Process-Aware Model-Driven Development Environments

Levi Lúcio[1], Saad Bin Abid[1], Salman Rahman[1], Vincent Aravantinos[1], Ralf Kuestner[2], and Eduard Harwardt[2]

[1] fortiss GmbH
{lucio,abid,aravantinos}@fortiss.org, salman.rahman@tum.de
[2] Diehl Aerospace
{ralf.kuestner,eduard.harwardt}@diehl.com

Abstract. Due to recent advances in Domain Specific Language (DSL) workbenches, it has become possible to build model-driven development environments as sets of individual DSLs that get composed for a specific purpose. In this paper we explore how model-driven development environments can become process-aware, to assist the user when building a model. We offer an explanation to our ideas at three levels of abstraction: 1) the *meta-meta* level, where brick DSLs are built using the Meta-Programming System (MPS) workbench; 2) the *meta* level, where brick DSLs are assembled into frameworks that are further tailored for particular modelling scenarios through the introduction of an explicit process for model construction; and 3) the *model* level, where models are built through progressive tool-guided refinements and automated steps based on the process introduced at the *meta* level. We exemplify our approach by providing the main highlights of the ongoing development of a model-driven requirements gathering environment for our industrial partners.

1 Introduction

The current trends in domain specific software engineering, domain specific languages (DSL) and domain specific modelling languages (DSML) demonstrate the interest for *tailored* software solutions. When it comes to model-driven engineering (MDE) tools, studies like [21] show that this trend is justified: among the MDE tools considered in the reported study, the ones which were successful in penetrating industry were precisely those which were developed in a tailored manner for a given audience, the recommendation being: "Match tools to people, not the other way around".

Many technologies now precisely enable this sort of tailoring, among which JetBrains' MPS [16], Xtext [11], Sirius [12], or the classic MetaEdit+ [19]. With the maturity of these technologies, one can safely say that building your own MDE tool has never been so easy – technology enthusiasts in various industries now develop their own domain- or even company-specific tools.

On the other hand, as [21] also mentions, tools are an enabler, but they are not everything: "More focus on processes, less on tools". Even when a tailored tool is available, allowing the modelling of one's domain through many sub-DSLs makes it such that new users often become overwhelmed by the amount of modelling techniques at their disposal. This is in fact the case for even very specialized tools like Sfit [9] for the modelling of industrial manufacturing – while modelling a restricted domain, the tool contains a large number of different models, mostly rendered as diagrams. The question of methodology or process then naturally follows: in which order

should one build the diagrams? More generally, which information should one model at a given point in time? The funding of research projects focusing on this question demonstrates the relevance of this question: for example SPES-XT [17], targets the development of methodologies for the domain of embedded systems. Similarly, Arcadia [10] emphasizes the importance of the methodology in connection with MDE.

Just like for tools however, methodologies and processes can seldom be general enough to match all use cases and answer all needs. There is thus also a need for tailoring at that level. To facilitate the acceptance of the methodology, it is essential that the tool *supports* the methodology: this is for instance the case with Capella and Arcadia. Capella however, supports only the Arcadia methodology which is specific to avionic systems engineering and to the processes of Thales. All the above points to the fact that, if the tools can be tailored, the process itself should be tailorable.

In this paper, we propose precisely an approach to develop DS(M)Ls in JetBrains' MPS, *equipped with a means to customize the tool in order to support a given process:* using out approach, in addition to the usual MPS mechanisms to develop their DSLs, developers can also explicitly express their own process. Such a process of model construction is declared as a statechart-like diagram, being that the current state is defined by the satisfaction of some properties of the model as well as by its previous states. For each state, the tool developer can define hints to be displayed as well as quickfixes in the form of creation of new artifacts.

In order to clarify our work we draw an analogy with the M-levels defined by the Object Management Group:

- M3: the MPS tool with its language definition capabilities.
- M2: development and composition of "brick DSLs" in a domain-specific model-driven development environment, together with a model construction process.
- M1: usage of the developed domain-specific tool.

We also identify four different *roles* in the development and usage of the framework:

- the *framework developer* (in our case JetBrains), who develop MPS (level M3),
- the *framework customizer* (the authors of this paper), who develop a library for process-customizable DSLs (level M2),
- the *(domain-specific) tool developer* (typically a consultant or the in-house technology department of a company) who actually develops the domain-specific tool, making use of our libraries (level M2),
- and the *user* (level M1).

In this work, we contribute a framework at level M2 to support the tool developer in developing a *process-aware* domain-specific tool. At the M1 level, a "dashboard" allows the user to know permanently what is the next step to achieve.

As a case study, we demonstrate how to use this framework for the specific domain of requirements engineering: the step-by-step formalization of requirements is a common approach, making it, therefore, an ideal candidate for the development of a process-aware tool. For this purpose, we have implemented a set of DSLs supporting the MIRA [18] framework, a general approach for the stepwise formalization of requirements focusing on quality assurance for requirements.

A *tool developer* can use this set of DSLs as a basis for their model-based development environment, by composing them with more specific DSLs of their own design.

She can also "drive" the user in her requirements formalization process by implementing a process using our Process language. We illustrate this approach by developing a requirements-engineering tool specialized in the development of hardware cooling systems, inspired from a case study from our industrial partners at Diehl Aerospace.

The remainder of this paper is structured as follows. In section 2 we describe the MPS framework, which we use as the technological basis for all the results presented in this paper. Section 3 then describes our case study – the construction of a model-driven development environment for the incremental gathering and refinement of requirements. Then, in section 4, we exemplify the construction of the requirements using the environment described in the previous section. Section 5 provides pointers to work in the literature that closely relates to the results we present here. Finally, section 6 presents a discussion of our research and potential future work.

2 The *metameta* Level: MPS: A Language Meta-Editor

The metameta level (M3, in MOF terms) is where the bricks for our approach are built. These bricks consist of DSLs, defined in the MPS (Meta Programming System) [3] framework. MPS is a stable and industrially-proven projectional meta-editor which provides edition capabilities at the meta-levels needed for our approach: M2 and M1. It uniformly integrates language and editor design capabilities, together with code generation tools and in-built correct-by-construction techniques such as metamodel conformance, syntax highlighting, auto-completion or type checking. MPS is developed by JetBrains, which assumes the role of *framework developer*. Throughout this paper we will often use vocabulary that is close to that used in the MPS world in order to remain aligned with the technical aspects of our work. In particular, the following terms are recurrently used in what follows:

- *Language*: an MPS language includes a metamodel, in the classical EMF sense. It additionally includes one or more editors for its metamodel, which provide concrete syntax.
- *Solution*: MPS solutions are projects where users can import MPS languages and create their models using those languages.
- *Concept*: the MPS equivalent of metamodel class.
- *Concept / language instance*: concepts can be instantiated, in the same way metamodel classes can. We will also sometimes write *language instance* to refer to an instance of the *root* concept of an MPS language.
- *Intentions*: actions attached to concepts of a language, available to the user.
- *Language composition*: Reference or containment relations can exist between instances of concepts of different languages, which is the primary language composition mechanism in MPS. Another composition mechanism is the MPS model, which can contain instances of concepts belonging to many languages.

3 The *meta* Level: Defining a Requirements Gathering Framework in MPS

3.1 A Generic Requirements Gathering Framework

The case study requirements gathering framework we introduce here borrows its structure from the Model-based Integrated Requirement Specification and Analysis (MIRA) Framework [18, 8]. The main parts of the MIRA framework are as follows:

– The *system context* describes the relevant elements that belong to the context of the system being developed.
– The *requirement list* documents the capabilities and limitations of the system under development.
– The *trace link list* keeps the relations between the artefacts being defined.
– The *QA* collection describes quality assurance activities and results.

This broad classification of the concepts necessary to build a requirements gathering framework is helpful to us in a operational manner. We wish to build our framework as a set of composed domain specific languages in the MPS environment. For that we need an architecture around which we can organize such languages. The MIRA architecture thus helps us in understanding how to group and compose those languages. A particularity of MIRA is that *quality assurance* is natively integrated in the framework. MIRA's quality assurance also has an operational counterpart in the MPS environment – in terms of language checks that come with the DSLs defined in MPS as well as other (formal) analyses that can be defined by the *domain-specific tool developer* building the framework. These analyses are orchestrated during the definition of the *process* when building a tailored model-driven development environment and correspond to MIRA's verification activities.

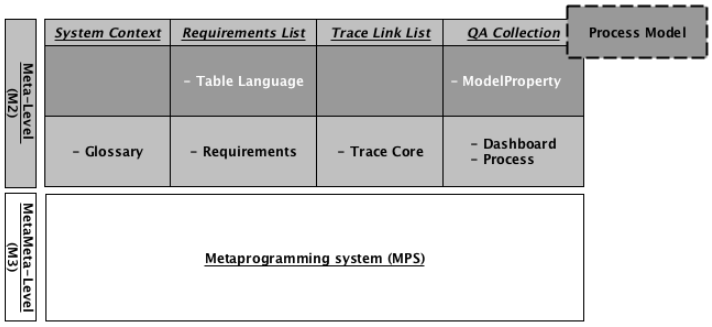

Fig. 1. A language stack for gathering software requirements

In figure 1 we present the first two meta-levels of our framework in the form of a stack of languages: at the bottom of the stack we have MPS itself, with its meta-edition capabilities; in the lower half of the layer immediately above we define the four aspects of the MIRA framework. Each one of the four aspects includes groups of DSLs that allow the requirements engineer to express parts of the complete requirements model. The *framework customizer*, in this case our group at *fortiss*, is responsible for building this part of the stack. The subset of MIRA we use for the work we present here is composed of the following languages:

– *System Context*: this aspect of the framework contains a *glossary language* to allow defining the domain specific glossary terms that are used across a requirements project, together with values associated to those terms.
– *Requirements List*: this aspect of the framework contains a Requirements language for expressing textual requirements, together with meta-information such as the author of the requirement or the requirement's current state.
– *Trace Links*: this group contains a generic *trace language* that can be extended to building trace links from any to any model element in MPS.

– *QA collection*: the QA collection group includes the Process and the Dashboard languages. The Process language allows defining refinement tasks for the requirements engineer. The Process language is used in conjunction with the Dashboard language in order to define at which moments a specialized requirements gathering framework will provide visual hints and press-button actions that guide the requirements engineer in the direction of achieving her task in a correct manner.

3.2 Customising the Framework: An Industrial Case Study

Nowadays, many embedded hardware systems (e.g. in laptops, cars or planes) are assembled together with cooling systems. The main purpose of those cooling systems is to maintain an appropriate temperature during the operation of those hardware electronics. In this section we will extend the generic requirements gathering framework that has been introduced in section 3.1 in order to specialize it for gathering requirements for software controllers for hardware cooling systems. In particular, we will add a process for specifying this type of requirements.

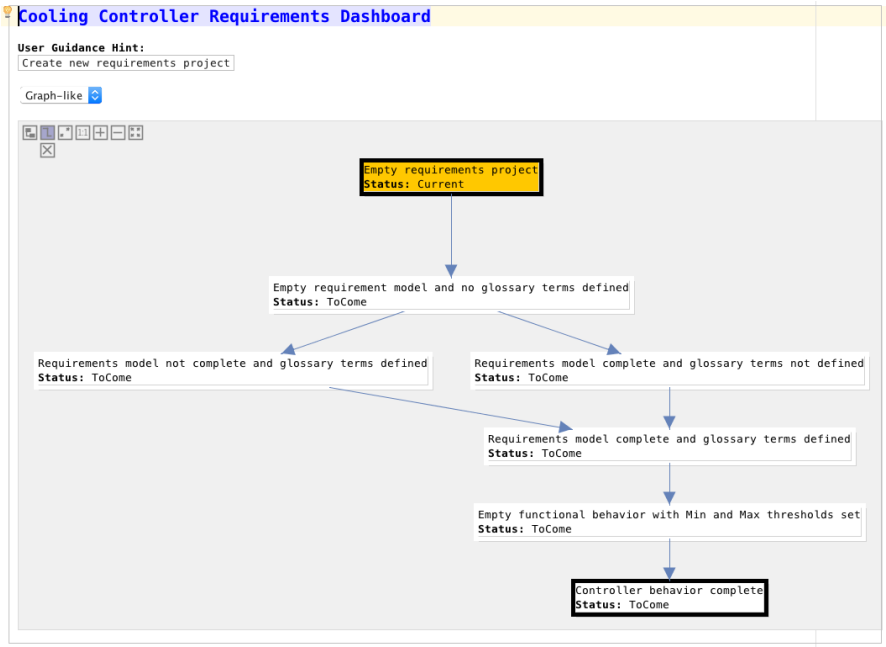

Fig. 2. A newly created dashboard for the requirements framework

The case study we present here is inspired by a requirements document (that for non-disclosure reasons we cannot cite here) which was made available to us by Diehl aerospace, one of our industrial partners. Diehl builds hardware for airplanes and, as such, cooling systems for that hardware also need to be produced. The process of gathering requirements the software that controls such cooling systems is currently almost completely manually done. This poses a problem to Diehl, as the current requirements gathering process imposes a large amount of effort to ensure that those requirements are documented in a fashion that is complete, correct and traceable. Additionally, the aerospace industry has to adhere to strict regulations such that their systems can be certified to be used in production. In order to specialize the two lower layers of

the language stack presented in figure 1 we have added the following languages: the Table language, for defining the behavior of the controller for the cooling system; the ModelProperty Language, which contains algorithms the check of properties that are specific to the cooling system requirements gathering system. Additionally, the *domain specific tool developer* also builds a process that will configure the dashboard's behavior. The dashboard assists the requirements engineer during the construction of the requirements by displaying hints and press-button actions that guide the refinement of the requirements model. The dashboard is configured by a description of which hints should be provided to the requirements engineer under which conditions. This information is provided as an instance of the Process language.

In figure 2 we provide an example of such a process, depicted directly in our tool's concrete statechart-like syntax, for the cooling controller requirements. The top part of each state in the figure includes the description of the conditions that need to be satisfied such that the hint associated to that state is displayed on the dashboard. The conditions described on the states of figure 2 are boolean properties that are checked on the current state of a requirements project. These properties are implemented in the ModelProperty Language in figure 1. This language holds the algorithms that implements the analyses required for our specific requirements gathering system. Note that, although this customization work has been done at fortiss, the idea is that in the future this work would be performed by a *domain specific tool developer* at Diehl. In a similar fashion, frameworks for gathering requirements for purposes other than for cooling systems could also be customized using as basis the same original set of languages as the one described in section 3.1.

4 The *Model* Level: A Domain-Specific Requirements Gathering Framework in Action

We will now exemplify how a user, in this case a *requirements engineer*, can make use of the specialized requirements framework we have defined in section 3.2. In particular, based on an existing requirements document provided by Diehl Aerospace and following the directions made available to us from engineers at Diehl, we will incrementally build the requirements for the software that controls a fan-based cooling system to cool down hardware boards embedded in the doors of passenger airplanes. Because of the fact that airplane doors include slides that should only be deployed when a plane lands under specific conditions, the logic running in those boards is non-trivial. The goal of the controller is to make sure the cooling fan runs at the correct duty cycle (fan speed) such that the the hardware board works under ideal temperature conditions. The calculation of the duty cycle for the fan at a given time depends on two inputs: 1) the current temperature of the hardware board that needs cooling and 2); whether the hardware board's temperature is increasing or decreasing.

Note that in the example that follows we do not pretend to be exhaustive in the construction of the requirements for the fan's controller. This section reflects part of our partners' requirements refinements process and demonstrates our framework's abilities in terms of providing automated assistance to the requirements engineer.

Tool Supported Requirements Definition and Refinement

The first thing to do in a requirements project for a cooling controller is to create a new dashboard by instantiating the Dashboard language that implements the

cooling controller requirements process. A newly generated dashboard for our fan requirements project is depicted in figure 2. The dashboard is the central artefact the requirements engineer refers to during requirements construction. It displays the current hint given to the *user*, as well as the current state of the requirements refinement process as a graph or a table. The hint displayed by the dasboard at the beginning of the project is "Create Project Structure". This hint is creational, which means the *user* can mouse-click on the hint to produce the instances of concepts that constitute the initial structure of the project in the same model where the dashboard instance has been created. Following this action a new hint, as shown in figure 3, proposes to the requirements engineer defining a new requirement for the system that includes the temperature thresholds for the functioning of the cooling system. Note that at this point the current state of the requirements model has now changed to "Empty requirements model and no glossary terms defined", as highlighted in orange in figure 3 – which now presents a tabular view of the refinement process.

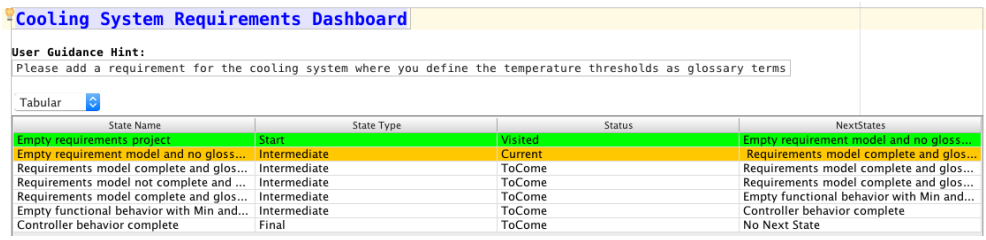

Fig. 3. The Dashboard provides a hint for adding a new requirement

The overall desired behavior of the cooling controller is stated as is an instance of the Requirements language in figure 4. From this requirementthe requirements engineer can then extract minimum and maximum threshold values using the MPS intention "Extract Into Glossary". When used, this MPS intention generates in the project's glossary an entry with the same names as words or sets of words selected from the text. This constitutes the first refinement of the original abstract requirement. The next step is to define the duty cycle of the fan as a function of the current temperature of the controller board and whether the temperature is going up or down. The dashboard assists the requirements engineer in this task by provinding a hint that, when mouse-clicked, automatically generates a table to define such behavior. An example of such a table is depicted in figure 5.

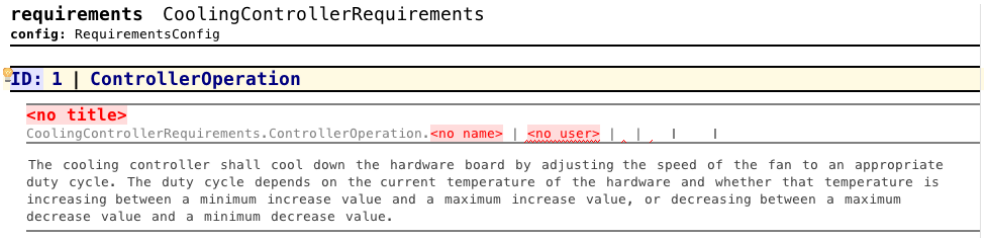

Fig. 4. The requirements engineer adds a new requirement

Note that the minimum and maximum threshold values in the table are preset in advance, as the process itself defines they should copied from the values previously stored in the glossary. We now come to the last refinement, which is to precisely

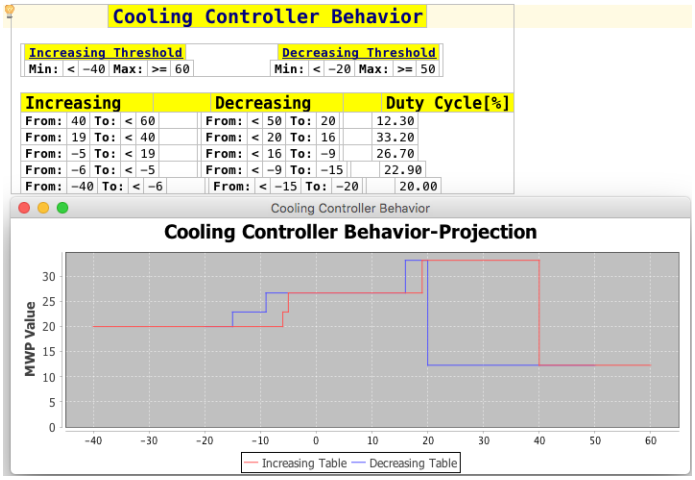

Fig. 5. A filled in table and associated 2D visualization of the defined function

define the behavior of the cooling fan's controller. In order to do this it is necessary to manually insert rows in the table that define the duty cycle for given intervals of temperature, when the temperature is going up or down. The Table language itself implements checks for completeness (all the temperatures between the thresholds are mapped onto duty cycles) and functionality (only one duty cycle value is given per temperature). Violations of these properties are pointed out in the editor as red markers. Figure 5 represents a filled in table holding, for non-disclosure reasons, a fictitious behavior of the fan's controller. In the figure a 2D-graph representation of the table is also visible and it is produced by applying the Visualize Graph MPS intention to the table.

The running example we have described in this paper can be downloaded at [1] as an MPS project. Another case study of using our framework is available at [2], where we have designed a process to assist the user in building natural langage-like requirements for embedded controllers. Two videos demonstrating our tool portraying these two case studies can be found at [5, 6].

5 Related Work

Mechanisms for providing some kind of guidance to the domain-specific language user are present, to a smaller or larger extent, in all DSL definition workbenches. In most cases, that guidance is provided in the form of correctness-by-construction (e.g. only correct models that conform to a metamodel can be built), or a-posteriori checks for conformance to certain well-formedness rules. This is the case for example for DSL workbenches such as Sirius [12], Xtext [11], AutoFocus3 [8], MetaEdit+[19] or MPS [16] itself. However, none of those tools is capable of natively providing the means to explicitely define a model construction process or methodology that can assist users when building instances of DSLs specified in those environments.

Model construction processes naturally depends on the domain the DSLs are aimed at. It is thus not surprising that the explicit notion of process is more present in modelling environments that are specifically aimed at certain domains – as previously mentioned, the Capella tool is a model-driven engineering solution for systems and software architecture engineering which enforces the Arcadia [10] methodology, aimed

at specific domains such as transportation, avionics, space or radar; the Soley tool suite [4], dedicated to model-based data extraction, processing and visualization, includes workflows as first-class citizens.

Coming back to generic workbenches for language constructions, there exists a large body of work in the area of model transformation chaining to orchestrate the flow of models to achieve a modelling goal. For example, authors such as Wagelaar [20] or Kolovos [13] propose mechanisms for automatically orchestrating model transformations such that certain modelling goals are achieved. In this area, the study which is the closest to the proposal in this paper is the FTG+PM framework [14, 15]. The FTG+PM defines an explicit process for the execution of model transformations. Enacting that process means that certain model transformations are performed automatically, while for others the user will have to input data at given points. The differences with the work we present in this paper have to do with the fact that our process is non-invasive, and is aimed at advising the user rather than executing a predefined work flow. With our approach automated actions are proposed to the user, who remains in complete control of the model edition process at all times.

6 Conclusions and Future Work

We have presented a technique for the construction of domain-specific model editors in MPS, where these editors are based on a set of composed DSLs and on a description of the process that should be followed when building the models for that domain. We have applied our approach to the construction of an editor for gathering software requirements at two levels: firstly, we build an abstract requirements gathering framework, following the guidelines in the MIRA framework [18]; secondly, we specialize that framework for our industrial partners at Diehl, by introducing a specific requirements refinement process for controllers for cooling systems.

The main technical contribution of this paper are the means to define a process to assist in building a model in a domain-specific model-driven development environment. This process is based on a set of automated model analyses and can guide the user until the model is complete. At a methodological level, our contribution regards our ideas on the separation of the *framework customizer* and the *domain specific tool developer* roles. While the former is responsible for defining a number of fundamental "brick" DSLs, the latter further specializes them for a particular application by adding more languages and organizing the whole according to a process.

In its current state, the main shortcoming of the technique we propose in this paper is the fact that it is the complete responsibility of the domain-specific tool developer to check the consistency of the properties being checked during the unfolding of the process, as well as their logical sequence in the process. Additional assistance during this step could be envisaged, for example in the form of automated checks for logical inconsistencies in the conditions that define each state in the process. Scalability is also an issue as analyses currently run very often in the background, which will rapidly lead to performance degradation in larger systems. Potential solutions to this problem are currently being investigated by colleagues of our at fortiss [7].

As future work, beyond mitigating the shortcomings above, we will continue working with Diehl Aerospace to develop the case study we present in this paper into a usable requirements gathering system. Other partner companies have shown interest in applying our work domains other than requirements engineering, which points to building or assembling different language stacks as well as different processes.

References

1. Cooling Controller Model Development Environment. https://github.com/levilucio/CoolingControllerReqsDash.git.
2. EARS-CTRL Model Development Environment. https://github.com/levilucio/EARS-CTRLDash.git.
3. Meta Programming System. https://www.jetbrains.com/mps/.
4. Soley Tool Suite. https://www.soley.io/.
5. Video for the Cooling Controller IDE. https://youtu.be/HcYH6AV1UEU.
6. Video for the EARS-CTRL IDE. https://youtu.be/vzhUwjH8eLE.
7. Vincent Aravantinos and Sudeep Kanav. Tool support for live formal verification. Submitted to the Practice and Innovation Track at MoDELS 2017.
8. Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In *Proceedings of ACES-MB (co-located with MoDELS)*, pages 19–26, 2015.
9. Andreas Bayha, Levi Lúcio, Vincent Aravantinos, Kenji Miyamoto, and Georgeta Igna. Factory product lines: Tackling the compatibility problem. In *Proc. of VaMoS 2016*.
10. Stéphane Bonnet, Jean-Luc Voirin, Daniel Exertier, and Véronique Normand. Not (strictly) relying on sysml for MBSE: language, tooling and development perspectives: The arcadia/capella rationale. In *Proceedings of SysCon 2016*, pages 18–21. IEEE, 2016.
11. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of SPLASH/OOPSLA 2010*, pages 307–309. ACM, 2010.
12. Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor N. Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of ASPLOS'15*, pages 223–238, 2015.
13. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. A framework for composing modular and interoperable model management tasks. In *MDTPI Workshop, EC-MDA*, 2008.
14. Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+PM: an integrated framework for investigating model transformation chains. In *Proceedings of SDL 2013*, pages 182–202, 2013.
15. Sadaf Mustafiz, Joachim Denil, Levi Lúcio, and Hans Vangheluwe. The FTG+PM framework for multi-paradigm modelling: an automotive case study. In *Proceedings of MPM@MoDELS 2012*, pages 13–18, 2012.
16. Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains MPS as a tool for extending java. In Martin Plümicke and Walter Binder, editors, *Proceedings of PPPJ '13*, pages 165–168. ACM, 2013.
17. Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger, editors. *Advanced Model-Based Engineering of Embedded Systems, Extensions of the SPES 2020 Methodology*. Springer, 2016.
18. S. Teufl, D. Mou, and D. Ratiu. Mira: A tooling-framework to experiment with model-based requirements engineering. In *Proceedings of RE'13*, pages 330–331, July 2013.
19. Juha-Pekka Tolvanen. MetaEdit+ for collaborative language engineering and language use (tool demo). In *Proceedings of SLE 2016*, pages 41–45, 2016.
20. D. Wagelaar. Blackbox composition of model transformations using domain-specific modelling languages. In *Proceedings of CMT 2006*, 2006.
21. Jon Whittle, John Edward Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *Proceedings of MODELS 2013*, pages 1–17, 2013.