Metamodeling for Medical Devices: Code Generation, Model-Debugging and Run-Time Synchronization

Juha-Pekka Tolvanen\textsuperscript{a}\textsuperscript{*}, Verislav Djukić\textsuperscript{b}, Aleksandar Popović\textsuperscript{c}

\textsuperscript{a}MetaCase, Ylistönmäentie 31, FI-40500 Jyväskylä, Finland
\textsuperscript{b}Djukić Software GmbH, Gärtnerstr. 17, D-90408 Nürnberg, Germany
\textsuperscript{c}University of Montenegro, Faculty of Science and Mathematics, Džordža Vašingtona bb, 81000 Podgorica, Montenegro

Abstract

Metamodels can be used to specify languages that capture the concepts and constraints of an area of interest. We describe a case and experiences on applying metamodeling for the development of medical devices. Metamodels are used to define Domain-Specific Modeling languages raising the level of abstraction in models close to the problem domain and producing formal specifications. Generators then read the models and produce executable code running in medical devices. The novel part of our approach, and particularly useful for device development, is extending the generation approach to model debugging and synchronization between a state of a model and a state of a program executed in a target runtime-system. This enables quick iterative feedback from the running code back to models, supports verification of the developed control logic, and helps to optimize the use of hardware resources.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Program Chairs

Keywords: metamodeling; domain-specific modeling; domain-specific language; code generation; medical devices

1. Introduction

Metamodels can be used to specify languages that capture the concepts and constraints of an area of interest, aka Domain-Specific Modeling (DSM) languages. We describe a case and experiences on applying metamodeling for the development of applications into blood separator machines. The prime reason for applying metamodels is
improving the quality of the applications, making development easier and more productive\(^1\). First, metamodels not only capture the domain terminology for a language but can also capture domain rules\(^2\). This makes impossible to specify with the language applications that are illegal, unwanted or otherwise undesirable – or at least inform developers on incorrectness of the models. Second, and thanks to the language rules, the models specifying applications of medical devices are formal and enable the use of code generators producing application code. Together with the platform this generated code can be directly executed in a device. The novel part of our approach – and particularly useful for developing applications into devices – is extending the generation approach to model debugging and synchronization between a state of a model and a state of a program when executed in a target runtime-system (RTS). These enable quick and iterative feedback from the running code back to models, support verification of the developed application logic and help to optimize the use of hardware resources.

We describe in the next section the metamodeling process and the language definition steps along with a sample model. In Section 3 we describe the use of generators producing the production code and in Section 4 the connection of metamodels and generators with target platform enabling model debugging and synchronization between a state of a model and a state of a running system. In Section 5 we describe our experiences and lessons learned and then provide conclusions.

2. Metamodeling and modeling

We started language definition from a set of already known and used controller concepts, such as Analog controller, Two-state switch, and Multi-state switch and then refined the language further by introducing more domain-specific concepts, such as Balance, Blood bag, Clamps and pipe, Timer, Device status and Blood weight. All domain concepts were specified into a metamodel by using MetaEdit+ tool\(^3\) and then instantiated immediately while modeling. This allowed us to quickly prototype and test the metamodel while it is been defined at the same time. Fig. 1 illustrates this incremental process of defining the metamodel: Left part shows a definition of one metamodel element called StepMotor with its four properties. Right part then shows the use of the StepMotor as language concept while modeling at the same time. Here ‘TopStepMotor’ is an instance of the StepMotor and it has values for the four properties.

Metamodel does not only specify the concepts of the languages, but it can also capture various rules and constraints of the domain. For example, such rules define how StepMotor can be connected to other domain concepts, how many times it can be connected, are connections directed, etc. We defined these rules using the formalism of the metamodeling tool\(^3\). In many cases it then became simply impossible to create functionality of medical device which is illegal, unwanted or otherwise undesirable, like allow StepMotor to control the direction of

---

Fig. 1. (a) Defining element of the metamodel; (b) using that element and instantiating the metamodel while modeling.
the blood Press. In cases where the domain rules cannot be checked at the modeling time, model checking can inform developer on incompleteness, inconsistencies and on possible errors in the models. In this way, it becomes cheaper to correct the specifications compared to finding and correcting the specification later in the development phase.

The language definition was finalized by defining a concrete syntax. For the notation we tried to follow good practices² and apply already known and used symbols as well as derive them directly from the actual domain. Fig. 2 shows an instantiation of one language used for Modeling Hardware Signals (MHS DSL). MHS DSL captures both the abstraction on hardware-specifics details and relations between hardware specific and general concepts. The hardware-specific signals using certain functions for transformation (Fig. 2, ‘virtSigExpr’) are virtualized. Pins and ports are linked to (i) variables used by control logic, and to (ii) software or mechanical switches (‘Switch 1’).

Fig. 2. An example of modeling hardware signals (MHS1).

Fig. 2 also illustrates the use of the language and how some of the other elements of the metamodel are instantiated. Four virtual signals denoted as VS1, VS2, VS3, and VS4 (right on the model) are defined and connected to the pins on the Raspberry Pi B+ development board. Pins 27 and 22 are used for the multiplexing of 8-bit number (gpio 10, 9, 11, 5, 6, 13, 19 and 26), while gpio 2 is linked to the VS1, over hwSig_1 hardware signal. Depending on the value of pins 27 and 22 (00, 01, 10, and 11) value of the 8-bit number is treated as VS2, VS3 or VS4. In the middle part of the figure, there are several labels (with red borders) aimed only at displaying values. Displayed values are actual values read from the board pins. Signals, which interpretation depends on the state of some other signals, are related to the pins using the relation denoted by rhombus symbol. Switches are depicted by symbols that resemble their appearance on a concrete medical device. Switch can be in one of the following states: 00(0), 01(1), 10(2), and 11(3). For example, VS1 virtual signal may be used as an indicator of the device state (active or stand-by). VS2-VS4 signals are flow rate, weight, and volume of blood in a test tube. VS2 signal is rather specific because its value is formed based on two separately read hardware signals.

The MHS DSL does not express operation types (e.g. read or write) as these are separate concern from signals. To specify control logic we provided two different ways: Integrating multiple languages and using code generators. On the language integration side, we defined a metamodel of function block diagram based on IEC 61131-3 and then integrated it by defining relation types and binding rules to other languages. We choose IEC 61131-3 for
function side as it is a host language to our target RTS enabling simultaneous model execution. The second way for specifying control logic relies on been defined as a part of code generator. For example, operation types like read_write and read_only are defined in code generators. Drivers are generated from a model like this in accordance to the specificities of function that is performed by a concrete hardware platform.

The approach using integrated modeling languages has the advantage of requiring minimal rewriting of existing generators created for function blocks. The approach based on generators does not require language integration but brings limitation in handling device variations as they are now fixed in the generator.

3. Generators for code, debugging and synchronization with runtime system

In model-based development, the main purpose of code generators is to produce various textual specifications from models, such as program code or configurations. Listing 1 shows generated definition of hardware signals (HWS) from model MHS1 (Fig. 2). CfD1 definition, which is part of a device configuration, is used as an input resource by the compiler, or it is sent to the RTS during run-time in order to predefine signals dynamically. This specification is used to map pins onto HWS, and vice versa. The right part of model MHS1 maps HWS to variables and provides means to compose variables from HW signals even when they are not available in the same time.

Listing 1: Configuration that describes hardware signals (HWS), output from model depicted in Fig. 2.

```
CfD1.hwSig_5 = &2
CfD1.hwSig_1 = (&27,&22=00)&10,&9,&11,&5,&6,&13,&19,&26
CfD1.hwSig_3 = (&27,&22=10)&10,&9,&11,&5,&6,&13,&19,&26
CfD1.hwSig_2 = (&27,&22=01)&10,&9,&11,&5,&6,&13,&19,&26
CfD1.hwSig_4 = (&27,&22=11)&10,&9,&11,&5,&6,&13,&19,&26
```

Generators may also be used for other purposes than producing production code. They can produce simulations from the same models, or produce early on the development process data, for example, for performance analysis needs. Generators may also produce test data, deployment data and documentation. Generators are also used in our case to integrate tools with auto-build: a generator not only produces the code, but integrates the results with the build process and executes the compiled code in the target RTS.

For the development of medical device applications, we extended code generators to play two important roles: Model debugging and synchronization between a state of a model and a state of a program executed in the RTS. Compared to traditional model-based code generation, this extension includes Action Reports (AR)\(^5\)\(^,\)\(^7\) which provide communication with the target RTS, applications for debugging and monitoring as well as operations that change a model/meta-model. ARs are used during the development flow as follows:

- When starting with an empty design, a model (and modeling tool) is connected to a RTS.
- On each action performed over a model (insert, remove, connect, disconnect, set value) control logic and meta-logic is generated and uploaded to a target RTS.
- From the models several application types are generated: drivers, control logic and meta-logic, applications for monitoring and debugging.
- When executed in RTS, modeling tools reports about execution status, such as consumption of hardware resources, performance and correctness of logic.
- Emulation and execution are basically the same processes and for both of them native code is generated in order to avoid any discrepancies between emulation and interpretation over a concrete RTS.
- A target RTS supports transactions and return to the previous model state is performed by a rollback operation.

ARs are defined both in the context of a sub-model and the whole model. This way ARs can use a single model state as a starting point and generate executable program code that is not necessarily complete, but performs planned subset of device functionalities. Focus on a sub-model also provides good performance and usability features, like allow frequent model changes and execute them as an optimized transaction.
4. Platform, drivers, virtual signals and target Run-time System

The granularity level of generated program code depends on a target framework, libraries, and operating system or it depends on the features of a device that execute control logic defined by the program. If the target framework includes specific functionalities of high-level abstraction then it is more likely to achieve 1:1 translation between domain concepts and the specific functions. On the other hand, if the framework mainly includes general-purpose functions then code generators will be more complex. We preferred the generation of specific functions from models that comprises platform-specific features, even modeling device drivers. Practical application of such an approach is possible because our compiler guarantees the execution performance on the level of optimized C++ code and also supports incremental generation and updating program code during execution on the RTS. We also generate drivers optimized for the concrete medical device rather than apply general-purpose drivers.

The target RTS is a part of the domain-specific solution for the development of medical devices that executes the code generated from models, i.e., it executes models according to the given rules for synchronization and visualization. Execution of the model depicted in Fig. 2 is presented in the video available at the address 6.

The development flow from the modeling tool to the target RTS is presented in Fig. 3. Starting from a model, IEC 61131-3/ IEC 61499 PLC code is generated, and then this specification is used by the parser and the native-code builder. The parser and builder are specialized in a way that they generate meta-logic beside logic expressed by the generated code. Meta-logic code is used to provide feedback information about the execution state of control logic, variables status, and unexpected states. Feedback from the RTS to a modeling tool is used for efficient refinement of a modeling language. RTS is a black-box accessible over various interfaces. It has different command groups for managing programs, debugger, scheduler, internal states, drivers, communication, tracing and watching variables. RTS can be used on local or remote machines and can execute specifications having million lines of code and a large number of tasks that depends on operation system (Win, WinCE and Linux). It can work as an interpreter or as a native-code executor. During execution, the status of each variable is propagated along with values to the process of calculations. Different strategies for recovery of missing values can be applied: propagate, reset on a new execution cycle, reset on assign, or ignore status. The use of the dynamic linker, along with other components, provides synchronization between a model and control logic code without stopping of RTS.

5. Experiences

Use of metamodels and related tools to specify and verify the created modeling languages early have been essential. A big development team was replaced by a team of three specialists: domain expert who can specify functional characteristics of medical devices, specialist for PLC code and target RTS, and language designer creating the metamodels. The actual language definition from a metamodel to a complete modeling tool took less than a week.

Implementing generators took more time as we also created different kind of generators. In our case, the generators produce and interpret functionalities of drivers, communication protocols, synchronization, and visualization rules. Since generators were freely open and modifiable, it would have been possible to connect with existing legacy applications and enable their use as a part of new devices. AR part of generators differ syntactically very little from the traditional code generators, so their implementation was not different.
For the RTS integration, engineers have preferred to use RTS as a separate executable unit (exe) and dynamic library (dll) or static library (lib) have been linked to top-level application for a medical device. From the point of integration with modeling tools and end-user application, it is possible to use RTS command language over TCP/IP channels of different priorities, or integrate tools and RTS using action reports that enable integration on a very high level of abstraction: directly from domain concepts used in design.

While domain-specific languages and code generators are applied in device development\(^1\),\(^8\), and we are aware of several cases in medical area (in the field of heart rate monitoring, hearing instruments, and pacemakers), we are not aware of other cases in which metamodel-based languages are been used with a tight connection to the run-time systems as in our case. The experiences on using the developed DSM languages, generators, model-debugging and run-time synchronization have been positive. No extra time is required to generate the application for validation of models. In most cases end-user application is only one of default views on the model.

For the developer the main task is to specify applications with the domain concepts and to describe hardware-specific signals and logic using some of more than two hundred already built-in functions and function blocks. The resulting models have then contained some thousand elements, like 3073 model elements in our case of blood separator. In practice, it takes less than two days for a new engineer to get running blood separator applications. Main reasons for this are the modeling languages providing high level of abstraction and target PLC based RTS enabling to build and debug applications without stopping their execution in the runtime-system. The actual synchronization between a running application and a model has been quick. Usually time spent for the synchronization between a model and code is 100-200 ms, which is not notable for a developer. Even for a very complex model, this synchronization time is less than 1 second. In this way, a device specification using DSM tools is executable and performed "on hot".

6. Conclusions

Use of models and code generators improve the development productivity and quality of the end product. We applied metamodels to specify DSM languages and related code generators for developing medical applications – blood separator devices in the reported case. We extended the traditional code generation approach with Action Reports providing model-debugging and synchronization between a state of a model and state of a program been executed in target runtime-system. These extensions provided several useful features: Developed control logic could be verified quickly and when needed corrected in a model at the same time. The created functionality for device development also enabled to monitor and optimize the use of hardware resources in parallel when modeling the applications. Our experiences indicate that the development of applications is significantly speeded yet the application quality is improved thanks to rules expressed in metamodel and linking models with the execution target via model debugging and synchronization. The approach itself can be generalized for developing other devices too as nothing in the approach is specific to blood separators as such. Naturally the metamodel and related generators along with ARs need to be defined for the selected device to obtain the benefits reported.

References