Computing and Informatics, Vol. 38, 2019, 497–524, doi: 10.31577/cai_2019_2_497

MODEL VARIATIONS AND AUTOMATED REFINEMENT OF DOMAIN-SPECIFIC MODELING LANGUAGES FOR ROBOT-MOTION CONTROL

Verislav Djukić

Djukic Software GmbH
Nürnberg, Germany
e-mail: `info@djukic-soft.com`

Aleksandar Popović

Faculty of Science, University of Montenegro
Podgorica, Montenegro
e-mail: `aleksandarp@rc.pmf.ac.me`

Ivan Luković, Vladimir Ivančević

Faculty of Technical Sciences, University of Novi Sad
Novi Sad, Serbia
e-mail: {`ivan, dragoman`}`@uns.ac.rs`

Abstract. This paper presents an approach to handling frequent variations of modeling languages and models. The approach is based on Domain-Specific Modeling and linking of modeling tools with adaptive Run-Time Systems. The applicability of our solution is illustrated on an example of domain-specific languages for robot control. Special attention was given to the following problems: 1) model-level debugging; 2) performing fast transformation of models to native code for various hardware platforms and operating systems; and 3) specification of views and view-based generation of applications for validation of meta-models, models, and generated code. The feedback for automated refinement of models and meta-models is provided by a custom adaptive Run-Time System. For the purpose of synchronizing models, meta-models, and the target Run-Time System, we introduce action reports, which allow model-level debugging. In order to simplify handling of frequent model variations, we have introduced the linguistic concept of a modifier.

Keywords: Domain-specific modeling, run-time system, model variations, model execution, model-level debugging

Mathematics Subject Classification 2010: 68Q60, 68T40

1 INTRODUCTION

In this paper, we communicate our experience concerning the development and application of Domain-Specific Modeling (DSM) and adaptive Run-Time Systems (RTS) for robot control. We present typical problems and our solutions related to the practice of:

1. constructing and applying robot-motion control languages (RMCL);
2. frequent variations of robot control models;
3. model execution and model-level debugging, i.e., incremental updating of control logic; and
4. parallel refinement of robot control models and modeling languages.

Our approach, which is based on the DSM architecture [1], is named the DVMEx Approach. In order to verify the approach in practice, we developed run-time systems, compilers, interpreters, and modeling tools, all of which comprise the DVMEx IDE. Besides using our components to verify the approach, we also employed MetaEdit+ WB and MetaEdit Modeler [2]. These tools have been successfully used in various domains, e.g., automation, control and embedded systems.

There is a need for significant improvements of software development in automation and robot control, especially in the development of tools for:

1. formal specification and execution of control processes, and
2. construction and application of RMCLs.

For each level in the architecture of DSM solution, we specify one of the most important problems.

The level of the modeling languages. General purpose graphical languages (e.g., UML) are often used for modeling of specific processes, but they are not sufficiently understandable to users in a particular application domain.

The level of the model transformations. Transformations are complicated for an average programmer. Moreover, they are focused, or even limited, to a single target general purpose programming language (GPL).

The level of the target interpreter or run-time system. The target interpreter in most cases does not contain meta-data describing the semantics of a control process, but only commands concerning the semantics of basic logical and arithmetic operations. A single invalid basic operation may lead to an unexpected or unresolvable state. Parsers used for reverse construction of class diagrams or state diagrams are part of most UML tools, but they do not solve the problem of losing the relationships between the modeling tools and the target interpreter of specification.

The DVMEx Approach solves these problems in the following manner:

The level of the modeling languages. Instead of GPLs and existing robot-motion control languages or operating systems, such as Robot Operating System (ROS [3]), more DSLs are constructed and used. Over the DSL models, three views are initially defined, one of which is focused on the topological properties of a robot arm, the other on motion and control logic, and the third on the real-world environment where the robot performs actions. Due to the usage of DSLs, all these views are close and comprehensible to end-users, domain specialists, and software architects. Figure 1 shows the three views on the robot control model. Three different DSLs, which are integrated at the meta-model level, are used simultaneously for the modeling robot task. The first language, which is presented in the left-hand side in Figure 1, is aimed at specification of topological properties of a robot arm, such as number of joints and fingers, length of each segment, constraints for rotation and elevation, etc. The second language, which is presented in the middle of Figure 1, is used for description of a state machine, i.e., an initial set of actions and states, as well as their relations with signals coming through various sensors, and commands explicitly issued by end-users. Since the function-block language (IEC 61131-3) is used in the automation and industry, it is convenient that this DSL uses function-block diagrams with graphical syntax that is close to users from the concrete application domain. The third language and submodel (right-hand side in Figure 1) are used for description of environment in which a robot performs actions. It is to the greatest extent specific for the application domain. When it comes to the drawing of portraits or sketches, this language is used to specify motions, canvas dimensions, canvas distance, rotation and elevation in 3D space, relative to the reference point of the robot arm. When there are obstacles between the canvas and robot arm, then the DSL for motion specification should include concepts for expressing the effects of obstacles on motion. Figure 1 presents an example of a motion variation, where curve is shifted upward, and then rotated.

The first and second DSLs are being constructed quickly in practice, and the validity of the specification can be more easily verified than in the third language. A simple construction of language concepts for 3D representation of motion is not expected from a general purpose DSM tool. Therefore, this specific problem is solved by using action reports [4] and more advanced libraries or 3D visualization components, which are not part of DSM tools. Additional descriptions may be found in the section devoted to the model-level debugging.

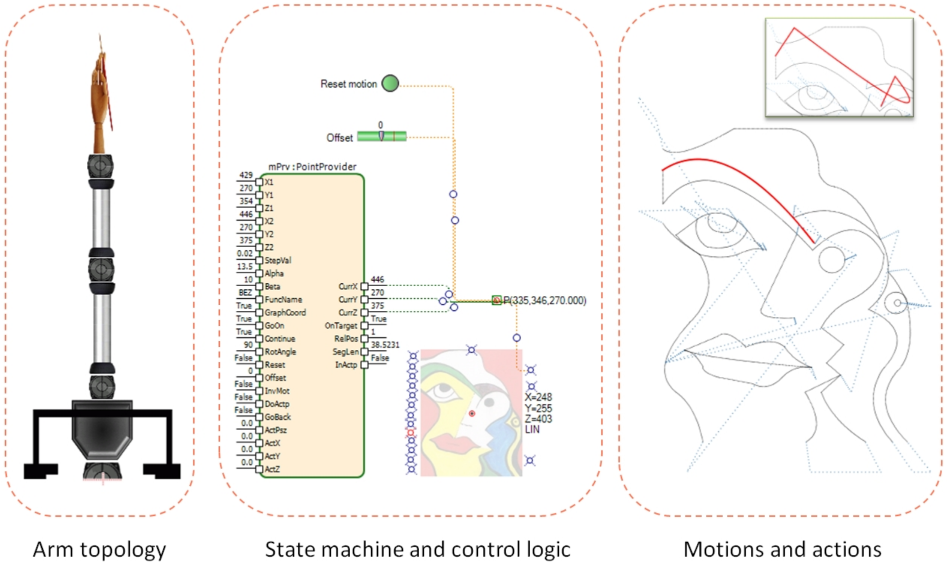

Figure 1. Views on the DSL for robot control

The level of the code generators. An interpreter of MERL-like specifications is implemented [2] for M2T (model to text) transformations, to create code and code generators (generators of generators). Starting from an instance of a model of certain type, the interpreter of MERL-like specifications generates the code concerning the control logic, "meta-logic", and "meta-arithmetic", as well as action reports [4]. Action reports synchronize the state between the model, RTS, and monitoring application, unless the modeling tool is used for monitoring. They are the means for model-level debugging and model execution. The code concerning the "meta-logic" and "meta-arithmetic" is described in Section 5. In brief, this code serves two basic purposes:

1. it increases reliability of control logic by implementing different strategies for recovering the system from an invalid state; and
2. it detects atypical states of the control logic and provides feedback to the modeling tool, which is used to refine the DSL.

The level of the target interpreter or run-time system. The Run-Time System (RTS) is conceived and implemented as an adaptive component that

1. executes both the instructions belonging to a high level of abstraction and binary (native) code;
2. receives and links specification increments without interruption; and
3. simultaneously executes the basic control logic and "meta-logic".

When compared to the similar Programmable Logic Controllers (PLC) used to execute IEC 61131-3 programs [5], it is extended with the concepts from IEC 61499 [6] and a set of libraries for various application domains. In this manner, the modeling of distributed controllers, which is based on finite state automata, is simplified.

Besides Introduction and Conclusion, this paper has six sections. We first give a short description of the architecture of our DVMEx solution for robot control (Section 2). Different approaches to solving the problem of unexpected system states, frequent model variations and their implications regarding the construction of modeling languages are described in Section 3. This section also features a short review of papers related to the RTS-driven approach to the application refinement and the construction of UML profiles for the purpose of model-driven development of industrial process control applications. In Section 4, there is an overview on the evolution of modeling languages, from more general to more domain-specific. We outline different approaches to language construction, from the classification using subtypes to modifications, which are a linguistic concept allowing specification of robot actions at different abstraction levels. In Section 5, we elaborate on ways to specify the "meta-logic" and "meta-arithmetic" at the level of the model, code generator, compiler and run-time system. In Section 6, we describe a platform for model-level debugging, which includes visual tracing. Section 7 contains related work. In Section 8, we conclude this paper by outlining our generic model of DSL refinement and listing our theoretical and practical contributions.

2 THE ARCHITECTURE OF A DVMEX SOLUTION FOR THE ROBOT CONTROL

The architecture of DVMEx Solution for the robot-motion control (Figure 2) represents an extension and a concretization of the basic architecture of DSM solutions [1], which encompasses a DSL, code generator, domain framework, and interpreter or target system. Special attention is devoted to:

1. extending a target interpreter (run-time system), which, in addition to executing complex control operations, executes also **meta-logic** operations and provides feedback to modeling tools;
2. extending code generators, to which we added **action reports**, which synchronize the state of the run-time system with tools for modeling and meta-modeling, and various applications; and
3. using **modifiers** to construct, slice and merge modeling languages.

The meta-modeler creates DSLs using tools for meta-modeling. These languages may be created separately for each domain, i.e., each type of the task that the robot is supposed to execute.

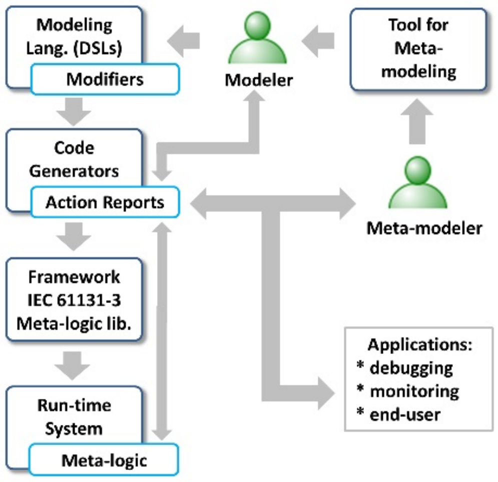

Figure 2. The architecture of the DVMEx solution

There are several groups of code generators. The first one includes generators that generate source code in IEC 61131-3, some GPL, quasi assembly instructions of higher abstraction level or, directly, native code for a particular processor type. The second group includes "generators generating generators", i.e., generators of action reports. They provide the high-level synchronization between the modeling tools, target RTS, and monitoring applications. The third group of generators is used to specify the semantic actions executed during model changes. These generators define rules concerning the integrity of models and transactions, as well as states in which there is synchronization between abstract models and code executed in the RTS. The fourth group includes generators that generate documentation in the PDF or HTML format, either directly or by using document description DSLs as intermediaries.

The framework of the DVMEx solution for robot control includes

1. compilers created to support more thorough specifications of control processes;
2. libraries; and
3. web services.

At the framework level, there are also libraries used to implement complex 3D motions and actions, rules of "meta-logic" and "meta-arithmetic", which is further elaborated on in Section 5.

The run-time system interprets or executes specifications created from abstract models. The available versions support Linux, WinCE and Win XP/7/8/10, from which they utilize memory and file managers, as well as the TCP/IP protocols. The RTS features a preemptive adaptive scheduler managing tasks of different types and priorities, synchronized with the drivers and monitoring applications. Unless the modeling tool is used for monitoring, client applications are generated using a code generator. In this manner, the action report interpreters, which also support Linux, WinCE and Windows platforms, are used as default applications for the model-level debugging. The visual debugging also uses services of DVDocGen Framework ([7]) to generate PDF or HTML documents. Document scripts, which are specified using a textual DSL for documents, are dynamically generated during the model debugging, providing PDF and HTML documentation for the verification of test cases.

The mechanism concerning the feedback from the RTS to the other levels that are part of the DVMEx architecture is implemented by monitoring the model execution in the RTS and by event triggering. The RTS recognizes predefined (or built-in) states of the interpretation or execution of control logic: Before variable initialization; After variable initialization; Before state changed; After state changed, etc. The monitoring application or modeling tool sets a filter that determines the particular state changes together with the parameter they produce, which are relevant and should be considered. Besides the default states of the RTS, the IEC 61131-3 language and compiler are extended with events, similarly to GPLs. The request for state change is defined as a 4-tuple (Condition, Event_ID, Parameters, RTS_State). These requests may originate from the hardware level, drivers, control logic and meta-logic code.

In addition to these states, which are considered valid, the RTS detects the invalid variable values or wrongly executed arithmetic and logical operations. The strategy for resolving these situations is presented in detail in Section 5. Each of the feedback connections from the lower to the higher abstraction levels provided metadata that are sufficient to make a reference to an instance of the object, relationship, role, model, and code generators that generated certain code portion.

We conclude the overview of the architecture of the DVMEx solution with a remark that it is an extension of DSM architecture. The level of the code generator is extended with action reports, enabling the visual debugging and model-execution without additional programming. The RTS is adaptive and supports updating of the code that is responsible for control logic and meta-logic during run-time.

3 UNEXPECTED STATE OF THE MODELS AND CODE AND MODEL VARIATIONS

An unexpected state represents a paradigm uncovering problems that are related to errors in RTS or in models, model variations, the incompleteness of a modeling language, or the non-adaptivity of the run-time system. There are at least three environmental causes of unexpected states of a system:

1. the lack of appropriate modeling concepts (an incomplete DSL);
2. the lack of code generators; and
3. insufficient power or flexibility of the target RTS that interprets or executes the generated code.

The software for automation and robot control that handles a large number of unexpected states is inefficient and expensive, particularly if general purpose languages or tools are used to create and maintain the software. The appearance of an unexpected state diverts production activities from the expected workflow and decreases the level of their automation. The main characteristics of the unexpected state with respect to the impact on the automation and robot control process are:

- during the execution of a task, an unexpected state cannot be abstracted as any existing model or pattern, described using the existing DSL concepts;
- the actors of a control process are capable of perceiving unexpected states by using their own experience, personal creativity, and the level of knowledge of the modeling framework and language;
- unexpected states are also the ones for which code generators cannot produce the expected code; and
- the control logic of systems in which unexpected states are frequent is most often generated or programmed again, and, during the switch to the new code, the whole control process is temporarily stopped.

A significant amount of research aims at providing support for modeling variant-rich software systems (Software Product Lines) in general. Patterns play an important role in solving the problem of specifying variations. There are a lot of references covering application of patterns in DSM. In [8], the process of creating UML profiles for particular domains is presented. In [9], the authors discussed the role of patterns in the construction of valid DSLs. One of the languages for this purpose is the Common Variability Language (CVL) [10]. CVL is a generic language for modeling variability in models in any DSL based on the Meta Object Facility (MOF). Although conceptually quite comprehensive, CVL still does not offer an adequate support for systematic refinement of models and modeling languages. The following practical constraints limit the use of CVL for this purpose:

- specifying CVL fragments and their referencing need to be more intuitive and thus easier for average users;
- specification of a large number of variations at the level of a model significantly diminishes model understanding and usability;
- specification of variations needs to be provided as a User-Driven Modeling (UDM) activity or influenced by the run-time system, due to the requirement of systematic gathering and classification of variations; and
- a specification of variations needs to be provided not only at the level of models, but also at the level of a target language to which the models are transformed. In practice, this is motivated by the requirement that the effects of a variation may be scoped to different abstraction levels. For example, variation effects may be scoped both to main control logic and monitoring applications.

For the purpose of synchronization we worked out incremental modeling, which includes reliable "on the fly" validation of as many states as possible at the side of the running target system, instead of emulators. We propose improving synchronization that provides:

1. specifications of model variations and synchronization of abstract models, and executable code at the level of meta-model and code generators;
2. systematic refinement of the DSL by means of an analysis system states; and
3. the introduction of the concept of a *modifier*, as means to flexible systematic classification of system states and model variations.

In Figure 3, we present a DSL for modeling of a robot arm. This language, with minor additions, may be used in practice for modeling different types of grippers and industrial robots. This model serves as a running example that illustrates how our approach may alleviate some of the important problems in construction and refinement of DSLs and models, multilevel modeling, model variations and model execution.

The robot arm consists of one or more fingers, each further composed of one or more segments. *FingerSegment* represents the base object type in the DSL for arm. By modifying this type, we created additional types: *Joint*, *Phalangs* and *DistalPhalangs* (see the upper section of Figure 3). Each modification is represented by modifier object, a modification relationship and the roles of the base object and the modified object. Furthermore, the arm also includes *Carpals*, *RootJoint* and *Underarm*. The *ArmAction* object acts as a provider of position, elevation and rotation of finger segments. During language testing, it may be convenient to test only some of its elements or submodels, e.g., only two fingers. The formed relationships with the ArmAction object determine which of the fingers are included in model interpretation. The purpose of such an approach to DSL construction is to modify the metamodel through a model instance. This is achieved by modifying default values, and introducing (or removing) attributes and relationships in order to create new types, supertypes, subtypes and object instances. On each model change, during visual debugging, the following activities are performed:

1. control logic code is generated;
2. a description of the arm's topology is generated; and
3. the model is executed to serve as a visual debugger [11].

Modifiers may be used to express significantly more complex relationships that are used for:

1. multilevel modeling, which combines inheritance and instantiation;
2. inheritance of a type from an instance, a type from a type, an instance from an instance, a type and an instance from the generated code; and
3. specification of a set of allowed model variations.

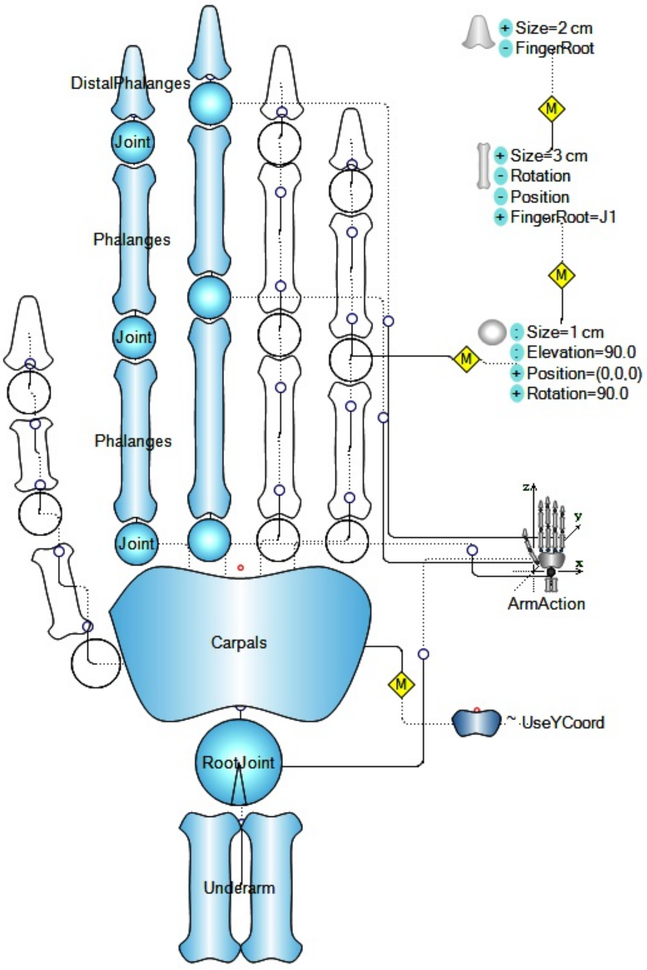

Figure 3. Modifiers as means to DSL refinement

In Figure 3, the *modification* relationship is represented by a diamond shape with the **M** symbol inside. Modified objects are denoted by ⟨*Mod:modName*⟩. The semantic of property modification may be: *instance*, denoted by the (:); *new property*, denoted by the (+); or *inapplicable property*, denoted by the (-) symbol. Within relationships, modifications can change and remove relation and role types. The first two property roles are intuitively clear as they are available with the same meaning in the majority of contemporary modeling languages. The inapplicable property is introduced as a counterpart of the potency concept from the UML extension [12]. However, in our solution it does not require predefining the allowed depth of instantiation and does not restrict any attribute to be applicable again to some subtype or instance.

The example from Figure 3 shows a portion of the DSL, in which the types of robot arms are described. At the most general level, the *FingerSegment* is defined as a segment for which the *elevation* and the *size* are known. For the sake of the completeness of the RMCL specification, a constraint must be used to explicitly express the rule that the *Elevation* depends on the elevation of the previous sequential segments. The *Joint* object is a modification of a segment with the default size of 1 cm and the default elevation of 90 degrees that is also extended with the values for the position of the joint center and the rotation angle (with respect to the X axis in an XY coordinate plane). *Joint* is a subtype of *FingerSegment.* The *Phalanges* object has the default size of 3 cm. However, the rotation angle is not applicable to this object because the corresponding segment may move only along a single axis. The same is true for the position because *Phalanges* is linked to a joint. As it is necessary to know the base joint for each *Phalanges*, we introduced the *FingerRoot* property. Although it may appear that *DistalPhalanges* is a subtype of the *Phalanges* object, that is not the case here. It has the default size of 2 cm and the *FingerRoot* property is redundant because there is at least one joint between *Phalanges* and *DistalPhalanges* that is not the root of the finger.

The presented sequence or hierarchy of object modifications is only one possible case, as variations and their semantics depend on mechanical and topological properties of the robot arm for which various types of programs have to be generated. The given example is an illustration of the problem of frequent variations of models and modeling languages, which may be solved by multilevel modeling, i.e., by integrating modeling and metamodeling.

In order to illustrate such an approach to language construction, we provide an examples of textual representation (the so-called DSL script) of robot arm objects. Each arm specification is a sequence of the DSL scripts given below.

⟨`Joint.Size:1.5 cm,Position(10,20,10)`⟩

The joint of size 1.5 cm at the initial position $(10, 20, 10)$

⟨`Phalanges`⟩

The default Phalanges of size 3 cm

⟨`DistalPhalanges.Size:2.2 cm`⟩

DistalPhalanges of size 2.2 cm

One expected purpose of the RMCL is to simultaneously express different aspects of robot control which may depend on mechanical, thermal, electrical or visual properties. For the purpose of integrating the languages used to model individual aspects, it is necessary to provide a sufficiently flexible mechanism for establishing semantic relationships between language concepts that are related to different aspects. The herein introduced modifiers could be the mechanism for this kind of meta-modeling. The code generators, together with action reports, which act as an interface between the model and the DSL scripts, allow for different interpretations of the examples from Figures 3 and 4.

4 DSL REFINEMENT ON EXAMPLES

The practical benefits of using DSM and modifiers are illustrated in the development of software for the robot that paints. We set up a small DSM team consisting of a software engineer (a DSM specialist), domain expert (a mechanics constructor) and end-user (a painter). The software engineer constructs the language, trying to identify domain-specific concepts from the lower to higher level of abstraction. In this process, the greatest assistance is rapid DSL verification using a set of initial models and a target interpreter, or run-time system supporting an incremental update. The domain expert knows the problems that his or her robot is capable of solving, current and potential user requests, and what kind of robot arm can be made. The greatest assistance is a graphical language and a software tool for quick functional specification, and verification of the application of existing and planned robot models in various environments. The end user, or a person in charge of the application testing, expects to have applications, which without special training, can be used for a variety of robots, and also to automatically document the ability to use them for various tasks.

Figure 4 shows several examples of motions of a robot arm that draws portraits and sketches using a set of curves. Curve parameters are specified by an end user or they are obtained automatically by analyzing the image or 3D objects and recognizing its parts. Figure 4 a) is obtained by combining the basic types of motion in 2D, such as straight lines, arbitrary jumps, circle, sinusoids, impulses, and Bezier curves. The DSL concepts for modeling robot motion logic that allows drawing a portrait on a canvas consists of: a canvas, base type "Motion" and subtypes that match curve types, as well as concepts for describing a robot arm topology. Target framework may be an existing robot control language, but after the construction of the first DSL for the robot motion control, it is already clear that the framework should be also improved and simplified. In our case, the DVRobCon Motion Framework reduces all the motions to the finite set of curves with properties: the start and end point in space, curve type, amplitude height, number of impulses or oscillations, rotation angle, and curve offset. With such a framework, model objects representing motions can be mapped one-to-one into motion framework or library. When a DSL should include concepts such as eye, nose, eyelash, such instances from a model are transformed into $1..M$ commands of the motion framework.

Figure 4 b) shows the motion where a user at the time of motion sends a signal or a command to the robot to draw the straight line of a certain length, relative to the current point, and then return and continue the motion. Such a request is solved by introducing a new language concept named **Deviation Point**, which is not a true subtype of the basic motion types, because it also contains an event (triggering an operation). If a robot arm uses a brush instead of a pencil or pen, some of the lines can be drawn with different thicknesses – thinner to the ends, and thicker in the middle (Figure 4 c)). We introduce a new DSL concept named **EyeLash**, as a modification of the Bezier curve. Considering the domain expert

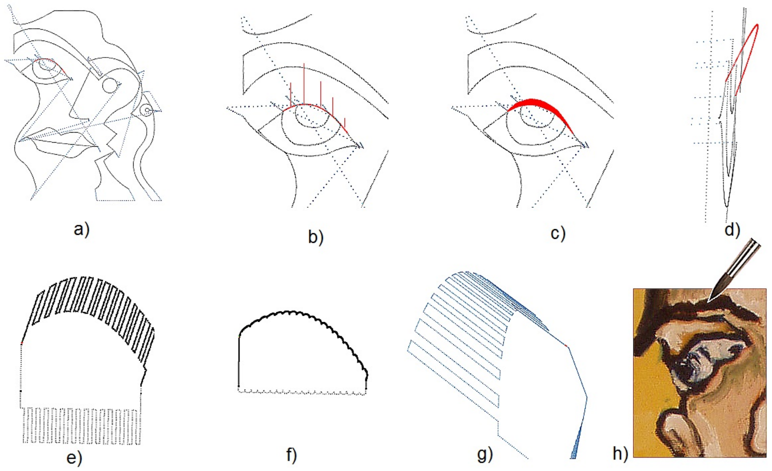

Figure 4. Refinement of robot motion control language

experience (painter) the line from Figure 4 c) can be obtained by setting rotation angle to 70 degrees and line (brush) offset to 5 mm (Figure 4 d)).

Further steps in refinement of the DSL for painting bring a modification of Bezier curve, which contains impulses (Figure 4 e)). In the electronics terms, this is the modulation of a Bezier curve with impulses. The curve performing modulation is called a **modulation pattern**. Therefore, the DSL is extended with the concept **Motion Modulator**. This modulator is a curve that affects one or a group of curves, and depending on the orientation in the space, it produces new, complex or composite motions. Although it is mostly clear, from the mathematical point of view, how to implement such a path, things coalesce when it is necessary to ensure continuous motion of uniform speed, accelerated, or motions which are re-modulated at the run time. Such modulators, even the simplest ones, solve the problems of performing complex 3D operations in CNC machines. These modulations give us the freedom to decide about the portrait while painting, not in advance. In Figures 4 e) and 4 f), modulation patterns are shown in the bottom, while in the top it is shown resulting modulated Bezier curve for EyeLash.

The extension of the DSL for painting from 2D to 3D, i.e., from the DSL for painting to the DSL for sculpting, is presented in Figure 4 g). A modern artist desires wired lashes bent in a pulse-like shape. We introduce a new DSL concept named **3D EyeLashes**, whose shape, dimensions, inclination, thickness, etc., are parameterized.

When it comes to a robot as a painter and Domain-specific Modeling, in the process of refining DSLs, the goal is to create a set of modeling concepts that express important characteristics of

1. painting epochs and directions,

2. techniques and materials, as well as
3. individual characteristics of the painter.

We are convinced that objectives 1 and 2 can be realized to a significant extent with at most two levels of modulation, i.e., modification of the base curves at the time of painting or sculpting. When it comes to the construction of painter-specific language concepts, modulation patterns should also express the motoric, intellectual and emotional properties of painters or sculptors. Such patterns are formed both in advance and at the time of work on a particular painting or sculpture.

Refining the DSL with concepts such as EyeLash that reflects the characteristics of a painting direction or a painter also affects the objects of the control logic and painting environment. Figure 5 shows the modifiers for identifying the subtype of the function block that calculates the points in 3D space (libID), the brush size (disCarpSize), the position of the canvas, the rotation center, and the set of curves for drawing EyeLash (motAndPos). All individual modifiers are grouped into EyeLash, as modification type. Default specifications of user applications are generated in a human-readable XML format for interpreting under different operating systems and hardware platforms (Listing 1). The specification contains several parts:

- submodels, or forms and subforms;
- visual properties of form elements in applications;
- commands for communication with the target RTS, and their invocation rules (cyclic or upon event occurrence);
- commands for the exchanging properties or events between the form elements; and
- list of modifiers with an identifier of the group they belong to.

ModelModifiers contain object modifications related to control logic, arm topology, and objects in the environment, as well as their different layouts. During the interpretation, an end user can select a group or individual modifier, to select view and associated layout. The end user is also allowed to change properties in the execution time. The DSL semantics cannot be changed through user applications, but some properties, having influence on default values, can be changed. Updated properties are taken from the modeling tool in run-time, and serve as the basis for updating default property values or domain definitions. Beside meta-logic and meta-arithmetic, which are described in the next session, feedback gained from the target interpreter of applications is also used for automated refinement. The software architect and end-users simultaneously "debug" the model and the generated code. In the scenario of model execution, or visual debugging, after each change in the model, the increment of program code for the robot controller, user applications and action reports are regenerated. The time elapsed from the change in a model to the new start of control logic and user applications is a few seconds. In most cases, this is also the time spent for demonstrating the code and application validity.

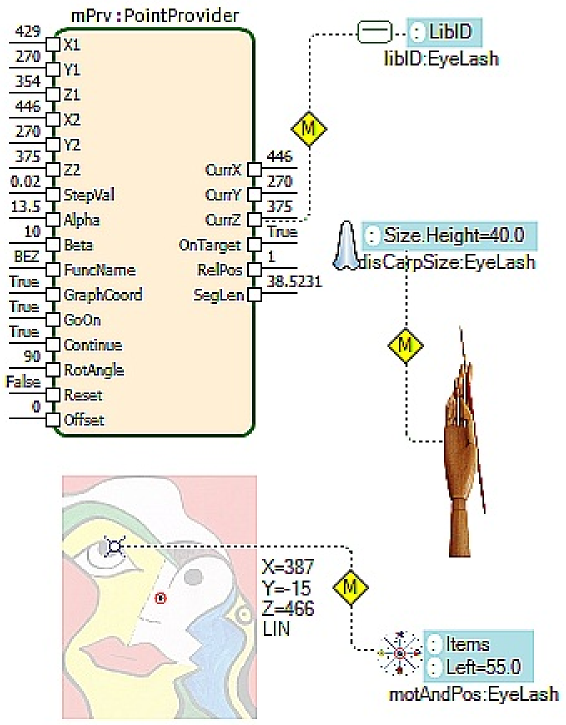

Figure 5. Representing and grouping of modifiers

5 META-LOGIC AND META-ARITHMETIC

By meta-logic and meta-arithmetic we denote a set of operations that provide additional information about the context of the execution of logical and arithmetic operations. In short, meta-logic includes tracking the status of logical operations, while meta-arithmetic includes tracking the status of arithmetic operations. Both operation types calculate the logical value regarding the correctness of operations during execution and, by applying different strategies, may help make the control process as stable as possible. Herein, both operation types are often referred to as meta-logic. The DVMEx approach allows for the definition of meta-logic at each of the levels forming the architecture of the DSM solution:

1. at the level of the meta-model and model, by means of language concepts, by specifying the semantic domains of attributes and model constraints;

2. at the level of code generator, where code is generated for the rules of meta-logic, as well as when rules are not explicitly expressed by the model;

```
<Submodels>
  <Submodel name="Control logic",...>
    <SubmodelElems>
      <Element name="libID" Value="MotLib_2",.../>
    </SubmodelElems>
  </Submodel>
</Submodels>
<Application>
  <Object name="Canvas" Left="60.0", ... />
</Application>
<RTS_Commands>
  <Events>
    <event object="DVRobCon.doMotions" name="OnSetValue">
      <![CDATA[DB SV "DVRobCon.doMotions","(())"]]>
    </event>
  </Events>
  <Cyclic time="200">
    <cmd condition="CycleID=1" action="DB GV (varList)"]]></cmd>
  </Cyclic>
  <DirectMappings>
      <mapping>
        <![CDATA[:.motionSpeed.SendToRTS(Value);]]>
      </mapping>
  </DirectMappings>
</RTS_Commands>
<ModelModifiers>
  <ModelModifier name="motAndPos" GroupID="EyeLash">
    <Object name="Canvas">
      <update_prop Items="..." />
      <update_prop Left="55.0" />
    </Object>
  </ModelModifier>
</ModelModifiers>
```

Listing 1. Specification of an end-user application

3. at the level of the target language compilers (in this case an IEC 61131-3), by using properties that define meta-logic for certain types of data and operations; and
4. at the level of the RTS, by defining filters that determine the detection and reporting rules about the operation status and unexpected states.

In Figure 6 there is an example of meta-logic that is explicitly specified by the model. The rounded upper right section of the figure illustrates the meta-model, i.e., the modifier type, while the central section of the figure is devoted to the model instance. The instance of the function block div2: DIV, which uses division

```
if Type='DIV' then
        'VAR
                validate1:VALIDATE;
        END_VAR'
        /* Code for DIV(slider_1 ,slider_2) */
        validate1:= VALIDATE(div2.OUT,SDef);'
endif
```

Listing 2. An example of program code for validation

to calculate the length of the robot step based on the distance to some object (sliders to the left), produces the undefined value. The domain-specific function block VALIDATE determines the length of the step based on the input value shown (produced) by the scale `Sdef` to the left. In order to resolve the unexpected state of control logic, which is also invalid in this case, function block VALIDATE is used.

In relation to the discussion of the modeling approaches in the previous section, the approach is most similar to the modeling using modifiers. This kind of a DSL is to a great extent an example of a modeling language (RMCL), which is often used in practice. The textual representation (DSL script) of the model featured in Figure 6 may be of the form ⟨`DIV.validate`⟩`valRepl_Val`.

In case the modeling language does not feature VALIDATE, but that kind of a function block is available in the target language or library, the same meta-logic may be specified using a code generator as presented in Listing 2.

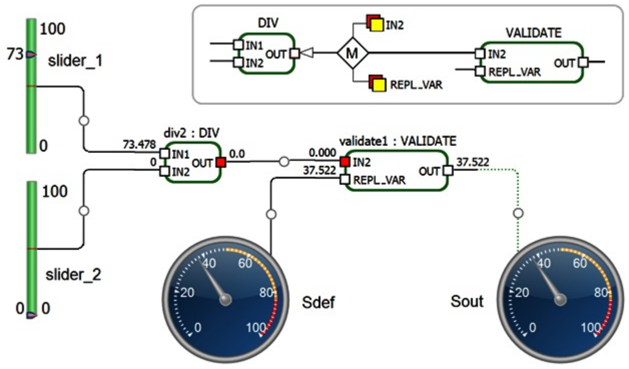

Figure 6. Meta-logic at the level of the meta-model and model

In case function block VALIDATE is not available, the native code that checks the input parameters IN1 and IN2 for DIV is generated. When a compiler is used to implement the meta-logic, it generates additional native code instructions. The native code checks the processor registers to determine the operation status (e.g., overflow, underflow, and divByZero) and assigns the status to a temporary result or variable. Native code for the meta-logic may be of the form shown below. In order to be more readable, Assembly for Intel x86 processors is given in Listing 3. instead. The assembly code sets status for a variable named DIV2. If any of variables used for calculating is invalid, then the variable status will be also invalid.

Irrespective of the level at which the code for meta-logic is provided, we defined several strategies concerning the evaluation of meta-logic expressions (Figure 7):

1. *propagate* – where operation statuses are propagated to all subsequent operations in which some variable or temporary result is used;
2. *reset_assign* – where the status is reset to valid whenever a valid value is assigned to some variable;
3. *reset_cycle* – where the statuses of variables are reset before each new program execution cycle; and
4. *ignore* – where meta-logic is not executed, i.e., operation statuses are not tracked.

In the example in Figure 7, a function block for the data type conversion is used, from a long to real value, but the input lreal value 5e+38 is greater than the maximum real value. The type convertor is marked by `tc:lreal_to_real`, and the symbol above which there is only `tc` is a graphical representation for assigning the constant, which changes the output variable.

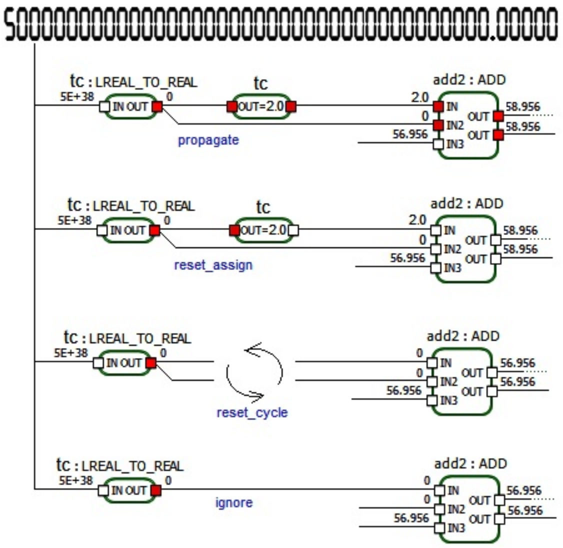

Figure 7. Different strategies concerning the meta-logic

The RTS allows for incremental updating of the code and dynamic change of the meta-logic strategy. For the reliable functioning of control logic, the most suitable

```
mov DIV2$status, 1
mov al, __SLIDER1$status
cmp al, 0
jne ok3
mov __DIV2$status, 0
ok3:
movsx ebx, WORD PTR __SLIDER1
mov al,__ SLIDER2$status
cmp al, 0
jne ok4
mov __DIV2$status, 0
ok4:
movsx ecx, WORD PTR __SLIDER2
cmp ecx, 0
jne ok5
mov __DIV2$status, 0
jmp skip6
ok5:
mov eax, ebx
cdq
idiv ecx
mov ebx, eax
mov __DIV2, bx
```

Listing 3. An example of generated Assembly code

strategy is *reset_assign*. For the testing of control logic and model-level debugging, the most suitable strategy is *propagate*. For the detection of deviations in the status of control logic between the cycles of program execution, the most suitable strategy is *reset_cycle*. The *ignore* strategy is used in well-checked control programs, which are automatically generated from well-checked models using well-checked modeling languages.

6 PLATFORM FOR DSL MERGING AND MODEL-LEVEL DEBUGGING

The efficacy of the construction, testing and application of new DSLs depends on swift and simple utilization and adaption of existing languages (DSL reusability), patterns and target systems that execute specifications. In the context of the theoretical discussion concerning syntax and semantics merging [13], we present our practical solution – a platform that supports model merging and model-level debugging which utilizes merged languages.

For the purpose of modeling in robotics and automation we use three DSLs, which were merged by meta-model integration. In the example from Figure 8, the first DSL (RMCL) is used to model topological properties of arm, foot or body, their motions and actions. The constructs from this DSL are shown in the form of a blue hand. The positions of segments, with the exception of *RootJoint*, are not relevant. However, relationships between segments do matter, as they determine topological characteristics of the hand. The particular layout of the segments was chosen for its more appealing look. For new cases, the construction of a DSL from the group requires several days.

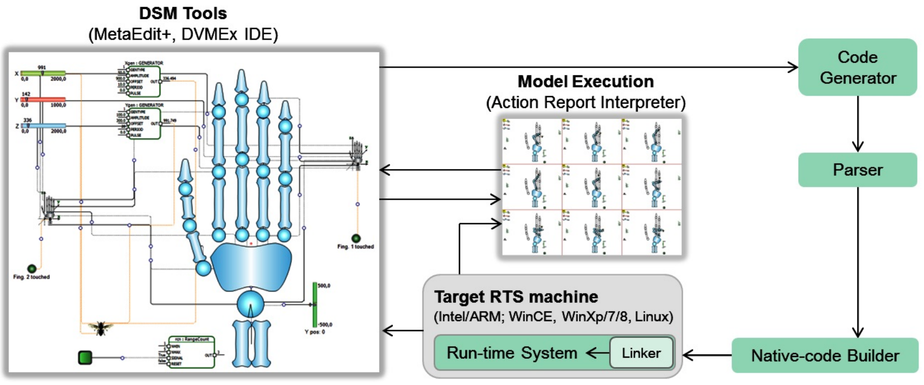

Figure 8. Platform for model execution and model-level debugging

The second DSL is a language for function blocks diagrams. The elements of the language (named FBD DSL) are depicted as green rounded rectangles with input and output ports. This language contains constructs of the target framework according to the IEC 61131-3 specification. It is also extensible and acts as an interface to an auto-adaptive run-time system. Code generators that are specifically written for FBD DSL are generic and applicable to the merged language. FBD DSL supports using a model to describe control logic to the greatest detail (the level of variables and operations), instead of making it part of the code generator. The relationships between objects of different DLSs are expressed by mapping the properties of RMCL objects to the ports of function blocks in FBD DSL and vice versa. The complexity of the model may arise as a result of the presence of objects from different DSLs and redefinitions of linguistic concepts in the same instance (Figure 8, left part). However, this may be resolved through the DSM tool by utilizing different model views (application, control-logic, topological, domain-specific) and applying decomposition.

The third DSL features linguistic concepts that describe objects and properties of the environment in which the robot operates. It is constructed by modifying a set of general-purpose linguistic concepts, such as analog and digital controllers, sensors, scales, switches, sliders, displays, etc. In the previous figure, the elements of this language include sliders in the upper left section and green switches. By using the objects of this DSL, it is possible to simply construct virtual signals and generate controller drivers [11]. This method of generating “domain-specific drivers” from models is beneficial as it leads to better utilization of hardware resources and faster native code, which is comparable to the optimized code provided by a C++ compiler. This DSL is part of the development environment and, therefore, it does not require additional time for its construction. Any additional modification requires at most one day of work.

Visual debugging is a process of executing models that is performed in parallel with editing "on hot" without stopping the execution of the current program within the RTS. On model change, the generated code may be forwarded to the parser or the native code may be directly generated. The target RTS uses a dynamic linker to receive specification increments and links the control logic code to variables. Changes are performed within transactions. The completion of a transaction is reported to the modeling tool by the RTS. Visual tracing is achieved by MERL-like generators (action reports), which change model state within the DSM Tool based on the state of interpretation or program execution. The average time between a model change and the start of the execution of new native code is approximately 200 ms.

7 RELATED WORK

Software engineers demand significantly improved methodologies and tools to develop and maintain reliable robotic applications. Robotic systems are inherently complex, and developers must integrate various software tools and electronics from different manufacturers ([14]). Recently significant efforts have been invested into the research of model-based approaches and tools aimed to facilitate software development in robotics ([15, 16, 17, 18, 19]). Also, numerous successful applications of DSLs are reported in various domains including robotics [20]. Our research is directed toward further evolution of the DSM approach with model execution and modifiers. The model-execution concept has recently attracted significant attention from the academic community. It may be found under different names, such as live programing [21]. Model execution proved to be a powerful means for the dynamic validation and verification of models. We implemented this concept using action reports that are extended with a set of commands for communication with the target RTS [22]. Also, we introduced modifiers and illustrated their use in several examples. In the context of graphical DSLs, modifiers are means to simple and intuitive merging and customization of languages, as well as language application to new problems.

In [23], Hästbacka et al. describe the application of the MDD and DSM approaches to the development of industrial process control applications. Their approach is based on the utilization of the UML Automation Profile modeling concepts (UML AP). Based on our experience, there are several reasons why the application of this approach to the generation of process control applications is limited. First, both UML and the construction of UML profiles are complicated and, to the average user in the domain of automation and process control, they do not offer a faster response to requests or a better insight into the system being modeled. Second, OPC interfaces are not abstract in the manner that they could be described using a simple language. For that reason, the high-level synchronization between the models and the generated process control applications is limited. Third, the approach described by Hästbacka et al. does not provide sufficient attention to run-time systems as interpreters of models that solve numerous shortcomings in the generation of code for GPL compilers, instead of doing that for domain-specific RTSs.

Song et al. [24] introduce an approach for connecting architectural models with run-time systems with bidirectional transformation and automated changing of architectural models according to changes in the run-time system. When compared to the approach of Song et al., our approach does not use architectural but domain specific models and, as a result, it may be suited to a wide variety of audiences, not only software architects. Furthermore, in our approach the implementation and its connection to models are automatically generated, which adds more reliability to the implementation of the approach. Moreover, the approach of Sung et al. allows only changes in the model that can be captured in the architectural language metamodel, while our approach also allows changes in the metamodel, i.e., introduction of new concepts for modeling systems. Finally, our approach also enables incremental changes of models and run-time systems during execution including changes in which novel modeling elements are introduced.

There is an analogy between the two seemingly unrelated domains of application: document engineering and robot-motion control. In both domains, the efficacy of modeling tools depends on their support for merging languages that model different dimensions/aspects of documents or robot control. A document is a multidimensional entity featuring the following dimensions: content, layout, structure, role in a real system and states. In some cases, it may not be possible to generate valid code for control logic, e.g., during robot motion modeling, when a robot step cannot be related to foot shape and other objects in the environment where the robot is moving. However, by using solutions for syntax and semantics merging of DSLs, we may simplify parallel viewing and modeling of different document dimensions or aspects of robot usage. In [25], the authors present some preconditions, as well as problems in merging syntax and semantics, and meta-model inheriting and slicing. On the other hand, our discussion and contributions are more of a practical nature. We introduced modifiers and illustrated their use in several examples. In the context of graphical DSLs, modifiers are means to simple and intuitive merging and customization of languages, as well as language application to new problems.

Another topic of related research is about multilevel modeling. The most recent advances in this field are related to the concept of deep instantiation, supported by DeepJava. The authors in [26, 12] propose a way for avoiding the shortcomings of programming languages that result from their two-level architecture, seen as type–instances paradigm. The proposed concept of deep instantiation is not applicable to the description of model variations because of the following three reasons:

1. it requires the instantiation depth to be specified in advance;
2. it does not support the relationships in which an attribute from the supertype may be removed from the subtype; and
3. it does not support meta-data-based inheritance of the type from an instance.

8 CONCLUSIONS

In model-driven software development (MDSD), meta-model refinement is an activity which improves the current expressiveness of a language, i.e., improves the capacity of the language to precisely express the properties of the real system being modeled. The DVMEx approach provides the means to refine each level of the DSM solution: the DSL, the code generator, the framework, and the RTS. The synchronization between the tools used separately for each level may provide good productivity and validity in the DSL refinement. In most of the existing tools, the synchronization between the levels is unidirectional.

In Figure 9 we outline the process of refinement. The left column containing the ellipses denotes standard activities that are related to DSM. In the right column, there are activities that are specific for the DVMEx approach. At the higher abstraction level, it is the modifier construction, which acts as means to flexible evolutionary extension of the language semantics. The second level is the usage of modifiers. Underneath that level, there is the execution of models or automatically generated applications. For each model, at least one default client application for model debugging and monitoring the model execution is generated. Since arbitrary controls may be used for monitoring by mapping the DSL concepts to user control properties during execution, model variations may be tested very fast. Modifiers may also be used at the level of the code generator. In this manner, there is a temporary solution for problems arising from the imprecisely specified hierarchy of DSL objects or imprecisely described relationships or object roles. During execution, the RTS detects unexpected states. Some of them may be trivial, such as division by zero, but they may also include states that cannot be recognized in the predefined state space of the control logic program.

For the purpose of developing intelligent controllers for robots, we have devised an approach that is based on the DSM approach. We significantly improved each of the levels of the DSM architecture. For the purpose of verifying the DVMEx approach, we created the tools that, together with MetaEdit+, may be used to verify the reliability, flexibility, speed, and the simplicity of the integration of the control logic into an arbitrarily complex real system. In the rest of Conclusion, we list the theoretical and practical contributions, whose application may improve the development of intelligent robot controllers and the development of measurement and control systems in general.

Contributions to the theory. The RTS is conceived as an adaptive system with dynamic scheduling and linking of the code of control logic and meta-logic.

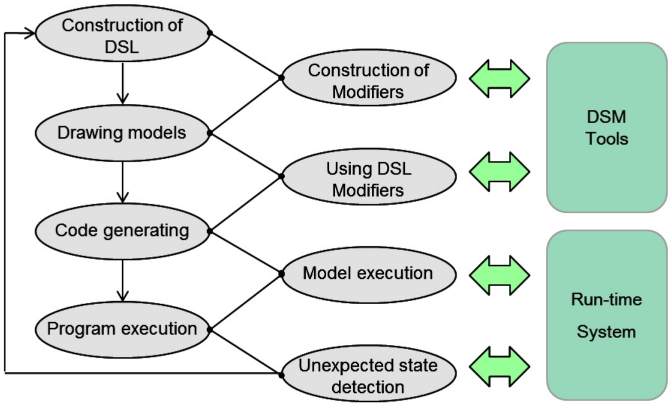

Figure 9. A generic model of DSL refinement

Changes in the rules of scheduling of control tasks in run time, as well as dynamic mapping of the signals of a real system to the variables of the control code allow for full flexibility when changing the purpose and behavior of the robot, even for the states that have not been programmed in advance. The generators are simultaneously used for code generation, debugging, execution and monitoring of the control processes. At the level of the code generator, modifiers may be described, as model or submodel variations, whose purpose is systematic collection and processing of knowledge for the refinement of the DSLs. We extended the IEC 61131-3 with concepts for the description for finite state machines and message exchange protocols, which simplifies the implementation of the event-driven control logic. At the level of DSL construction and usage, we define an approach of evolutionary refinement with modeling variations. We introduced the modifier concept, which integrates inheritance, instantiation and supports multilevel modeling.

Contributions to the practice. We made an auto-adaptive RTS for ARM and Intel hardware platforms. It is applicable to problems involving simple control logic in embedded systems, as well as complex problems, such as automatization of the whole production process or the control of intelligent human, bird or snake-like robots, in which the RTS interprets complex motion models, models of energy consumption and models for the recognition of the signal of a real system.

The IEC 61131-3 compiler with different variants of meta-logic offers a significant advantage over the existing GPL compilers, where the logic has to be embedded into source code. The code generator level provides synchronization of RTS, modeling tools and monitoring applications without the need for programming. This is achieved using the code generators that generate code generators. Despite the fact that nesting a language in another language may be bad because of the lowered readability, this approach to the generation of monitoring applications provides good productivity in software development. The issue of the lower readability of the generators that generate generators is straightforwardly overcome within tools that support the construction of DSLs (meta-modeling) using a graphical interface. The semantics of such generators is described at the level of the meta-model. All components that are part of the DVMEx solution are simply integrated into various tools for meta-modeling and modeling.

In addition to software engineering, validity of the approach, devoted to handling variation of products and DSL models describing these products, has been demonstrated in electronics, and partly mechanics. We made several usable prototypes of controllers for managing robots with sensors, for which drivers and virtual signals are also generated from the model. Also, we have developed several variants of pneumo-mechanical grippers for robots in the textile industry, which perform complex operations in a confined space. Model-level debugging is not limited to generating and debugging the control logic and user applications program code, but it is also applicable to debugging in the electronics and mechanics domain.

Our further research work is aimed at extending the DVMEx approach to the needs of the development of intelligent robots of various shapes and purposes, but primarily for those that need to learn the command language during task execution and provide information about the need to refine these languages.

Acknowledgement

The authors would like to kindly thank Dr. Juha-Pekka Tolvanen from the University of Jyväskylä for his valuable support and proof reading.

REFERENCES

[1] Kelly, S.—Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, 2008. ISBN: 978-0-470-03666-2.

[2] MetaEdit+ Workbench, Workbench User's Guide.

[3] Robot Operating System (ROS). Availaible at: `http://wiki.ros.org/`.

[4] Djukić, V.—Luković, I.—Popović, A.—Ivančević, V.: Model Execution: An Approach Based on Extending Domain-Specific Modeling with Action Reports. Computer Science and Information Systems (ComSIS), Vol. 10, 2013, No. 4, pp. 1585–1620, doi: 10.2298/CSIS121228059D. ISSN: 1820-0214.

[5] International Standard IEC 61131-3: Programmable Controllers – Part 3: Programming Languages. International Electrotechnical Commission, 2003.

[6] International Standard IEC 61499: Function Blocks, Part 1–Part 4. International Electrotechnical Commission, 2005.

[7] Djukić, V.: DVDocGen Framework, Application Interface. 2009, 88 pp. Availaible at: `http://www.dvdocgen.com/Framework/DVDocFramework.pdf`.

[8] Kim, D.-K.—France, R.—Ghosh, S.: A UML-Based Language for Specifying Domain-Specific Patterns. Journal of Visual Languages and Computing, Vol. 15, 2004, No. 3-4, pp. 265–289, doi: 10.1016/j.jvlc.2004.01.004.

[9] Schaefer, C.—Kuhn, T.—Trapp, M.: A Pattern-Based Approach to DSL Development. SPLASH '11, Workshop on DSM '11, 2011, pp. 39-46, doi: 10.1145/2095050.2095058.

[10] Common Variability Language (CVL), CVL 1.2 User Guide. Availaible at: `http://www.omgwiki.org/variability/doku.php`.

[11] Djukić, V.: Various Demos of DSL Construction, Application and Refinement in Robotics, Automation and Design of Medical Devices. Availaible at: `https://www.youtube.com/channel/UCqyYnYD6J5fEeb6Ni3YLuKg`.

[12] Kühne, T.—Schreiber, D.: Can Programming Be Liberated from the Two-Level Style: Multi-Level Programming with DeepJava. Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07), 2007, pp. 229–244, doi: 10.1145/1297027.1297044.

[13] Degueule, T.—Combemale, B.—Blouin, A.—Barais, O.—Jézéquel, J. M.: Melange: A Meta-Language for Modular and Reusable Development of DSLs. Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015), 2015, pp. 25–36, doi: 10.1145/2814251.2814252. ISBN: 978-1-4503-3686-4.

[14] Djukić, V.—Popović, A.—Tolvanen, J.-P.: Domain-Specific Modeling for Robotics – from Language Construction to Ready-Made Controllers and End-User Applications. Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering (MORSE '16), Leipzig, Germany, ACM, 2016, pp. 47–54, doi: 10.1145/3022099.3022106. ISBN: 978-1-4503-4259-9.

[15] Trojanek, P.: Model-Driven Engineering Approach to Design and Implementation of Robot Control System. 2nd International Workshop on Domain-Specific Languages and Models for ROBotic Systems (DSLRob '11), 2011.

[16] Piechnick, C.—Götz, S.—Schöne, R.—Assmann, U.: Model-Driven Multi-Quality Auto-Tuning of Robotic Applications. Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-Based Software-Engineering, L'Aquila, Italy, ACM, 2015, pp. 35–40, doi: 10.1145/2802059.2802063.

[17] Adam, K.—Butting, A.—Heim, R.—Kautz, O.—Rumpe, B.—Wortmann, A.: Model-Driven Separation of Concerns for Service Robotics. Proceedings of the International Workshop on Domain-Specific Modeling (DSM 2016), Amsterdam, Netherlands, ACM, 2016, pp. 22–27, doi: 10.1145/3023147.3023151. ISBN: 978-1-4503-4894-2.

[18] Pradhan, S. M.—Dubey, A.—Gokhale, A. S.—Lehofer, M.: CHARIOT: A Domain Specific Language for Extensible Cyber-Physical Systems. Proceedings of the Workshop on Domain-Specific Modeling (DSM 2015), SPLASH '15, Pittsburgh, USA, ACM, 2015, pp. 9–16, doi: 10.1145/2846696.2846708. ISBN: 978-1-4503-3903-2.

[19] Saglietti, F.—Meitner, M.: Model-Driven Structural and Statistical Testing of Robot Cooperation and Reconfiguration. Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering (MORSE '16), Leipzig, Germany, ACM, 2016, pp. 17–23, doi: 10.1145/3022099.3022102. ISBN: 978-1-4503-4259-9.

[20] Nordmann, A.—Hochgeschwender, N.—Wrede, S.: A Survey on Domain-Specific Languages in Robotics, Simulation, Modeling, and Programming for Autonomous Robots. In: Brugali, D., Broenink, J. F., Kroeger, T., MacDonald, B. A. (Eds.): Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014). Springer, Cham, Lecture Notes in Computer Science, Vol. 8810, 2014, pp. 20–23, doi: 10.1007/978-3-319-11900-7_17.

[21] van Rozen, R.—van der Storm, T.: Model Execution: Toward Live Domain-Specific Languages: From Text Differencing to Adapting Models at Run Time. Software and Systems Modeling, Vol. 18, 2019, No. 1, pp. 195–212, doi: 10.1007/s10270-017-0608-7. ISSN: 1619-1374.

[22] Djukić, V.—Popović, A.—Lu, Z.: Run-Time Code Generators for Model-Level Debugging in Domain-Specific Modeling. Proceedings of the International Workshop on Domain-Specific Modeling (DSM 2016), Amsterdam, Netherlands, ACM, 2016, pp. 1–7, doi: 10.1145/3023147.3023148. ISBN: 978-1-4503-4894-2.

[23] Hästbacka, D.—Vepsäläinen, T.—Kuikka, S.: Model-Driven Development of Industrial Process Control Applications. The Journal of Systems and Software, Vol. 84, 2011, No. 7, pp. 1100–1113, doi: 10.1016/j.jss.2011.01.063.

[24] Song, H.—Huang, G.—Chauvel, F.—Xiong, Y.—Hu, Z.—Sun Y.—Mei, H.: Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach. Journal of Systems and Software, Vol. 84, 2011, No. 5, pp. 711–723, doi: 10.1016/j.jss.2010.12.009.

[25] Tolvanen, J.-P.—Kelly, S.: Integrating Models with Domain-Specific Modeling Languages. Proceedings of the 10th Workshop on Domain-Specific Modeling (SM '10), Reno, Nevada, USA, 2010, Art. No. 10, doi: 10.1145/2060329.2060354.

[26] Atkinson, C.—Kühne, T.: The Essence of Multilevel Metamodeling. In: Gogolla, M., Kobryn, C. (Eds.): UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools (UML 2001). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2185, 2001, pp. 19–33, doi: 10.1007/3-540-45441-1_3.

Verislav Djukić is Software Architect employed in Djukic Software GmbH, Germany. He is involved in the development of modeling tools for robotics and automation and PLC based runtime systems for the purpose of model execution. He holds his Ph.D. in software engineering from the University of Novi Sad, Serbia. His current research interests are related to the handling of model variations in DSM tools and construction of robot motion framework in humanoid robot painting.

Aleksandar Popović graduated from the Faculty of Science at the University of Montenegro. He completed his Master's degree (2 year) at the University of Novi Sad, Faculty of Technical Sciences. He received his Ph.D. in computer science from the University of Montenegro in 2013. He is Assistant Professor at the Computer Science Department of the Faculty of Science and Mathematics, University of Montenegro. His current research interest includes domain-specific languages and domain-specific modeling. Also, he has been actively involved in the development of DSLs for embedded and real-time systems.

Ivan Luković received his graduate diploma degree (5 years) in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his Masters's degree (2 year) at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he is Full Professor at the Faculty of Technical Sciences of the University of Novi Sad, where he is Lecturer in several Computer Science and Informatics courses. He is the head of B.Sc. and M.Sc. study programs in Information Engineering – Data Science. His research interests are related to database systems, business intelligence systems, and software engineering. He is the author or co-author of over 150 papers, 4 books, and 30 industry projects and software solutions in the area.

Vladimir Ivančević is Assistant Professor in applied computer science and informatics at the Faculty of Technical Sciences, University of Novi Sad, Serbia. His main research areas include data science, databases, and information systems. He has participated in diverse research projects involving application of computer science and informatics in education, public health and epidemiology, and software engineering.