APPLICATION OF MODEL DRIVEN DEVELOPMENT TO CONTROL SYSTEMS

Klemen Žagar, Anže Vodovnik, Jožef Stefan Institute and Cosylab, Ljubljana, Slovenia

Abstract

Model Driven Development (MDD) is a technique employed by software architects that allows them to first model the essence of a system, and then uses that model for further development. In contrast to non-MDD approach, the model is not just one of the forms of the system's documentation, but a central artifact from which all the others, such as program source code, documentation, reference manual, etc., are derived in an automated or semi-automated fashion using generators and Computer Aided Software Engineering (CASE) tools. In this paper the state-of-the-art of MDD is presented. First, a brief overview is given of the Unified Modeling Language (UML), which is the *de facto* standard for representation of models. Then, the evolution of CASE tools in the past two years is outlined. Finally, our work on XML-centric code generation tools and their application to control system development (e.g., the ALMA Common Software collaboration with European Southern Observatory) is presented.

INTRODUCTION

There is nothing new about the use of models in software engineering. In fact, software itself is a model of reality, e.g., for the purposes of simulation, prediction or analysis. Ultimately, such model is represented in machine-readable code.

Because the only things machines are really good at are basic arithmetics and pushing data back and forth, real-life issues require a great deal of effort to explain using such a limited – and unnatural – conceptual system. To make the process of explaining to machines (i.e., *software engineering*) easier, high-level programming languages were introduced for modeling lower-level ones, and compilers were developed whose task was transformation of the higher-level *models* to the lower-level ones (e.g., from the Java programming language to Java P-code, or from the C++ programming language to assembly language, and finally machine code).

Ideally, the software model would be equivalent to the model through which we, humans, perceive reality. In such a model, one would not talk of *functions*, *variables*, and, god forbid, *pointers*, but instead of *power supplies*, *bending magnets*, *beams*, *antennas* and *stars*. This is where **Model Driven Development** (**MDD**), and its cornerstone, the **Model Driven Architecture** (**MDA**), enter the scene.

DEVELOPING SOFTWARE FOR CONTROL SYSTEMS

Software development in the field of control systems is very much like software development in general. Development disciplines, such as *requirements analysis*, *design*, *implementation*, and *testing* are all present, and activities performed within each are very much as everywhere else. Also, standard technologies are employed: *client/server*, *three-tier architecture*, *Java*, *web servers*, ...

What is specific, however, is the *problem domain* which the software targets. Typical software found in business is concerned with manipulation of records in databases: insertion of *purchase orders*, generation of *invoices* based on them, ... In control systems, data – typically information related to logging and archiving – is only part of the story, as the basic entities that control systems are concerned with are the devices that are subjected to its control.

Initial Construction

Before software is constructed, its general structure (the *high-level architecture*) is defined and documented. Afterwards, design details are specified. Finally, the design is implemented.

At the end of construction, the delivered software and its *blue-prints* are only rarely synchronized. This is mostly because synchronization requires a lot of effort and tenacity, habits, that one tends to abolish quite fast due to lack of time, discipline or both.

Maintenance

What typically happens when software or any other product of engineering is made to face the challenges of real life is that some things don't quite work as expected. In some cases, even the functionality which was the project's mission doesn't work, or it dangerously interferes with other subsystems. Defects in software, also known as *bugs*, are gradually resolved during *software maintenance.*

Another reason why software needs to be maintained is because its environment or requirements change.

In any case, accurate blue-prints are highly desirable. Again, any modifications to the software should be affected at the level of design documents, or else further maintenance will be becoming more and more difficult, possibly resulting in a complete rewrite of the software.

MODELS

The purpose of a model is to describe reality. Models are typically simplifications, and focus on only a particular aspect of what they model. For example, a physical phenomenon can be simultaneously described using a thermodynamical, electrical and mechanical model.

MODEL DRIVEN ARCHITECTURE

Model Driven Architecture tries to accomplish two goals: easier ways of synchronizing the blue-prints with the software, and a taxonomy of models along with transformations between them.

The term *Model Driven Architecture* was coined and precisely defined through various specifications by the Object Management Group (OMG) [1].

Platform-Independent Model (PIM)

The *Platform-Independent Model* is a model in which the intricacies of concrete implementation platform are no longer visible – instead, only the models of entities from the *real life* are present.

In the domain of control systems, one would talk about devices, what their properties are and how they relate to each other.

Ideally, one would also be capable of describing *behavior* of a system using a platform independant model (such an example is shown in Figure 2). Unfortunately, a *platform independant* language capable of describing actions is not yet standardized, but this effort is under way in the form of UML 2.0 specification (a draft version as of this writing).

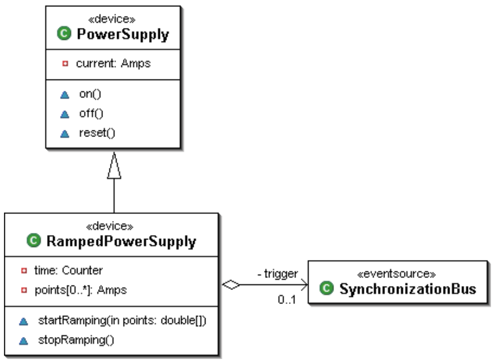

Figure 1: The Platform-Independant Model (PIM) of a power supply and its specialization that also supports ramping via a trigger (the synchronization bus).

Platform-Specific Model (PSM)

The *Platform-Specific Model* models the software. As such, it depends on the platform upon which software is built. Thus, if software were written in C++, the PSM would be different than if it were written using Java.

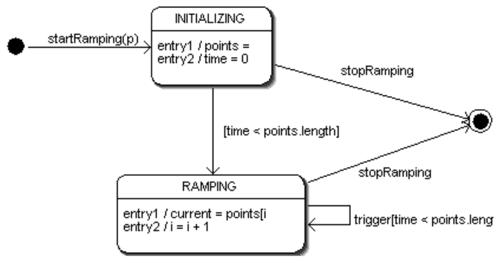

Figure 2: A state diagram describing the behavior of a ramped power supply from Figure 1.

Ideally, one would be able to automatically produce the PSM model from the PIM model. This transformation could be conducted with a (CASE) tool, whose knowledge would encompass:

- The structure of the PIM model (the *meta-model*).
- *Design patterns* to be used in the PSM model.

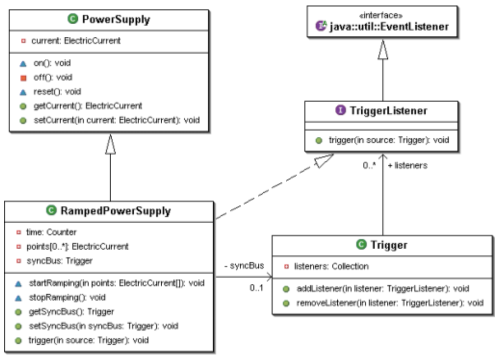
PowerSupply
current: ElectricCurrent
on(): void
off(): void
reset(): void
getCurrent(): ElectricCurrent
setCurrent(in current: ElectricCurrent): void
«interface»
java::util::EventListener
TriggerListener
trigger(in source: Trigger): void
0..*
+ listeners
RampedPowerSupply
time: Counter
points[0..*]: ElectricCurrent
syncBus: Trigger
startRamping(in points: ElectricCurrent[]): void
stopRamping(): void
getSyncBus(): Trigger
setSyncBus(in syncBus: Trigger): void
trigger(in source: Trigger): void
- syncBus
0..1
Trigger
listeners: Collection
addListener(in listener: TriggerListener): void
removeListener(in listener: TriggerListener): void

Figure 3: The platform specific model of the (ramped) power supply, whose platform independant model is depicted in figure 1.

The Unified Modeling Language

Nowadays, the **Unified Modeling Language** [2] (UML) is the de facto standard of the software industry used to describe models. The standard defines the notation of various kinds of diagrams, among them:

- **Class Diagrams** describe the entities of the system, their attributes (associated data) and operations (behavior). Figures 1 and 3 are examples of class diagrams.
- **State Diagrams** (e.g., Figure 2) describe behaviour of a system in terms of states and transitions between them, as well as actions that occur once inside a state, or when entering or leaving it.
- **Activity Diagrams** for describing workflows.
- **Sequence Diagrams** for defining the sequencing of interactions between entitities.

- **Use Case Diagrams** for specifying the requirements from the user's perspective.

The most recent version of UML specification is 1.5, but 2.0 is in the process of finalization. Version 2.0's major improvement is specification of a formal, platform independant, language, which will be used for describing actions.

CASE TOOLS

Computer-Aided Software Engineering (CASE) tools enable developers (in particular system analyists and software architects) to create and maintain a high-level abstraction of the system. Typically, CASE tools support visualization of various aspects of the software. These days, the notation most commonly supported by CASE tools is the UML.

IBM/Rational XDE

One of Rational's (now a subsidiary of IBM) best known products is *Rational Rose*, which allows architects to create UML diagrams, and in some particular cases also provides them with *forward-* (producing code from diagrams) and *reverse-engineering* (producing diagrams from code).

The successor of Rose is Rational XDE [4]. It ships in two editions, one for IBM's WebSphere and Eclipse development environments (Java), and another for Microsoft's .NET. Rational XDE offers customizable code generation (also preserves user's modifications) and support for design patterns.

Omondo UML

Unlike Rational XDE, Omondo UML [5] is freely available (but not open source). All diagrams created with Omondo UML are closely bound to Java code, either at the class or package level. The tool assures that the diagram and the code are always up-to-date: newly added method will immediately appear in the diagrams, and vice-versa.

A very welcome benefit of Omondo UML is that it stores its models in XML files, conformant to the XML for Metadata Interchange (XMI) format. This makes it possible to programmatically manipulate the model, e.g., to generate code from it, or to make transformations upon it using widely available tools (XSL/T transforms, for example).

CODE GENERATORS

Once a model is available, it is, in principle, possible to *transform* it into working code. The transformation can be performed on either the platform independant, or the platform specific model.

Many *Integrated Development Environment* (IDE) tools contain so called *wizards*, which allow developers to enter several parameters, from which code is generated automatically. Wizards allow the programmers to get up-to-speed in a very short period of time, as typically the output of a wizard is an already executable program. The output of wizards serves as a basis which developers then modify to suit their purposes.

However, wizards do not take any steps to assure synchronization between the code of the model – as a matter of fact, they are even unaware of any model. Code generators exist, however, which are given a higher level model as input, and generate code that implements that model. To name but a few:

- *CORBA IDL compiler*, which takes the interface definitions as inputs, and produces CORBA proxies and stubs in a given programming language.
- Generators written using templating languages, such as *Extensible Stylesheet Language* (XSL, [6]), *Eclipse Modeling Framework*'s JET [7] or Apache Velocity [8]. Here, the developer has a lot more control over what is generated, but is still incapable of modifying the generated results without risking loss of his effort during re-generation.
- Advanced generators, such as the *Extensible Program Generator Language* (XPGL, [3]), or those embedded in Rational XDE, especially dedicated to generating code and preserving user's modifications.

CONCLUSION

Today, modeling techniques are used primarily as a means of designing and documenting the system. Unfortunately, it takes a great deal of effort to properly synchronize all the views of a complex model with the actual state of implementation.

The field where modeling techniques have been particularly successful are the embedded systems. This is mostly because the behavior of embedded systems can be well defined using state machines and Petri nets, which are fairly easy to derive from code.

REFERENCES

[1] Object Management Group, "Model Driven Architecture", `http://www.omg.org/mda`

[2] Object Management Group, "Unified Modeling Language", `http://www.omg.org/uml`

[3] K. Zagar, A. Vodovnik, "Program Generators and Control System Development", PCaPAC 2002, Frascati, Italy, October 2002

[4] IBM/Rational, "Rational XDE Developer Plus for Java"

[5] Omondo, "Omondo UML", `http://www.omondo.com`

[6] W3C, "Extensible Stylesheet Language (XSL)", `http://www.w3.org/Style/XSL/`

[7] *eclipse.org*, "Eclipse Modeling Framework (EMF)", `http://eclipse.org/emf/`

[8] The Apache Jakarta Project, "Velocity", `http://jakarta.apache.org/velocity`