Evaluation of BridgePoint Model-Driven Development Tool in Distributed Environment

Nenad Ukić
Ericsson Nikola Tesla d.d.
Poljička cesta 39,
HR-21 000 Split
+385 91 365 5970
nenad.ukic@ericsson.com

Marijan Zemljic
Ericsson Nikola Tesla d.d.
Krapinska 45, P.O. Box 93,
HR-10 002 Zagreb
+385 91 365 3279
marijan.zemljic@ericsson.com

Ivan Markota
Ericsson Nikola Tesla d.d.
Poljička cesta 39,
HR-21 000 Split
+385 91 365 5760
ivan.markota@ericsson.com

Pál L. Pályi
Ericsson Hungary
Irinyi József u. 4-20.
1117, Budapest
+36 1 439 5460
pal.palyi@ericsson.com

Domonkos Asztalos
Ericsson Hungary
Irinyi József u. 4-20.
1117, Budapest
+36 1 437 7952
domonkos.asztalos@ericsson.com

Abstract: In this paper, we describe experiences gained in the project that had the goal to evaluate the Model-Driven Design methodology and BridgePoint tool in distributed development environment. Although design object was not of crucial importance for the project, the Diameter protocol was chosen because of its relevance for the corporation. Experiences gained from project are neither too positive nor too negative. BridgePoint does not allow definition of system-wide class model, which greatly affects modeling possibilities. Also, it misses some of the important version control features such as comparing and merging models. However, development process is, after initial learning phase, much faster than in usual development process. Possibility to verify the model behavior before code generation turned out to be very useful feature. During the project, some of the good practices as well as issues for distributed model-driven development using BridgePoint tool are recognized. The tool may yet not be ready for large-scale projects but it surely has great potential for drastically improving productivity of small to medium development teams.

1. INTRODUCTION

Within Ericsson Corporation, there are several target platforms / environments used to run and test telecom software. The same or similar functionalities are often required for multiple platforms. However, software developed using traditional programming languages and methodologies, for one specific target environment, often cannot be reused in other environments. In such situations, it is not rare that the same functionality is developed more than once. There is obvious need for cross-platform, programming language independent software development methodology and tool.

Model-Driven Design (MDD) based on the Executable UML (xtUML) [1][2] paradigm is relatively mature technology with very high potential. The main idea behind this paradigm is to introduce higher level of abstraction into the development process and facilitate significant time and resource savings. It is based on reuse of common Platform-Independent Model (PIM) across different platforms. PIMs can be easily translated into different platform and programming language specific code by using appropriate model compilers.

Already several prototypes have been executed within Ericsson to examine the model driven development (MDD) using the BridgePoint [3] xtUML tool. Most, or all, of these have been small, one-person prototypes focusing on the resulting code quality and design efficiency for one designer. All of these prototypes have been positive. The next logical step was to test the scalability of the development paradigm and the tool through a slightly larger prototype involving an international, geographically distributed team in the project called xDIA.

Good practices, issues and tool limitations that are recognized during project are presented in this paper across several chapters: First, we discuss approaches for system-level modeling. BridgePoint class-modeling issues are presented in third chapter. In fourth and fifth chapters, we described BridgePoint behavioral modeling capabilities and possibilities to interface with external code. Model verification is explained in sixth chapter. Next two chapters deal with code generation and version control. Ninth chapter gives more details about Diameter protocol, a modeling subject. In the end, future work and conclusions are given.

2. SYSTEM-LEVEL MODELING

Selecting appropriate architecture for the system-level model is probably the most important decision during modeling process. Problem of system-architecture selection
hides well-known domain separation issue. In BridgePoint, model may be organized in separate components (domains) or separate subsystems within single component. System architect should be familiar with tool limitations and consequences they have in each approach. The main driver towards single-component approach is shared class model, which greatly simplifies communication between layers. The main argument for the multi-component architecture is clear interface between components and reusability it brings. That was the key argument for xDIA project to use multi-component approach, although that decision was not an easy one.

Besides selection of components, system level modeling with BridgePoint usually includes defining interfaces between components, data structures that will be passed across those interfaces and wiring components into component configurations. Good practice is to separate component definition from its usage by using library and configuration components packages. Once defined, components may be re-used (utilizing components references) in several different component configurations.

After components and configurations are defined, most of the changes on system level happen on interfaces. Interface modifications have impacts on all components and configurations that use that interface, which in turn may produce severe conflicts in the project. Thus, the component interfaces should be defined with great caution and should not be changed too often in later phases of the project. The real distributed (concurrent) modeling can be started after the system-level model is agreed and committed to the version control server.

3. CLASS MODELING

Class modeling is central and most time-consuming phase of the overall modeling process. It requires good knowledge of the subject matter, which has to be formalized in the models using object-oriented abstractions such as classes, attributes and associations. Although the task of producing class models may look simple, it is not. Biggest challenge here is to change perspective, or mind-set, from traditional software design to modeling PIMs, which are not concerned with any specifics of a target platform. With xtUML method, these implementation issues are left to model compiler, which translates PIMs into code. In addition, the goal of the PIM is not to model all the knowledge on the subject matter (i.e., it should not be used as ontology); rather it should reveal associations between classes, which will be later used in action language for actual manipulation of class instances. Since class modeling precedes the action language specification, this is sometimes very difficult to detect. Furthermore, class modeling with xtUML is somewhat different than with standard UML. One example is inheritance, one of the main characteristics of object-oriented UML design, which is not directly supported in xtUML. Rather, the xtUML uses the more generic generalization concept, which could be later translated to language specific inheritance concept using appropriate model compiler.

Creating class diagrams is closely related with two other modeling phases: the system-level modeling and the creation of the state diagrams. Relationship with state diagrams may be obvious: if the class diagram is oversimplified, then the state diagrams within classes may turn to be rather complex. On the other hand, if the class diagram is precise and detailed enough, state machines will probably be quite simple. Relations between class modeling and system-level modeling are not obvious and are result of limitations in the BridgePoint tool. Main problem is that there is no possibility of defining system-wide classes, i.e., classes that are visible to all components in the system. This means that a class instance created in one component cannot be passed to another component in a simple way. This is serious limitation which developers should be aware of during system-level modeling phase. Alternative for passing complex data between components are user-defined data types (structures) which can be defined with system-wide visibility. Once data structure is passed across component interface, it could be mapped to class instance in another component. Data structures can also be used to represent data within component as attributes in some classes. However, having complex data structure as attribute in a class hides complexity of the data model that is usually exposed in class diagram and is not considered as good practice.

4. BEHAVIORAL MODELING

The system-level model and the class diagrams are concerned with defining the model structure. However, executable UML models should be able to be executed. Therefore, the definition of the system behavior is also very important phase of the modeling process. For that purpose, state machines and actions are used.

The state machines are used to define the life cycle of instances of a given class. The modeling person has to detect the possible states in the instance lifecycle, the logical transitions between those states and associate those transitions with some domain events. Since it is very difficult to detect all relevant transitions, especially if a state machine contains considerable number of states, producing state machines is usually error-prone process. Therefore, each state machine diagram must be checked against the so-called state-transition table, which is a table representation of the state machine. This table contains appropriate actions for all possible combinations of states and events, including those that are not formally expressed in the state machine diagram.

Actions in BridgePoint are specified with the use of the Object Action Language (OAL), a very simple and
lightweight language that enables easy navigation through class diagram as well as manipulation of class instances and associations. Although actions may be stated on several different “places”, the most natural one is within the states of a state machine diagram. Other “places” where OAL code is usually placed are the class, instance or system wide operations as well as the interface implementations called ports. Although BridgePoint misses some common editor functionalities, such as auto complete or code formatting, writing the action language code is pretty straightforward and probably the simplest part of the modeling process.

5. INTERFACING WITH HAND-WRITTEN CODE

Interfacing with already existing code represents the strong requirement in the telecommunication domain, which still heavily relies on proven legacy solutions. Having legacy code that integrates with the code generated from PIMs is the most likely case in future telecom solutions. Also, beside the fact that not all functionalities of traditional programming languages are fully supported in current version of the BridgePoint tool (e.g., data type conversions, string manipulations, bitwise operations etc.), there are also some functions that are considered platform specific and, as such, must partly be implemented using platform specific languages and methods. Therefore, it was imperative to test the mechanism of the BridgePoint tool for interfacing with the hand-written code. Number of modeled entities (external entities, system functions, classes, class operations or component ports) can be marked as external. Those items will be omitted during code generation process.

External code may present pre-existing legacy code that must be re-used or functionality that is not possible or convenient to achieve in the model. The important difference is that legacy-code cannot be altered in any way and must be used as is. Therefore, interfacing with pre-existing legacy code presents much greater challenge. It often requires development of platform specific adaptation layer, which adjusts generated code to already existing legacy code.

In xDIA project, external code is used for different reasons but it was not used as pre-existing legacy code. One example is message codec that required intensive use of bitwise operations that are not well supported by BridgePoint tool. Other use of external code was for network transport management, which is platform-specific issue and should not be part of PIM model. Although the scope of the prototype did not require interfacing with already existing legacy code, it provided an invaluable experience for any possible future requirements regarding this problem.

6. MODEL VERIFICATION

BridgePoint provides industry-standard testing and debugging environment in its debugging perspective. Core of that perspective is the Verifier, an engine for execution of OAL code, which enables designers to verify the models before they are converted into code. Debugging perspective provides functionalities for executing models with graphical presentation of transitions in state machines, setting breakpoints, observing printouts, checking variable values, etc.

BridgePoint model testing should be prepared during system-level modeling phase. It is a good practice to define one test component for each component in the system. Test component simulates “rest of the system” to the component under test and may contain chained automated regression tests. The two components are “wired” to each other into a component test configuration, which can be executed in verifier. This practice enables the independent testing of each component. Another good practice is to define a test subsystem for each subsystem used in a component. Test subsystems are often used for unit testing of the functionalities in subsystems.

The Verifier engine provides the possibility to interface the model (OAL code) with Java code in order to implement functionalities that are chosen not to be modeled. In this way models are complete and can be executed in verifier the same way as real generated code. This functionality enables model testing in a real environment before any code is generated.

7. CODE GENERATION

Once the model has been tested and verified, it is ready for code generation phase. We used Ericsson’s proprietary model compiler that generates code in C programming language. Before actual code generation, it is necessary to define marks for the model compiler. Marks are used to fine-tune the model compiler by providing details and exceptions in the code generation process. After defining marks, process of code generation usually takes several minutes. Most of the marks we used can be grouped into one of the following groups:

- Marks for wiring components across interfaces
- Marks for interfacing with external code
- Definition of common data types
- Definition of initial memory pool size for class instances

The code generation process also provides nice statistics on the model complexity. In our case the figures are: 54 classes, 32 functions, 162 class operations, 9 state machines and 5942 OAL statements. This summary includes not only DBP/SH model, but also the test components and test classes as well.
8. VERSION CONTROL

The xDIA project used SVN based version tracking. Main version tracking strategy in the xDIA project was to mitigate the risk of possible conflict situations. According to our experiences it is very difficult, if not impossible, to resolve complex conflicts in the BridgePoint tool. Unlike the textual source code, the BridgePoint models cannot be merged automatically. All conflicts have to be resolved manually, item by item. With limited model-comparison feature, process of model merging and resolving conflicts is very laborious and time-consuming.

Best practice for avoiding conflicts is to clearly separate modeling work between developers. Experiences from the xDIA project have shown that despite clear work separation, conflicts cannot be completely avoided. This is particularly true in the early phase of the project when system-level modeling is done. Models produced by BridgePoint are organized in hierarchical manner. System-level modeling affects the highest-level xtuml files (version controlled with SVN) and this necessarily produces conflicts if changed concurrently by other team members.

Due to a limited model merge and comparison features, it is practically impossible to merge concurrent project branches. Therefore, distribution of work should be done in single development branch, which makes BridgePoint unsuitable for larger development teams.

During model merging process, it is often the case that concurrent changes on two different modeling entities produce a conflict. This happens if entities are written in same file in file system. The problem may be alleviated by applying finer granularity to xtuml files (use of larger number of smaller files), which would result in lower number of conflicts but would not solve conflict-resolution problems.

9. DIAMETER CASE STUDY

Probably one of the best candidates for software reuse in Ericsson Corporation is the Diameter protocol stack. There are at least four different implementations in use, each for different target platforms. Although design object was not of crucial importance for the project, the Diameter protocol was chosen because of its relevance for the corporation.

The Diameter protocol is an application layer protocol used for authentication, authorization and accounting services. It consists of the Diameter Base Protocol (DBP) [4] and Diameter applications, upper layer protocols that use DBP for common functionalities. Since the project was limited with time and resources, only a subset of DBP functionalities is implemented. Those include connection management, routing management and message verification.

As DBP cannot operate on its own, for demonstration purposes, Sh protocol [5][6] was selected as upper layer Diameter protocol. One of the reasons for choosing the Sh interface is its relative simplicity. It is used in IP Multimedia Subsystem (IMS) between the Application Server (AS) and the Home Subscriber Server (HSS) (see Figure 1) to retrieve and update subscriber data. It was also implemented with reduced scope, i.e., all optional information elements were excluded from the modeling. In order to show functionalities of the Sh/DBP protocol stack, basic AS and HSS functionalities were also modeled (See Figure 1).

Since DBP and Sh upper layer both use diameter messages, they had to share knowledge about it. That fact was driving architectural choice towards single component, but DBP reusability requirement was main reason Sh and DBP are put in separate components. That resulted in rather complicated communication between the two components with several parallel representations of Diameter message. Architectural choice as well as the fact that Diameter is quite flexible protocol, made the modeling of DBP to be quite a challenge.

---

**Figure 1** Architecture of xDIA protocol stack
10. FUTURE WORK

The xDIA project is continued in the follow-up MaxDIA project, which aims at deploying enhanced xDIA Diameter stack in a real production system. One of the key issues that will be addressed is integrating with complex telecom-domain legacy code. Other future work may also include comparison with other MDD tools available.

11. CONCLUSION

Using SVN with BridgePoint with several geographically distributed teams requires some experiences and practice by all team members. The work separation should be clear as with any distributed software project, but when applied to BridgePoint this imperative dramatically gains on strength. Before using in a real production-level environment, the vendor of the BridgePoint should resolve all minor, but very annoying, bugs that were discovered during the project. There are also some features and possible improvements that would facilitate better utilization of the tool. The most important are definitely support for system-wide classes, introduction of object-oriented class inheritance as well as model merging and comparison improvements.

Importing to and from standard XML Model Interchange (XMI) is also a feature that would be very welcome. It would enable at least partial reuse of existing models, not necessarily produced with BridgePoint tool.

The xtUML methodology and the BridgePoint tool are very promising technology, they contribute to substantial reductions in lead-time and resources used in the project. This is especially true if multiple platforms are to be supported because PIM models can be easily translated into code by using different model compilers. This is the reason why substantial efforts should be given in the future to developing new and improving existing model compilers.

12. ACKNOWLEDGMENTS

During our project, we were mentored and had great support from John R. Wolfe from Mentor Graphics and Toni Siljamaki from Ericsson AB, Sweden.

REFERENCES

[5] 3GPP TS 29.328 V9.2.0, IP Multimedia (IM) Subsystem Sh interface; Signalling flows and message contents