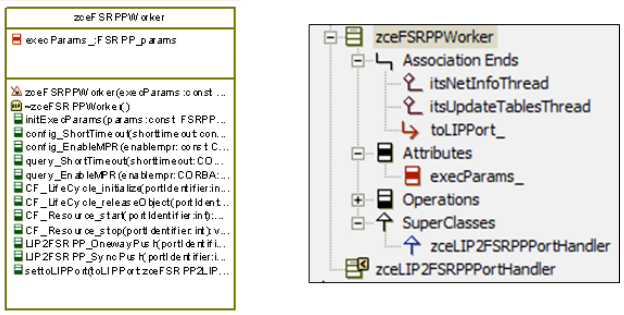

Figure 4: Worker Class in Rhapsody

2.1.3. Step 3: Compilation

As mentioned, makefiles are generated from the component and the implementations in the SCA model. These makefiles are then used to compile the codes for MS Windows XP using ACE/TAO and executed in the HARRIS dmTK CF and for INTEGRITY using OIS ORB*express* with the SCARI++ CF (SDR-4000). Note that the code for a component is the same for all targets. Only the makefiles are different for each target — they are used to select the appropriate compiler, linker and so forth.

The other benefit of the generated makefiles is build avoidance. During the first compilation, all the code of a component is compiled. In next iteration, only the source files that have changed will be compiled. This saves time and makes build-a-little, test-a-little workflows faster.

2.2. Iterative Development

The waveform models are built incrementally. That is, an initial architecture model is developed and then, in multiple steps, parts of the behavior are incrementally implemented. As the engineers learn more about the waveform, the radio and the SCA, they elaborate on this model and re-factor where necessary. This means that modifications will be made to the waveform model that needs to be propagated to the Rhapsody model. Modification must be done in both the Zeligsoft CE waveform model and in Rhapsody model. The engineer can choose to modify both models manually, or to modify in CE, re-generate code, and reverse-engineer into Rhapsody in order to update the Rhapsody model. Modifications can require changes to:

1. interfaces
2. properties
3. ports on a component
4. implementations of a component
5. components instantiated in the waveform
6. connections in the waveform

Modifications 1 – 3 require changes in the Rhapsody model, whereas modifications 4 – 6 can simply be handled in the SCA model in Zeligsoft CE.

The fact that changes need to be manually propagated is clearly something that can be improved upon, See the section titled “Future Improvements” later on in this paper.

2.3. Platform Independence and Reusability

One major achievement of the SCA is platform independence, both in the sense of radio hardware, as well as CORBA ORB, RTOS and CF. In this section, we shall elaborate on how this has been realized.

As reported before, two types of test platforms were used in the project. The first setup is a host environment consisting of x86 compatible PCs operating in MS- Windows XP. Installed on the platform is the HARRIS dmTK CF and ACE ORB (TAO) CORBA. The second setup is the SDR-4000 embedded system. The SDR-4000 processing engine supports a Xilinx Virtex-4 FPGA, a TI TMS320C6416T DSP and a Freescale MPC8451E GPP. The OE consists of INTEGRITY RTOS, SCARI++ CF and ORBexpress RT CORBA.

2.3.1. CORBA

Two separate CORBA ORBs were used. ACE ORB (TAO) on the development host and ORB*express* RT on the SDR-4000. The ORB functions are encapsulated in the

infrastructure code. The switch between the different ORBs is done by the setting the compilation flags in the makefiles. The engineer is not exposed to CORBA, the worker class is completely encapsulated. The benefit is that the engineer does not have to be well versed in CORBA, and these codes are portable by design.

One benefit of this separation between infrastructure and functional code is that it is a trivial task to replace the CORBA ORB with a different communication framework at a later stage, if required for performance or other reasons. A different communication framework could be deployed by direct function calls, RTOS messaging or streaming over Serial RapidIO (SRIO), or by the FlexFabric contained in the Spectrum Signal Processing boards.

Additionally, this separation allows for re-use of the functional code outside of the context of the SCA. This is something that is both possible and desirable, but has not yet been attempted by the project team.

2.3.2. Operating Environment

Two separate RTOSs and SCA CFs were used: Harris on Windows XP and SCARI++ on INTEGRITY. The waveform was easy to move and the engineers always maintained two execution environments during the timeframe of the project.

2.3.3. Platform

The host-based platform clearly did not have sufficient processing capability to run the PHY layer of the network. To still be able to execute useful scenarios, a test-bed was set up using the QualNet IP Network Emulation (IPNE) module attached the Windows XP host. This test bed allowed the project team to create a multi-hop radio scenario. Once the software was tested, it was then moved to the SDR-4000 with a fully implemented PHY layer running on the DSP processor.

Porting of the waveform only required a re-compilation of the functional and infrastructure code for the different OE, and then loading it onto the SDR-4000. The Rhapsody model remained exactly the same for this.

3. RESULT

3.1. Time Saving

With the use of this development process, protocols can be developed in a generic format that can be easily targeted for a specific system in a later stage. Porting effort for future systems could be reduced by 70%. Moreover, key codes could be re-used, saving 50% of design effort for future projects. In the WNW development, the porting of protocols codes from host OE to the SDR-4000 system took less than 3 weeks. This freed up substantial time for the engineers to perform value-added activities such as optimization, testing and verification.

3.2. Improved Engineering Resource Utilization

Deploying this process eliminates the requirements to build a large waveform development team. There is no requirement to have in-house experts in the CORBA and SCA domains. Moreover, using a common resource UML model tool like Rhapsody, project management can allocate or assigned engineers to other software development projects when necessary.

3.3. Performance

Figure 5: Port Abstractions for CORBA-Worker

Zeligsoft Code Generator is capable of generating code that has small overhead. The project team's experience shows that CORBA port communications are the most frequently run operations compared to other CF operations. Packets generated externally or by the Routing component need to be sent downstream to the PHY layer, and similarly the packets received by the PHY layer need to be sent upstream. These are CORBA operations that are sent through ports on the components. Figure 5 shows the relationship between the two components' ports with the port abstraction codes in between the workers on each side and CORBA in between the two. This is the code path that is executed during message passing.

On the sending side the worker calls an operation on the uses port abstraction code as a normal C++ operation, data is passed in by reference. This operation checks an attribute to see if the port is connected and then calls a CORBA operation to pass the data.

On the receiving side provides port code is the implementation of the CORBA servant. This implementation again checks an attribute to see if the worker is registered, if so, it calls a C++ operation on the worker and passes the data as a reference.
In both the sending and receiving cases the overhead is limited to a single if-statement, a function call and the passing of data by references. This is negligible compared to the overhead of inter-process or even inter-processor communication.

3.4. Quality

Quality of the code is improved by using a UML model. The UML model is easier to interpret and maintain by engineers, especially if multiple engineers are collaborating on the same project. Code generation introduces fewer errors, resulting in better overall code quality.

4. CONCLUSION

We have presented an integrated tool chain that reduces time, effort, and risk in developing and porting waveforms and improves code reusability. It clearly lowers the technology hurdle in developing an SCA waveform. The development process leverages modeling, code generation and automation of the two tools to achieve all the above mentioned benefits.

Code generation is often considered to be less optimal then hand-written code. In our experience, cost is incurred as soon as the engineer steps away from writing assembly-code. Abstraction always has a cost, but we argue that for our R&D project, such cost is overwhelmingly compensated by the time savings that can be achieved by using modeling and automatic generation.

The project was delivered on-time, on-quality and within the budget, which TTCS partially attributes to the acceleration provided by the modeling tools selected. TTCS is inclined to follow a similar development approach in upcoming projects.

5. FUTURE IMPROVEMENT

The above development process is targeted towards GPP components. It can be applied to DSPs, however, DSPs are optimized for signal processing and control code has a tendency to flush the pipelines with frequent jumps, thereby impacting the performance of the processors. Hence care should be taken to achieve proper optimization when using general purpose UML tools. A development process for DSP would also benefit from an integration with a signal processing tool (e.g those that are supplied from vendors like The MathWorks).

As for FPGAs, it may be possible to apply the UML tool chain integration for FPGA development. A technological project sponsored by THALES demonstrated the use of UML to develop an FPGA component using Rhapsody and generate code for SystemC [3][4][5]. This potential capability may be investigated further to enable more automation on the development process for waveform development.

Lastly, the integration between the SCA modeling and the general UML modeling tool is done manually. There is clearly opportunity for improvement here. Zeligsoft has already done work on this together with Telelogic, more information on this collaboration is available through Zeligsoft.

6. REFERENCES

[1] Zeligsoft Component Enabler in Action: Integrating Zeligsoft CE and Telelogic Rhapsody for rapid implementation of SCA compliant components available through www.zeligsoft.com

[2] http://en.wikipedia.org/wiki/Eight_queens_puzzle

[3] http://www.martes-itea.org

[4] T. Arpinen, M. Setälä, P. Kukkala, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, “Modeling Embedded Software Platforms with a UML Profile,” Forum on specification and Design Languages (FDL'07), Barcelona, Spain, September 18-20, 2007.

[5] P. Andersson, and M. Host, “UML and SystemC - a Comparison and Mapping Rules for Automatic Code Generation,” Proceedings of the Forum on specification and Design Languages conference (FDL), Barcelona, Spain, September 18 - 20, 2007.