Run-time Code Generators for Model-level Debugging in Domain-specific Modeling

Verislav Djukić
Djukic Software GmbH
Nürnberg, Germany
info@djukic-soft.com

Aleksandar Popović
University of Montenegro
Faculty of Science
Podgorica, Montenegro
aleksandarp@rc.pmf.ac.me

Zhenli Lu
Changshu Institute of Technology
Changshu, China
zhenlilu@cslg.cn

Abstract

Code generators in the Domain-Specific Modeling (DSM) provide transformation from abstract models to specifications that can be executed, interpreted, or compiled using a target language compiler. Modeling tools may include parsers assisting software engineers to perform an inverse task; i.e., to create a set of language concepts based on existing program code. Practical benefits from such a reverse approach are limited, since source code cannot be used as an adequate base for creating language concepts on the high-abstraction level. Goal of application of the Model-Driven Development (MDD) approach, and DSM in particular, is to formally specify, using abstract models, a complete real system to the extent that models can be executed ([15]). In this paper we present an approach that completely achieve this goal. Model execution is implemented by the visual debugging of models, and submodels, which are dynamically created. In this approach, the most important role play performant code generators, so called run-time generators, and feedback that DSM tools get from run-time systems executing specifications.

Keywords: Run-time Code Generator; Metamodels; DSLs; Models; Visual Debugging

1. Introduction

Code generator languages, as well as their interpreters, may significantly simplify and improve the model-level debugging. These languages, especially navigational such as the MetaEdit+ Reporting Language (MERL) ([10], [14]), in their existing implementations completely solve following problems:

1. handling variations of a real system that software is developed for,
2. systematized refinement of a model, and program code generated from the model, as well as the refinement of modeling languages, and
3. managing an archive of models and code generators, instead of managing an achieve of program code.

The first part of our research is directed toward further evolution of the DSM tools for model execution. The second part is devoted to the model-level debugging in the field of robotics and automation. Some of our previous research efforts are presented in papers listed in the references section ([2], [5], and [16]). Those papers, along with associated appendices and video examples[1], refer to the different levels in the DSM architecture. We present an auto-adaptive run-time system (DVRTS) aimed at execution of control logic in automation and robotics in particular ([4]). DVRTS, with the associated compilers and linkers, is a system that implements control logic and meta-logic. We developed the level of meta-logic in order to maximally increase reliability of the program code execution, and to achieve run-time synchronization between elements of models and elements of implementation on the high abstraction level. The auto-adaptive run-time system, with various strategies for detection, documentation, and recovery from unexpected states, is a core element for model-level debugging. Action reports are extensions of the code generator level ([3]). On the syntax level, action reports are negligible extension of MERL. This extension includes the following:

1. feedback that DSM tools get from the system aimed at executing specifications,

[1] https://www.youtube.com/channel/UCqyYnYD6J5fEeb6Ni3YLuKg
http://www.djukic-soft.com/Framework/Robotics.html

2. dynamic creating and updating default visual representation of DSL concepts, and
3. end-user application generators that during run-time synchronize state of a target system, client applications, and modeling tool.

2. Run-time Code Generators

Run-time code generators (RTCGs) are Model-To-Text (M2T) and Text-To-Model (T2M) transformations that enable formal, precise, and efficient incremental transformation of models to executable program code for various target platforms. Beside, RTCGs are means by which DSM tools get feedback from a target system executing specifications. Fig. 1. outlines role of RTCGs in the architecture of the DSM solution.

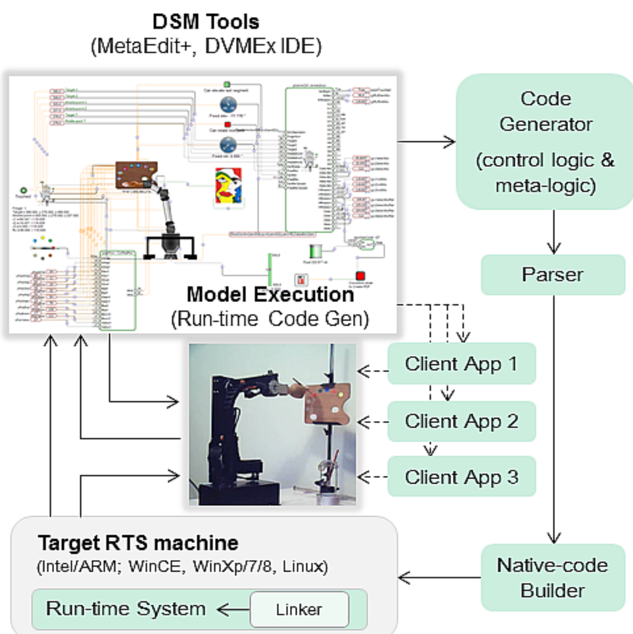

Figure 1. Role of RTCG-s in the DSM architecture.

RTCGs, using meta-models and model instances, generate source code in a target language. When modeling is performed by high-abstraction level DSLs, then those models are also used to generate meta-logic program code, as well as for various end-user applications. This one-way transformation from a model to program code is not suitable for the proper model-level debugging, so we paid special attention to the following extensions:

1. run-time construction of submodels and end-user views on a model,
2. integration of various command languages for communication between DSM tools and the target run-time system (RTS),
3. transactions, that are a mean for reliable transitions between RTS states and model states,
4. multi-client debugging, for simultaneous validation of model and generated end-user applications, and
5. generation of meta-logic code, and visual representation of control logic and meta-logic execution.

Run-time constructors of submodels are MERL-like programs for submodel construction in the time of control logic execution. The constructor parameters are types and instances of objects, roles and relations, values of properties, and statuses of operations and variables included in arithmetic and logic operations. On one hand, submodels, and code generated from them, significantly reduce the application state space, and on the other hand, enable variation of user-specific and domain-specific representations of control process.

Features of a target RTS executing specifications are crucial for model-level debugging. A DSM tool communicates with the RTS using a command language whose syntax is similar to the syntax of command languages used in operating systems. Real-time run-time systems, where it is expected to get prompt response when executing commands, use communication channels of different priorities, synchronous and asynchronous, and various communication protocols and techniques (TCP/IP, named pipes, etc.).

Analogously to the database systems, in DSM, transactions provide the reliable execution of the set of operations that shift the target RTS from one valid state to another. Transaction content largely depends on the model semantics, and end-user view on the model. The topological sort applied over graph representing a model also affects number, structure, and order of transactions. Topological sort is a procedure that rearrange model layout according to certain criteria, mostly according to the role of objects in relations. If a target RTS does not support the transaction concept, then code generators generate sets of elementary operations that initialize, execute, confirm, and cancel commands or restore previous values. Otherwise, it is enough to envelope part of program code for communication with action report commands *begin_trans* and *end_trans*.

Multi-client debugging is a procedure that dynamically generates and executes submodels. In a typical use-case scenario, part of a model is transformed to control logic program code, while another part is transformed to end-user applications. Several end-user applications, generated in the run-time, are simultaneously directed towards single target RTS and instance of a program being debugged. This approach, where end-user applications for debugging are dynamically created, is applicable due to following reasons: 1) simplicity of mapping of DSL to implementation concepts; and 2) speed of code generation, which in most cases takes 200 milliseconds per a model of medium complexity.

DSM significantly increases the execution reliability of code because code automatically generated from models is generally less error-prone than manually written code. We notably improved model-level debugging concerning operation statuses by introducing the meta-logic and meta-arithmetic; i.e., by updating statuses of each arithmetic and logic operation. In this way, we provide direct mapping statuses of the DSL concepts to variables and status registers of a processor executing logic. General purpose language compilers, such as Java and C++, are not flexible enough to support visual debugging using high level language concepts, therefore we prefer specially tailored compilers, such as our extended compiler for programmable logic controllers (PLC).

3. Model-level debugging and DSLs

In this section we describe our experiences in the construction of DSLs in the automation and robotics, where model-level debugging is raised to such level that DSM models, along with various views, are end-user and debugging application at the same time. In our DVMEx approach for automation and robotics, systems are modeled using three DSL types. The first DSL contains concepts for specification of control logic, and it is similar to the function block language. The second one is aimed at specification of topological and mechanical features of a machine executing actions; e.g., properties of a robot arm. The third language is aimed at describing an environment where control logic is executed. DSLs are integrated on the meta-level. Three types of roles and relation are used in integrated models since a system is modeled using three types of languages. The first type is for objects used to specify control logic. The second type of roles and relations is used to specify relations between elements of control logic and elements of the environment where actions and activities are executed (input and output signals). The third type is used to specify relations between concepts for defining topology of machines (robots) and elements of control logic and environment. Model-level debugging is a procedure for incremental generation of applications by means of RTCGs and their execution on the target RTS. This iterative procedure includes dynamic construction and updating of submodels, roles and relations, and dynamic construction of different visual syntaxes; i.e., visual representations of language concepts. The "dynamic" term denotes update of a (sub)model, generated code and synchronization with an running program instance. In the rest of this section we describe how to generate program code for control logic, how to communicate with the RTS, and how to exchange messages and values of properties. Also, we describe how to invoke methods over client applications and the target RTS in the modeling and run-time.

3.1 Submodel construction

Navigational languages for code generation are suitable for the submodel construction because they support graph traversal by types and instances of roles, relations, and objects. In the model-level debugging, purpose of the submodel construction is to: 1) reduce the set of application states to the level that it can be visually traced; 2) vary values of object included in a model in an efficient and simple way; and 3) generate representative documentation.

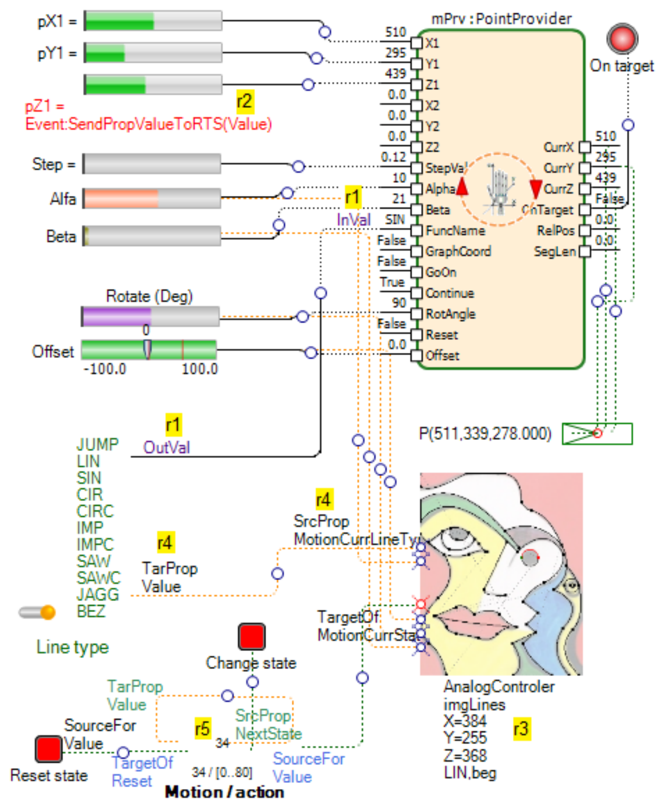

Figure 2. A submodel example including different DSLs.

3.2 Exchanging values of properties, invoking functions and events

Submodel in Fig. 2. contains elements providing enough information for generating applications, and control logic code for robot controllers. Roles, whose types are *OutVal* and *InVal* (r1), are translated into control logic program code, consisting of variables and expressions such as *pPriv.X1 := pX1;* according to the IEC61131-3 language syntax ([8]). On pZ1 slider (r2), it is defined that each change of the slider value on a model, or within a client application, causes setting the value of an appropriate variable in control logic. This value may be also obtained from the imgLines analogue controller instance (r3), that contains set of all possible motions of the robot arm. The *SrcProp* and *TarProp* roles (r4) between the imgLines and the *LineType* object determines a way for exchanging properties values of objects independently from control logic, but only within an debugging application or a modeling tool. The *Reset* switch is a source for resetting the *Motion/action* state machine (5). A source object has the *Value* property, while a target object has *Reset* property. In the transformation process, these roles and relations are transformed to action reports code that takes following form: *.swAction.Reset=stReset.Value;*. These expressions are interpreted locally, using the MERL interpreter, but variables are on the RTS side. Their values are obtained from the RTS, before calculations that are executed locally. In Fig 2. the *Motion/action* state machine includes 81 possible state, which corresponds to the set of actions that the robot arm performs while drawing a portrait sketch. The *ChangeState* switch controls transition from one state to another. The state machine includes the *Value* and *NextState* properties. The *SourceFor*, by the *Value* property on the right-hand side of the machine, provides value for selecting the current line that will be drawn on a portrait sketch.

Properties defined over roles, as well as their values and domains, may be functions, expressions, events or reports that are in the run-time evaluated locally or on the RTS side. This approach, combined with incremental code generation, enables ad-hoc construction of test scenarios for testing models and generated program code. MERL syntax for referencing property values is slightly extended, as shown in the following example:

```
:.mPrav#FuncName.Value=$mList[$cnt];
```

The *Value* property of the *FuncName* port of *mPrav* object are assigned with the value of the variable that belong to the *mList* list, on the *$cnt*-th position.

```
:ConnPointAbsFor(,x);
```

Returning the X-position of the current connection point belonging to the object on top of the stack.

```
:Left=ConnPointAbsFor(,x);
```

Setting the left-position of the current object, to be the position of the current connection point for linking objects.

```
:GetPropValueAsString(Top);
```

Invocation of a function whose input parameter is a name of the property whose value is required as a string.

```
:SendPropValueToRTS(prName),1;
```

Sending to the RTS value of the *propName* property belonging to the object being on top-1 of the stack.

```
:.TargetObjectID.SendPropValueToRTS(prName);
```

Sending to the RTS value of the *propName* property belonging to the object identified by TargetObjectID.

3.3 Arbitrary user components as a part of a modeling tool

For visual debugging of domain-specific models, each language concept require at least one visual representation. For the modeling purpose those representations (visual syntax) may be rather simple, but for the model-level debugging it is important to have proper and functional representation as possible. Therefore, the requirement naturally arises that an arbitrary user component; e.g., user control, may be easily integrated within a modeling tools. On the other hand, including arbitrary and platform-dependent components requires a meta-model extension, and writing platform-specific parts of a generator. The solution we have applied in the robotics and automation field is based on writing generators for creating code generators for platform-specific components.

Properties included in the model execution may belong to the one of three groups. The first group includes properties that have default representation in a modeling tool (mostly textual), they are not, or cannot be, mapped to a property of some user component. The second group are properties directly mapped to one or more properties of one or more user components. The third group of properties are those belonging to user components, but they are not a part of the language definition. In order to reference these properties from MERL, no matter if a property value is set or only get, expressions in following forms are used: *:propName;* and *:propName=newValue;*. First, the interpreter tries to find property that is part of DSL definition, and then checks if it is mapped to a property of a user control. If such a property is not found, then it tries to find a property of a user control. In this way, using meta-model, equality of syntaxes is achieved, and platform-specificities are abstracted.

```
<Type name ="TwoStateControler"
id="DVLangObject">
<ctrlList>
       <ctrl type="DVMExTwoStateSwitch" id
       ="ID" connProp ="Connections"
       dll="DVControl.dll" ns="DVMExControls">
<pList>
  <prop name="ID" propType="Text" impName
  ="Name"/>
  <prop name="PortAddress" propType="String"
  impName ="" label="HwPort" domain="String"
  defaultValue=""/>
</pList> </ctrlList>
```

Listing 1. Platform-specific components in metamodel

In the previous XML listing, which is a part of the metamodel for DSL depicted in Fig. 2, the following is defined: A *TwoStateControler* object in some implementation (modeling tool, debugging application, or client application) uses a set of controls named *ctrlList*. One of these controls is a .Net component named *DVMExTwoState Switch*, whose implementation is in *DVControl.dll*, within the namespace *DVMExControls*. Object instances are identified by the *ID* property, which is mapped to property *Name*. Apart from identifiers, the control includes the *PortAddress* property used for linking hardware signals, and this property is not mapped in any other control property.

3.4 Synchronization of a model and the RTS

We extended the MERL language with several commands in order to synchronize a model and program code executed on the target system. The DVRTS target system for robotics and automation have several communication channels aimed at receiving commands and sending responses. The DVRTSComLang command language is a main interface. Besides, there is a direct interface, where commands are called by invoking appropriate functions, and there is an interface where command are sent in the XML form. Communication channels may be both synchronous and asynchronous, and the priority of command execution may be changed. Interfaces using the named pipes and message queues techniques are suitable for standalone solutions, when the RTS and a client-application, or a modeling tool, are running on the same machine. The TCP/IP channels, or remote consoles are used to access remote RTSs.

Command used for synchronization of models and applications are following:

1. `begin_trans` and `end_trans`, enveloping set of commands that should be executed as one transaction by the RTS,
2. `webservice`, calling a web service on the modeling tool or RTS side,
3. `f:external`, executing DVRTS ComLang or any external command over the target RTS,
4. `foreach {.} where ROOT RoleType`, generating code or set of commands whose execution order depends on the topological sort of a model by roles of objects in relations,
5. `function`, calling built-in functions, and
6. `toset` and `tosetunique`, transforming results of the command execution to MERL collections.

4. Multi-client model-level debugging

In our approach debugging is a user-driven activity for executing different dynamically created submodels and variations of their visual representations. DSM tools and applications generated from models use metamodels and mapping of object and properties to platform-specific components. This approach, owing to use of metamodels, creates an opportunity to improve existing client components and applications so they can be used as modeling tools. Based on our initial experience with DSM, usage of metamodels gains greater practical benefits than reverse engineering of existing source code.

One typical use-case scenario of multi-client debugging is presented in Fig 3. Models are extended with relations and roles for exchanging values of properties, and the synchronization with the RTS, as it is described in section 3.1. Using the MERL language and interpreter, set of submodels, or model views, is created. Each submodel becomes separate application that is executed over different platforms. Each target platform includes one instance of the MERL interpreter. This interpreter uses metamodel and submodel definition. Client application receives commands using user components that the DSL concepts are mapped to or directly from hardware signals. Such a distributed debugger solves problems related to the graphics on embedded devices, as well as problems related to the limited hardware interfaces of desktop computers.

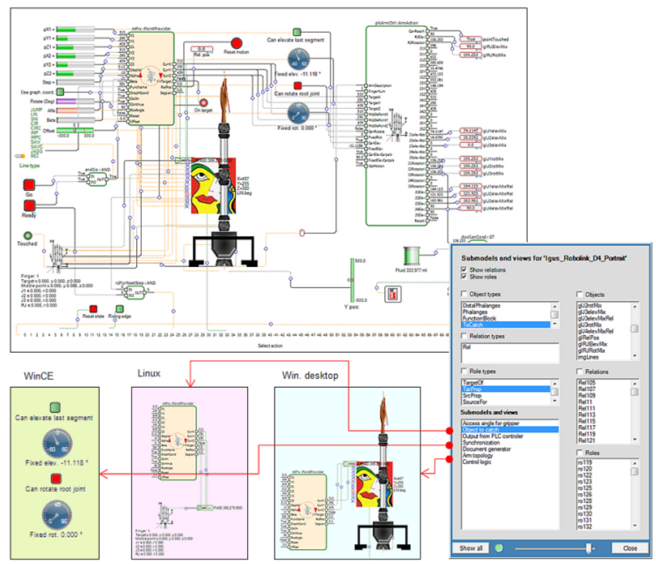

Figure 3. Multi-client debugging.

5. Related work

Nowadays, significant research efforts have been invested in model level debugging, especially for embedded systems ([9], and [17]). Updating properties of objects, relations and roles, as well as inserting and deleting connections, turned out to be an advanced feature for model-level debugging, and automated refinement of models and modeling languages. Debugging software for embedded systems on the platform-level is tedious and error-prone, since obsolete techniques are used such as "printf" statements, data monitors, etc. In most of these approaches, middleware captures actual run-time data for inputs and outputs of real-time tasks executed on the target platform, and then this information is mapped back to the corresponding model ([9]). UML concepts and diagrams are frequently used for modeling and data visualization. We additionally apply the DSM principles enabling designer to use an arbitrary user control for language concepts. Also, by means of action reports, information obtained by run-time system is dynamically mapped into user controls properties. In this way, a user can define own DSL by means of existing user controls for concrete domain. There are many successful applications of the DSM approach, and DSLs in general, in these fields ([1], and [11]). A domain-specific language (DSL) is a language tailored to a specific application domain. DSLs offer abstraction and notation close to the application domain and therefore provide better expressiveness ([9]). Numerous successful applications of DSLs are reported in various domains ([5], [7], and [13]), including also robotics domain ([12]).

6. Conclusion

Model-level debugging, as a part of the DSM application development process, is the most productive way for the software verification. This approach, and the debugging scenario, easily includes user components (controls) that in most cases already exist for various application domains. Those components, which are primarily developed as a part of applications, are integrated within the modeling tool. Therefore, DSM models and submodels became a valid default end-user applications constructed by well-known navigational languages for code generation. Using simple mapping of abstract and domain-specific model elements to platform-specific controls and properties, the code generators are used for model execution on various target platforms. The RTCGs expand list of existing advantages of MDD and DSM application in software engineering, with the model execution. The model execution is specified on the code generator level. Owing to direct connection between modeling tool and a target system executing models, there is no need to write separate program code for debugging and simulations, but production code is used for this purpose. Also, there is no need for separate specification of client applications and debugging applications, since applications are only submodels and user specific views on a model where the model elements are associated with specific visual representation.

Our present experiences with the software development where RTCGs are used indicate tenfold increase of productivity, and high reliability of program code. These experiences are related to the document engineering, automation, and robotics. Our current research efforts are directed towards further development of concepts aimed to support multi-client debugging in DSM tools, and debugging models containing elements from different abstraction levels.

References

[1] Berger C., S*enseDSL: Automating the Integration of Sensors for MCU-based Robots and Cyber-Physical Systems*, SPLASH '14, Workshop on DSM'14, 2014, Proceedings.

[2] Djukic V., Lukovic I., and Popović A., *Domain-Specific Modeling in Document Engineering*, Proceedings of the Federated Conference on Computer Science and Information Systems, Poland, 2011

[3] Djukic V., Lukovic I., Popović A., and Ivančević V., *Model Execution: An Approach based on extending Domain-Specific Modeling with Action Reports*, Computer Science and Information Systems (ComSIS), DOI: 10.2298/ CSIS121228059D, ISSN: 1820-0214, Vol. 10, No. 4, 2013.

[4] Djukić V., Popović A., Tolvanen J.P., *Domain-Specific Modeling for Robotics – from language construction to ready-made controllers and end-user applications*, Model-Driven Robot Software Engineering (MORSE), July 1, 2016, Leipzig, Germany

[5] Djukić V., Popović A., Tolvanen J.P., *Using domain-specific modeling languages for medical device development*, Web journal for Embedded Systems, 2014, http://www.embedded.com/design/ programming-languages-and-tools/4429401/Using-domain-specific-modeling-languages-for-medical-device-development

[6] Fister I. J., Fister I., Mernik M., and Brest J., *Design and implementation of domain-specic language easytime*, Computer Languages, Systems & Structures 37 (4) (2011) 151{167. doi:10.1016/j.cl.2011.04.001.

[7] Haberl W., Herrmannsdoerfer M., Birke J., Baumgarten U., Model-Level Debugging of Embedded Real-Time Systems, CIT 2010, pp. 887-1894

[8] International Electrotechnical Commission, *International Standard IEC61499, Function Blocks, Part 1-Part 4*, IEC, 2005.

[9] Iyenghar P., Pulvermüller E., Westerkamp C., Uelschen M, Wuebbelmann J., *Model-Based Debugging of Embedded Software Systems*, Softwaretechnik-Trends 31(3), 2011.

[10] Mernik M., Heering J., and Sloane A.M., *When and how to Develop Domain-Specific Languages*, ACM Computing Surveys, 37(4):316–344, 2005.

[11] MetaEdit+ Workbench, *Workbench User's Guide*, http://www.metacase.com/support/45/manuals/mwb/Mw.htm l

[12] Nordmann A.;Hochgeschwender N., and Wrede S., *A Survey on Domain-Specific Languages in Robotics, Simulation, Modeling, and Programming for Autonomous Robots*, 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings

[13] Pradhan S. M., Dubey A., Gokhale A. S., Lehofer M., *CHARIOT: a domain specific language for extensible cyber-physical systems*, SPLASH '15, Workshop on DSM'15, 2015, Proceedings

[14] Schaefer C., Kuhn T., and Trapp M, *A Pattern-based Approach to DSL Development*, SPLASH '11, Workshop on DSM'11, Proceeding.

[15] Steven Kelly, and Juha-Pekka Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, ISBN: 978-0-470-03666-2, March 2008, Wiley-IEEE Computer Society Press.

[16] Tolvanen J.P., Djukic V., Popović A., *Metamodeling for Medical Devices: Code Generation, Model-debugging and Run-time Synchronization*, Procedia Computer Science, Elsevier, DOI: 10.1016/j.procs.2015.08.382, ISSN: 1877-0509, Vol. 63, 2015, pp. 539-544.

[17] Zeng K., Guo Y., Angelov C, *Graphical Model Debugger Framework for Embedded Systems*, DATE, 2010, pp. 87-9