Domain-Specific Language Architecture for Automation Systems: An Industrial Case Study

Christopher Preschern, Andrea Leitner, and Christian Kreiner

Institute for Technical Informatics
Graz University of Technology, Austria
christopher.preschern@tugraz.at
andrea.leitner@tugraz.at
christian.kreiner@tugraz.at
http://www.iti.tugraz.at

Abstract. This paper presents a domain-specific language (DSL) design for automation systems. We describe basic components of the language, its mapping to automation devices and to automation software elements. The DSL design achieves low domain model complexity and is easy to maintain. Furthermore, it allows easy and intuitive modeling of systems in a domain. We present an industrial case study using the proposed DSL design and evaluate it regarding its maintainability and complexity. For this evaluation we use existing metrics to evaluate the domain model complexity and we introduce novel metrics to evaluate the code generator complexity.

Keywords: domain-specific language, automation system, domain model metrics, code generator metrics

1 Introduction

Domain-specific languages (DSL) allow product modeling on a high level of abstraction and enable structured software reuse through code generation from these models. To develop a DSL, a meta-model has to be constructed for a specific product family. Meta-model development requires careful design of the domain model structure and the mapping of DSL elements to artifacts in the solution space. This is a sophisticated task and several guidelines on how to construct a good domain model exist. Such guidelines can be more detailed if they just address specific domain families, but are rarely present for domain families where DSLs are not often applied. An example for such a domain family where no detailed guideline for DSL development exists is the automation domain.

In this paper we present a flexible design to develop automation system DSLs. We discuss design decisions and their rationale concerning the meta-model and the mapping of DSL elements to automation devices and to generated artifacts like the automation system software. We present and evaluate PISCAS (Pisciculture Automation System), an industrial case study which applies the discussed

DSL design guidelines. For the evaluation of the DSL complexity, we use existing domain model metrics and we introduce novel metrics to measure the code generator complexity. Furthermore, we evaluate the DSL in terms of modeling effort and maintainability.

2 Related Work

Issues regarding the construction of domain models for automation systems are discussed in [9], where experiences with different domain model granularities are presented. Hierarchical, nested domain models are suggested for automation systems to provide different levels of granularity and abstraction. In Leitner's work [8] an evaluation method for the domain model complexity for DSLs and for feature oriented modeling is presented. We use the proposed DSL metrics in our domain model evaluation.

Graphical domain-specific languages are used in [4] to model home automation systems. Eclipse GMF is used to create DSLs where systems can be modeled on different levels of abstraction which is shown on an industrial case study. A graphical DSL for automation systems in the railway domain is presented in [3] where special focus is put on safety constraints of the system. Here, the DSL is used as a formal specification of the system containing system verification functionality. The MetaEdit+ tool suite is used in [2] to model high rack warehouse information systems. These literature examples show case studies for automation system DSLs. None of them, however, handles the topic on a more abstract level and discusses generic design decisions for these DSLs.

Automation system modeling is handled on a more general level by the Christian Doppler Laboratory in Linz, Austria. They developed a tool for variability management and show several case studies in the automation domain [1]. In [11] a textual DSL for general automation systems is suggested. In a more recent work of the Doppler Laboratory, a DSL is applied to the automation domain using hierarchical structuring of the domain model [10]. Compared to our paper they do not focus on specific automation domains, but address generic automation system solutions.

3 Domain-Specific Language Design for Automation Domains

In this section we present general design decisions for the development of automation system DSLs. First we present the required meta-meta-model which we later use to provide guidelines for the development of a DSL for automation domains. We present the DSL mapping to automation devices and to the automation software. Finally, we discuss the rationale of the presented design decisions.

3.1 Meta-Meta-Model

The DSL design suggested in the next section requires the GOPPRR (*Graph-Object-Property-Port-Role-Relationship*) meta-meta-model [6]. This meta-meta-model allows defining the meta-model in form of a DSL. *Objects* as basic DSL elements can be connected with *Relationships* which define a *Role* for the connection to an *Object*. The connection to an *Object* can be further refined by a *Port* to which the connection is attached. The *Port* is attached to the *Object*, while the *Role* is attached to the *Relationship*. *Objects* and their *Relationships* can be gathered in a *Graph*. *Properties* can be added to each of these elements (*Object*, *Relationship*, *Role*, *Port* and *Graph*).

3.2 DSL Design

Physical automation devices connected to the automation hardware (e.g. to the PLC) are represented as basic DSL *Objects*. Variants for device types are defined by *Properties* of an *Object*. Automation I/O modules are also modeled as *Objects*. Wire connections between the automation devices are directly modeled as *Relationships* between *Objects*. The semantics of the *Relationship* is given by the *Port* it is connected to. Each DSL *Object* has a minimum set of basic elements: *Property* 'Name', *Property* 'Voltage', *Port* 'Input', *Port* 'Output'. This set of basic DSL elements represents our meta-model for the automation domain which is shown in Figure 1. Additional *Relationships* or *Ports* can be used to connect concrete objects with additional semantics, but concrete *Objects* still have to adhere to the rules specified for the abstract automation domain object.

All GOPPRR entities apart from *Roles* are used in the mapping of DSL elements to the automation domain. When using the proposed DSL design, *Roles* might still be of interest for some domains which might need additional semantics for their interfaces. The *Port* entity of GOPPRR is especially useful due to its straight semantic mapping to wire connections of automation devices.

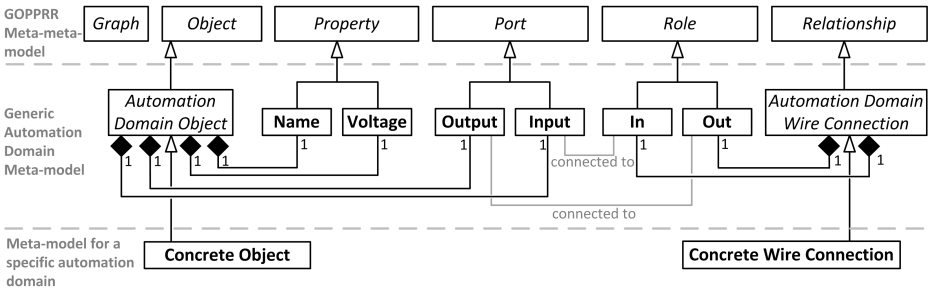


Fig. 1. Suggested meta-model for the automation domain

In the automation software, each *Object* is represented by a function block. Function blocks at least implement the interface variables 'input' and 'output' which represent the corresponding *Ports* in the DSL. Function Block parameters allow configuring the variants modeled with *Object Properties*. *Relationships* between DSL *Objects* are mapped to function block connections (e.g. in a function block diagram). Table 1 contains an overview of the mapping between the DSL, the automation software, and the physical automation devices.

Figure 2 illustrates this mapping and shows how two different aspects of the automation system (physical devices and source code) can be represented by the DSL using the proposed design. This direct mapping between the DSL, the automation software, and the physical devices is possible, because of the nature of the automation domain, which already provides a tight relationship between concepts in the physical world such as physical wires, and corresponding concepts in the automation software such as function block connections representing wires.

Physical system	GOPPRR concepts	Automation software
automation plant	*Graph*	overall software
device	*Object*	function block
wire	*Relationship*	connecting function block interface variables
-	*Role*	-
wire connection	*Port*	function block interface variables
device attribute	*Property*	function block parameters

Table 1. Mapping of the physical system to GOPPRR concepts and to the automation software

3.3 DSL Design Rationale

Graphical DSL - choosing a graphical model representation allows capturing information about the assembly and position of physical objects. This information can be used to generate the system documentation and the visualization, which is an essential part for automation systems.

Directly mapping of physical devices to DSL Objects - Directly mapping physical devices to DSL Objects and physical wires to DSL *Relationships*, makes modeling of the automation system more intuitive for domain experts, because then the DSL is quite similar to function block diagrams which are well known to automation system developers.

Explicit modeling of I/O modules integrates the physical view of the automation system into the DSL and allows capturing all physical wires of the project. Information on the system topology and the electrical wiring is then

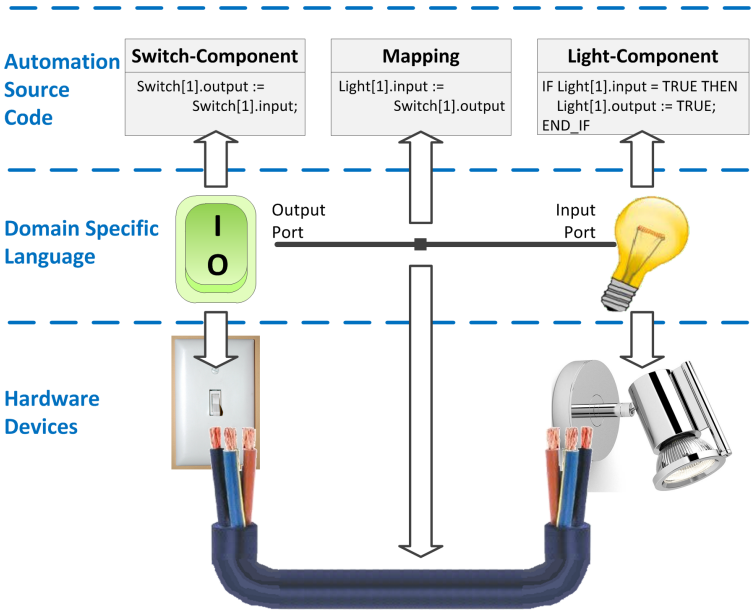


Fig. 2. Mapping between the DSL, physical devices, and automation software

present in the model. Explicit modeling of the I/O modules is not absolutely necessary, because the information about the I/O module type to which a device has to be connected to, is implicitly present in the device type. This would allow automatic generation of the wiring connections, which might not always be desired. To model systems with already installed hardware, this automatic wiring plan generation would most likely be inconsistent. The labels for wiring closets can be generated from the information of the module connections. The *Property* 'Voltage' adds the necessary information to a model with explicitly modeled I/O modules for generating the complete wiring plan for the automation system as well as the list of hardware parts required for the automation project.

Abstract DSL Object - The convention that each *Object* has an 'Input' and 'Output' *Port* and a *Property* 'Name', makes the code generators much simpler, because they can use this abstract *Object* interface. The generators for the visualization, the graphical system overview in the documentation, and the mapping between automation system function blocks can be made independent from the *Object* type. Therefore, adding new *Object* types to the DSL or changing existing ones does not affect those code generators. The reason for explicitly making *Object* types rather flexible is that in the automation domain the basic automation elements, which are represented as *Objects* in our meta-model, change rather often [9].

4 Case Study: PISCAS

In this section the industrial case study is presented. The automation domain and the developed DSL are described and advantages regarding the use of the proposed DSL design are evaluated.

PISCAS is a product line for fish farm automation systems. The project was carried out as a master's thesis at the Institute for Technical Informatics at Graz University of Technology [12]. The core functionality of PISCAS includes water oxygen control and fish feeding. Additionally, water level supervision including an alarm system and standard automation system functionality like steering actuators, such as lights, are part of PISCAS. The automation system can be controlled and configured with a visualization integrated into a web portal. Typically fish farms just vary in the amount of ponds and the functionality for ponds like feeding and oxygen supervision. Fish farm automation elements are rather independent from each other and do not interact a lot. Further information on the PISCAS project can be found on the PISCAS website[1].

A graphical DSL was developed to model fish farm projects. The information in the model is used to parametrize a generic fish farm automation software. Fully executable automation code is created including the hardware mapping and the visualization of the automation project. Additionally PISCAS generates a system documentation including an electrical wiring plan and a list of needed hardware components. Configuration files for the web portal and for network devices are generated. For the web portal, SQL configuration files needed for system initialization are generated. The network routers installed for PISCAS automation systems require configuration of a VPN connection which is needed for remote maintenance of the fish farm automation systems. Furthermore, Configuration files for the router to set up this VPN connection are generated. Figure 3 gives an overview of generated artifacts.

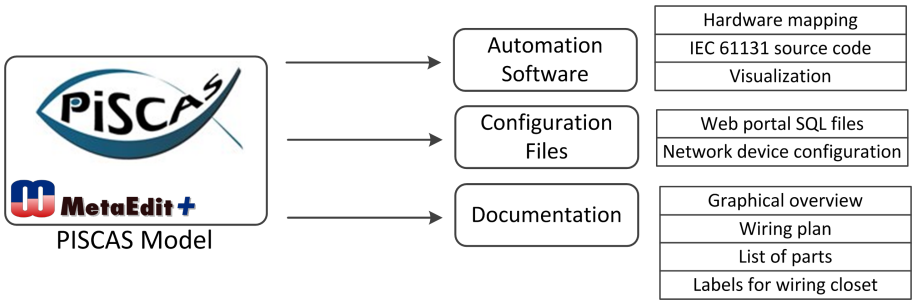


Fig. 3. PISCAS - generated artifacts

[1] http://www.piscas.eu

Bernecker+Rainer[2] (B&R) automation products were used for the PISCAS project. The reason for choosing B&R is that compared to other automation system vendors, the B&R software is easier to generate. All project files including the visualization and hardware mapping files are stored in XML format. Therefore, they are easy to parse and to modify. Metaedit+[3] was used for development of the DSL and for system modeling. Several DSL tools were evaluated according to an evaluation method suggested in [7]. The tools were evaluated regarding technical, management, and product line related criteria. The full evaluation is available in [12]. MetaEdit+ implements the GOPPRR meta-metamodel [5] and is therefore suitable to apply the DSL design proposed in Section 3.2.

4.1 PISCAS DSL

The PISCAS DSL consists of *Objects* representing basic fish farm automation devices, such as a feeder or an oxygen control unit. Each *Object* implements an abstract *Object* definition as presented in Figure 1. The abstract object consists of two *Ports* (Input and Output) and two *Properties* (Name and Voltage). Different variants of a concrete *Object* (e.g. different feeder types) are configured via additional *Properties*. The DSL consists of one *Relationship* type (wire connection) and two *Roles*. A *Relationship* represents an actual physical wire connection.

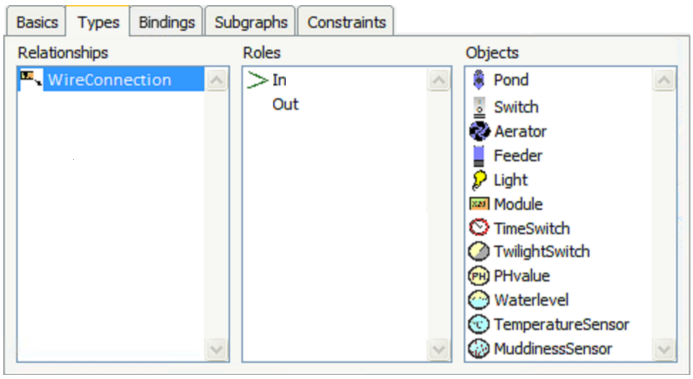


Fig. 4. MetaEdit+ DSL elements of the PISCAS language

Figure 4 gives an overview of PISCAS language elements. PISCAS consists of 7 basic DSL elements (2 *Roles*, 1 *Relationship*, 2 *Ports*, (at least) 2 *Properties*). These elements are needed to use the proposed DSL design. Additionally, 18 *Properties* and 12 *Objects*, are used for domain-specific elements. Therefore, the PISCAS DSL consists of overall 37 elements. Code generators are kept as

[2] http://www.br-automation.com
[3] http://www.metacase.com

independent as possible from the *Object* type. This means that each *Object* provides a well defined interface (defined minimum set of *Properties* and *Ports*) which is accessed by the code generators. General code for the generation of the visualization, the wiring plan, the documentation and the automation software function block parametrization and connection can be generated by *Object* independent generators which access this interface. The DSL is independent from basic functional changes or bug fixes in the automation code, because model is just used to configure a generic fish farm automation software. This generic software has a well defined interface to the DSL through the function blocks and their interface variables. Therefore, the generic fish farm automation software can be maintained independently from the DSL as long as the interface between the automation software and the DSL is not affected by a change. Figure 5 shows an example for a fish farm model constructed with the PISCAS DSL.

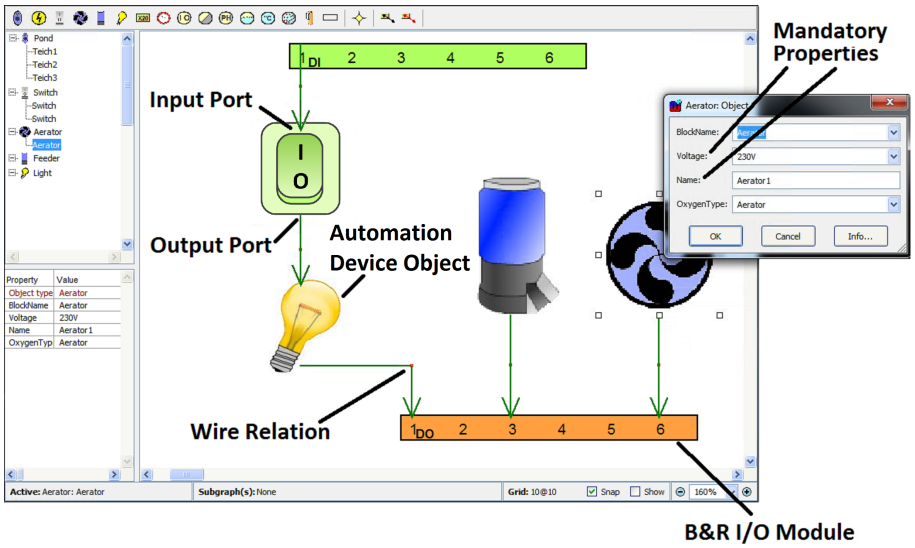


Fig. 5. Example for a PISCAS fish farm model in MetaEdit+ Modeler

4.2 Evaluation

This section contains experiences from applying the suggested DSL decisions presented in Section 3.2 to the PISCAS project.

Application modeling - Two fish farm systems were modeled with the PISCAS DSL and are currently in operation. Both systems could actually be modeled during meetings with the fish farm owner. This was possible due to the intuitive

	Fish Farm element modeling (ponds, switches, lights, ...)	B&R I/O module modeling
Fish farm A	2h	3h
Fish farm B	1h	1.5h
Add new components to B (model approximately doubled)	1h	2h

Table 2. Time spent on application modeling for the PISCAS systems

system representation and allowed the fish farm owner to directly check the fish farm DSL model.

Explicitly modeling the hardware connections took the most time during the modeling process. Table 2 shows the effort (in hours) required for system modeling for different PISCAS fish farms. The high modeling effort is caused by the high number of connections in the hardware mapping. To reduce this effort, we suggest to generate the hardware mapping in the model automatically the first time. The mapping can still be changed in the model after initial generation.

Bug fixes - Most PISCAS changes were related to bugs in the automation software. Therefore, decoupling automation software maintenance from the DSL maintenance is very important for PISCAS and allows decreasing the overall maintenance effort.

For the generation of a concrete PISCAS system, a generic fish farm automation software is configured with the information in the DSL model. The generic automation software is a complete automation program which compiles and can be used to maintain and debug the automation code without the need to work with the DSL tools. This from the DSL decoupled, generic code allowed easy bug fixes and did not require PISCAS DSL modifications often.

Decoupling the generic automation software allows to test new features in the generic automation system without the need work with MetaEdit+. Therefore, MetaEdit+ did not have to be installed on the computer which was used to develop the fish farm automation software. New automation code can easily be integrated later on into the DSL as long as the interface constraints regarding the automation software (physical automation elements are mapped to configurable function blocks with input and output interfaces representing wire connections) are met. Taking this thought one step further, the generic automation software could even be programmed by someone who does not construct the DSL, as long as the constraints regarding the automation code interfaces where the DSL is mapped to, are met.

DSL Complexity - To assess the complexity of the DSL, two different metrics were used. One metric describes the domain model complexity and one describes the code generator complexity.

The domain model was assessed with the metrics suggested by Leitner [8]. The domain model complexity consists of values describing the complexity of interfaces, elements, and properties. These complexities are defined in general and explicitly for the GOPPRR meta-meta-model used in MetaEdit+. The metrics are shown in Equation 1 where C stands for the complexity and n is the number of items. We modified the element complexity metric, by taking the number of *Ports* (n_{Port}) into account. This number is added to the element complexity, because *Ports* are attached to *Objects*. The reason for preferring these metrics to other domain model metrics such as presented in [13] is that they separately handle the complexity of DSL interfaces and DSL elements which allows us to reason about the element complexity, which is especially interesting in the automation domain due to the high semantics these elements usually carry [9].

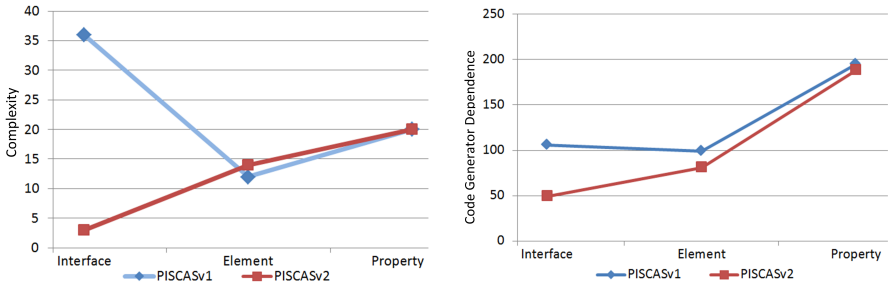
$$\begin{aligned} C_{interface} &= n_{Relationships} + n_{Roles} + n_{Constraints} \\ C_{element} &= n_{Objects} + n_{Ports} \\ C_{properties} &= n_{Properties} \end{aligned} \tag{1}$$

To assess the code generator complexity, Leitner's metrics [8] have been adopted. The interface complexity of the code generators consists of the number of lines of code ($\#LOC$) where any *Role* or *Relationship* type is explicitly used in the generator source code. The element complexity describes the same for the number of *Objects* and *Ports* and the properties complexity handles the occurrence of *Property* types in the code generators. Equation 2 shows these metrics which describe the dependence (D) of the code generators on DSL items. The code generator dependence is a metric describing the affect of domain model changes on the code generator. A lower code generator dependence value suggests less necessary changes in the code generators if DSL items are changed.

$$\begin{aligned} D_{interface} &= \#LOC_{Relationships} + \#LOC_{Roles} + \#LOC_{Constraints} \\ D_{element} &= \#LOC_{Objects} + \#LOC_{Ports} \\ D_{properties} &= \#LOC_{Properties} \end{aligned} \tag{2}$$

We calculated the two metrics for two versions of the PISCAS DSL. The first version (PISCASv1) did not follow the DSL design guidelines given in Section 3.2. The interface *Roles* carried semantic information which was in some cases redundant. In some other cases this semantic information was later, in the second version, put into the simple 'input' and 'output' *Ports*. The second version (PISCASv2) is a refactored version of PISCASv1 and adheres to the guidelines given in this paper. *Objects* follow the specified interface conventions (input *Port*, output *Port*, name *Property*, voltage *Property*) and, therefore, most *Relationships* became unnecessary for the PISCAS DSL. This makes the domain model a lot simpler, because many *Roles* could be deleted. For the transition from PISCASv1 to PISCASv2, the two *Port* types 'input' and 'output' had to be added. The *Properties* and the number of *Objects* did not change. Most of the *Roles* were removed leading to a much lower interface complexity (see Figure 6(a)). This reduced domain model interface complexity, obviously, lead to a decrease of the code generator interface dependence (see Figure 6(b)).

The code generator element dependence also decreased even though the element complexity of the domain model increased. The lower code generator dependence suggests, that the DSL can easier be modified, because less changes to the code generators are required if domain model items are changed. In particular changes of *Objects* in the domain model seem to have lower affect on the code generators in PISCASv2 due to the decreased code generator object dependence.



(a) Complexity of the PISCAS domain models

(b) Dependence of the code generators from the domain model

Fig. 6. Complexity of the PISCAS domain models

5 Conclusion

In this paper, we presented design decisions to develop a flexible DSL for automation systems. We presented the domain model design and the mapping of automation elements to the DSL. The presented design decisions can be taken as a guideline for automation system DSL developers and can help to develop a flexible and easily maintainable automation DSL.

The proposed DSL design was applied to a fish farm automation system DSL (PISCAS) which was evaluated in terms of modeling effort and maintainability. The maintainability is evaluated by measuring the DSL complexity with domain model and code generator complexity metrics. The introduced code generator dependence metric used for this evaluation works very well to describe the PISCAS code generator complexity. For future work it would be very interesting to evaluate the maturity of the proposed code generator metric by applying it to other code generator based systems. It would also be of high interest to evaluate the proposed DSL design, by developing automation DSLs in other domains by following the proposed DSL guidelines.

Acknowledgments. We would like to thank the company HOFERNET IT Solutions and the FFG for financing the PISCAS project with an 'FFG Innovationsscheck'.

References

1. Dhungana, D., Grünbacher, P., Rabiser, R.: The dopler meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering 18 (2011)
2. Haselsberger, A.: Design and implementation of a domain specific architecture for programmable logic controllers. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2009)
3. Haxthausen, A.E., Peleska, J.: A domain specific language for railway control systems. In: Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology. ACM (2002)
4. Jiménez, M., Rosique, F., Sánchez, P., Álvarez, B., Iborra, A.: Habitation: A Domain-Specific Language for Home Automation. IEEE Software 26 (Jul 2009)
5. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment. In: Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering. Springer (1996)
6. Kern, H., Hummel, A., Kühne, S.: Towards a Comparative Analysis of Meta-Metamodels. In: 11th Workshop on Domain-Specific Modeling. ACM (2011)
7. Leitner, A.: A software product line for a business process oriented IT landscape. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2009)
8. Leitner, A., Kreiner, C., Weiß, R.: Analyzing the complexity of domain models. In: IEEE International Conference and Workshop on the Engineering of Computer Based Systems (2012)
9. Maga, C., Nasser, J., Göhner, P.: Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity. In: 18th World Congress of the International Federation of Automatic Control (IFAC). vol. 18 (Aug 2011)
10. Prähofer, H., Hurnaus, D.: Monaco - a domain-specific language supporting hierarchical abstraction and verification of reactive control programs. In: 8th IEEE International Conference on Industrial Informatics (2010)
11. Prähofer, H., Hurnaus, D., Wirth, C., Mössenböck, H.: The Domain-Specific Language Monaco and its Visual Interactive Programming Environment. In: IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE (2007)
12. Preschern, C.: PISCAS - A Pisciculture Automation System Product Line. Master's thesis, Graz University of Technology, Institute for Technical Informatics (2011)
13. Rossi, M., Brinkkemper, S.: Complexity metrics for systems development methods and techniques. Information Systems 21(2), 209–227 (Apr 1996)