Process-Aware Model-Driven Development Environments

Levi Lúcio, Saad Bin Abid, Salman Rahman, Vincent Aravantinos
fortiss GmbH
Email: {lucio,abid,aravantinos}@fortiss.org, salman.rahman@tum.de

Ralf Küster, Eduard Harwardt
Diehl Aerospace
Email: {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 such model-driven development environments can become process-aware, in order 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.**

I. 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 [25] 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 [18], Xtext [10], Sirius [12], or the classic MetaEdit+ [23]. With the maturity of these technologies, one can safely say that building your own MDE tool has never been so easy. This is an essential enabler and we can now clearly observe how technology enthusiasts in various industries put this opportunity to good use, developping their own domain- or even company-specific tools.

On the other hand, as [25] 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 overwhelmed by the amount of modelling techniques at their disposal. This is in fact the case for even very specialized tools like Sfit [7] 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 use the diagrams? More generally, which information should one model at a given point in time? The funding of several research projects precisely focusing on this question, in particular in connection with MDE, demonstrate the relevance of this question: CESAR [21], SPES [20], SPES-XT [19], all target the development of methodologies for the domain of embedded systems. Similarly, Arcadia [8] 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 henceforth also a need for tailoring at that level. To facilitate the acceptance of the methodology, it is however also 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 expressed declaratively 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 for that environment.
- M1: usage of the developed domain-specific tool.

In addition, we 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. In particular we have implemented a so-called Process language for describing a set of refinement steps including descriptions which are then used to guide and help the user. At the M1 level, a "dashboard" allows the user to know permanently 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 [22] framework, a general approach for the stepwise formalization of requirements focusing on quality assurance for requirements.

A *tool developer* willing to implement their own requirements-engineering tool 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. They can also "drive" the user in their requirements formalization process by implementing a dedicated 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 coming from our industrial partners at Diehl Aerospace.

The remainder of this paper is structured as follows. In section II we describe the MPS framework, which we use as the technological basis for all the results presented in this paper. Section III then describes our case study – the construction of a model-driven development environment for the incremental gathering and refinement of requirements. Then, in section IV, we exemplify the construction of the requirements using the environment described in the previous section. Section V lifts the veil over some of the implementation details of our work and section VI provides pointers to work in the literature that closely relates to the results we present here. Finally, section VII presents a discussion of our research and potential future work.

II. 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 Domain Specific Languages, defined in the MPS (Meta Programming System) [3] framework. MPS is a stable and industrially-proven projectional meta-editor. Being a meta-editor, MPS provides edition capabilities at the meta-levels we need for our approach, in particular M2 and M1. It uniformly integrates language and editor design capabilities, together with code generation tools and in-built correct-by-construction tactics such as meta-model 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. Other aspects of a language can be defined and custom new aspects can be built by the MPS user (see [3] for details).
- *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*: arbitrary actions attached to concepts of a language. Those actions can be launched by the user when the focus of the editor is on objects which are instances of those concepts.
- BaseLanguage Java: most of MPS' complex language operations are coded using the predefined BaseLanguage MPS language, a projectional replica of Java enriched with MPS-specific constructs.
- *Language composition*: Reference or containment relations can exist between instances of concepts of different languages, which is the primary language composition mechanism in MPS. Additionally, an MPS model can contain instances of concepts belonging to many languages (not necessarily referring to each other), which provides an additional means for language composition.

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

A. 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 [22]. 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 for 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 *domain-specific tool developer* building the framework. As we will explain further ahead in this paper, 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.

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 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. We have opted at this point for not including in the figure all the DSLs that would be required to have a full-blown implementation of the MIRA framework[1], but rather a subset of those languages that is coherent and sufficient to exemplify our approach in practice. That subset is as follows:

- *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 which refinement tasks 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. Note that the Process language allows structuring a *schedule*, or *script* for building a complex model which relies on incrementally satisfying formal properties of the model. These formal properties are defined by the user or exist natively in MPS in the form of e.g. metamodel constraints or type checks.

[1]The MIRA framework has been implemented as a requirement specification plugin for the Autofocus environment [6]

Note that although we have used MIRA as a reference for our project work with our industrial partners, additional explanations on the framework are beyond the scope of this paper. We refer the interested reader to [22].

B. Customising the Requirements Gathering 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 III-A 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.

The case study we present here is inspired by a requirements document (that for non-disclosure reasons we cannot cite here) that 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. Software reviewers require precise documentation for requirements, which is difficult to produce and maintain when little automation is available.

The work presented in this section constitutes our effort to provide a proof-of-concept model-based development environment to the Diehl engineers for the construction of requirements for hardware cooling systems. 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 III-A.

In order to specialize 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.

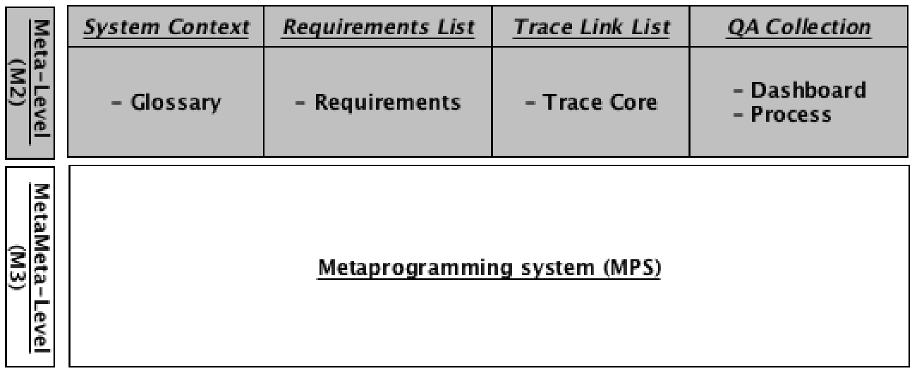


Fig. 1. A customizable language stack for gathering software requirements

- The ModelProperty Language, which contains algorithms implementing analyses 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 3 we provide an example of such a process, depicted as a statechart, for the cooling controller requirements. The top part of each state in the figure includes the conditions that need to be satisfied such that the hint in the lower part of that state is displayed on the dashboard.

The conditions described on the statecharts' states are boolean properties that are checked on the current state of the requirements project. These boolean properties are implemented in the ModelProperty Language in figure 2. This language is dedicated to holding the algorithms that run the analyses required for our specific requirements gathering system.

Hints can be of two types: *creational* or *informational*. Creational hints enable the user to automatically generate new instances of languages in order to complete the model. On the other hand, informational hints serve as guidance steps to the user but are not associated with automatic actions. This means the requirements engineer has to manually perform a set of actions in order to further refine the requirements model and reach the next state of refinement of the requirements model as defined in the hint model in figure 3.

For clarity reasons, let us now briefly explain the process model depicted in figure 3. The requirements engineer is initially provided with an empty dashboard with a creational hint to start constructing the requirements for the cooling controller. At this point an empty requirements project can be automatically built using the creational hint. An informational hint is then provided to guide the requirements engineer through defining the threshold values as glossary terms in the requirement and/or complete the requirements with all necessary data. Upon successfully building a requirement and defining the glossary terms, the requirements engineer is provided with a creational hint to perform the refinement step from a requirement and its threshold values into a cooling function describing the behavior of the controller. An empty cooling function (i.e., a model of the Table language) is then automatically created with the minimum and maximum thresholds values coming from the glossary terms. The user is then provided with feedback from the Table language until a correct cooling function is defined.

IV. 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 III-B. 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.

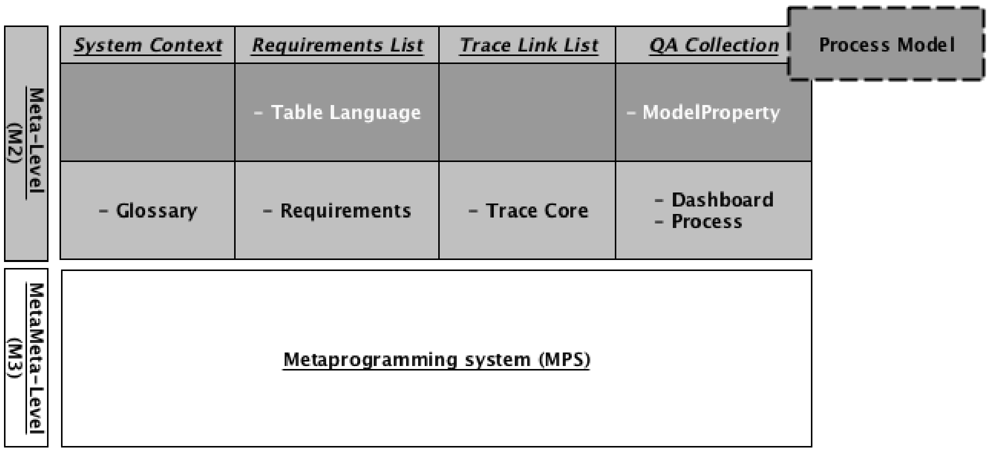


Fig. 2. A specialized language stack for gathering software requirements

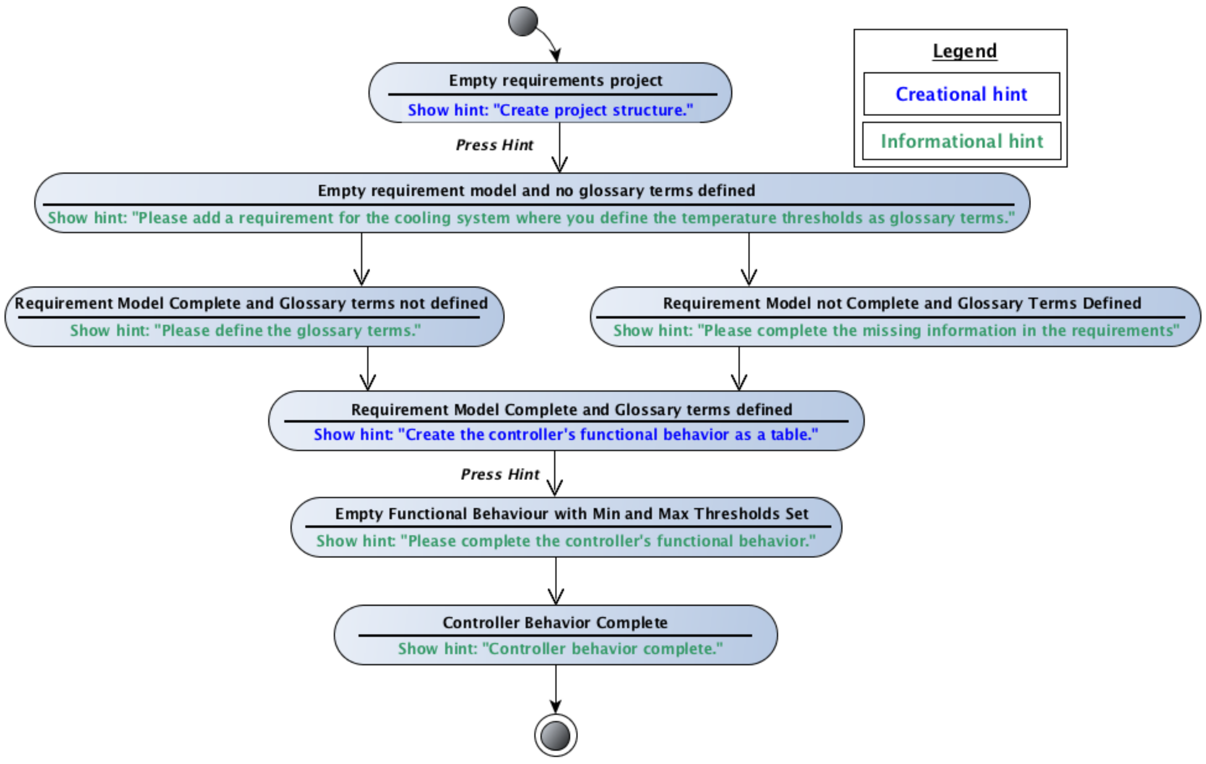
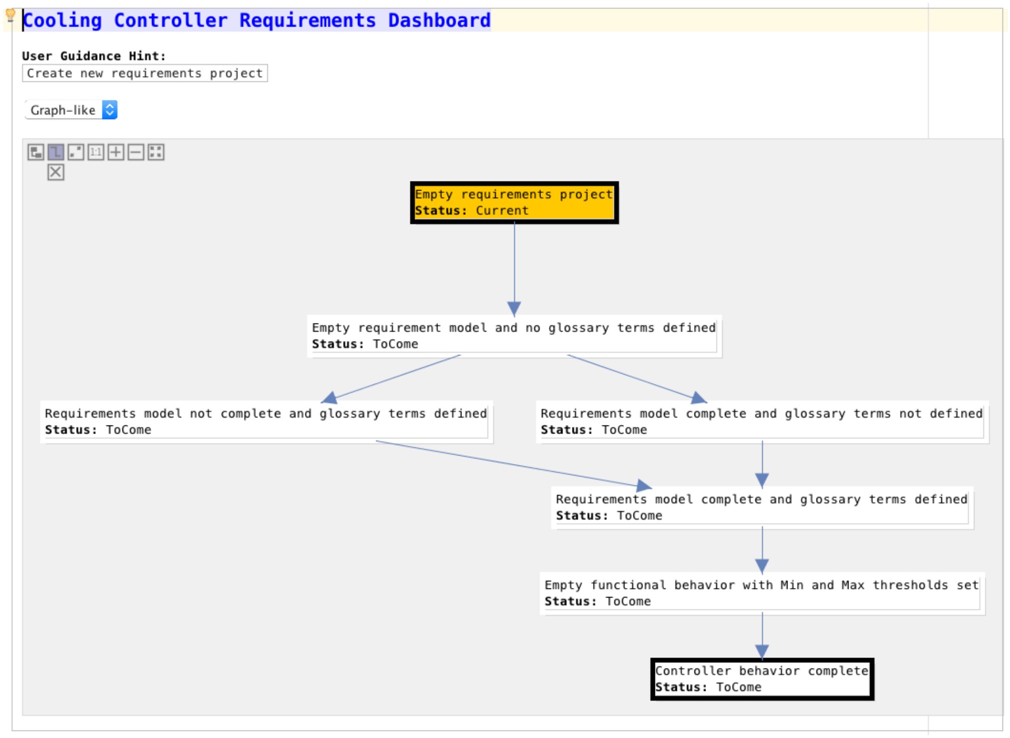

Legend
Creational hint
Informational hint
Empty requirements project
Show hint: "Create project structure."
Press Hint
Empty requirement model and no glossary terms defined
Show hint: "Please add a requirement for the cooling system where you define the temperature thresholds as glossary terms."
Requirement Model Complete and Glossary terms not defined
Show hint: "Please define the glossary terms."
Requirement Model not Complete and Glossary Terms Defined
Show hint: "Please complete the missing information in the requirements"
Requirement Model Complete and Glossary terms defined
Show hint "Create the controller's functional behavior as a table."
Press Hint
Empty Functional Behaviour with Min and Max Thresholds Set
Show hint "Please complete the controller's functional behavior."
Controller Behavior Complete
Show hint: "Controller behavior complete."

Fig. 3. A statechart-like representation of an instance of the Flow language

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 as part of the requirements model being built. This is achieved by instantiating the dashboard concept that implements the process depicted in figure 3. A newly generated dashboard for our fan requirements project is depicted in figure 4.

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 overall 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”. As shown in figure 3, this hint is creational. This 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. Among those instances are a placeholder for textual requirements and a glossary for


Cooling Controller Requirements Dashboard

User Guidance Hint:
Create new requirements project

Graph-like

Empty requirements project
Status: Current

Empty requirement model and no glossary terms defined
Status: ToCome

Requirements model not complete and glossary terms defined
Status: ToCome

Requirements model complete and glossary terms not defined
Status: ToCome

Requirements model complete and glossary terms defined
Status: ToCome

Empty functional behavior with Min and Max thresholds set
Status: ToCome

Controller behavior complete
Status: ToCome

Fig. 4. A newly created dashboard for the requirements framework

the project, as well as a couple of instances where users and project configurations are declared. The initial structure of the project as a set of instances of different languages is depicted in figure 5. The project itself is a model inside an MPS solution.

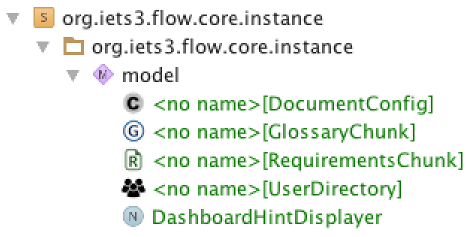

```
org.iets3.flow.core.instance
  org.iets3.flow.core.instance
    model
      <no name>[DocumentConfig]
      <no name>[GlossaryChunk]
      <no name>[RequirementsChunk]
      <no name>[UserDirectory]
      DashboardHintDisplayer
```

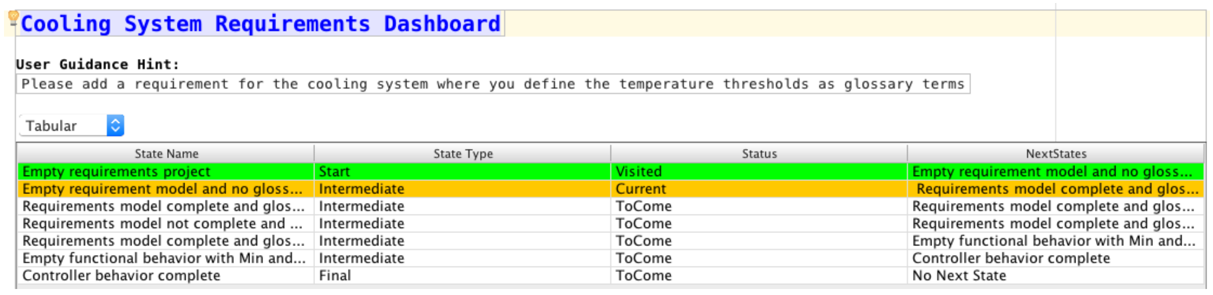
Fig. 5. A new cooling controller requirements project

After the structure of the project has been built a new hint, as shown in figure 6, 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 6 – which now presents a tabular view of the refinement process.

The overall desired operation of the cooling controller can be stated in the following requirement:

"*The cooling controller shall cool down the hardware board by adjusting the speed of the fan to an appropriate duty cycle. The duty cycle depends on the current temperature of the hardware and whether that temperature is increasing between a minimum increase value and a maximum increase value, or decreasing between a maximum decrease value and a minimum decrease value.*"

Figure 7 formalizes this high-level requirement in the Requirements language instance that was previously created when the project structure was automatically generated. From this text the requirements engineer can then extract minimum and maximum threshold terms for expected temperatures into the glossary model, that was itself also created with the project structure. Extracting these terms into the glossary is achieved using the MPS intention "Extract Into Glossary" associated to the root concepts of the Requirements language. When used, this MPS intention generates in the the project's glossary an entry with the same names as words or sets of words selected from the text. The values for the threshold terms need then be manually added to the glossary as can be seen in figure 8. This constitutes the first refinement of the original abstract requirement.


Cooling System Requirements Dashboard

User Guidance Hint:

Please add a requirement for the cooling system where you define the temperature thresholds as glossary terms

Tabular

State Name	State Type	Status	NextStates
Empty requirements project	Start	Visited	Empty requirement model and no gloss...
Empty requirement model and no gloss...	Intermediate	Current	Requirements model complete and glos...
Requirements model complete and glos...	Intermediate	ToCome	Requirements model complete and glos...
Requirements model not complete and ...	Intermediate	ToCome	Requirements model complete and glos...
Requirements model complete and glos...	Intermediate	ToCome	Empty functional behavior with Min and...
Empty functional behavior with Min and...	Intermediate	ToCome	Controller behavior complete
Controller behavior complete	Final	ToCome	No Next State

Fig. 6. The Dashboard provides a hint for adding a new requirement

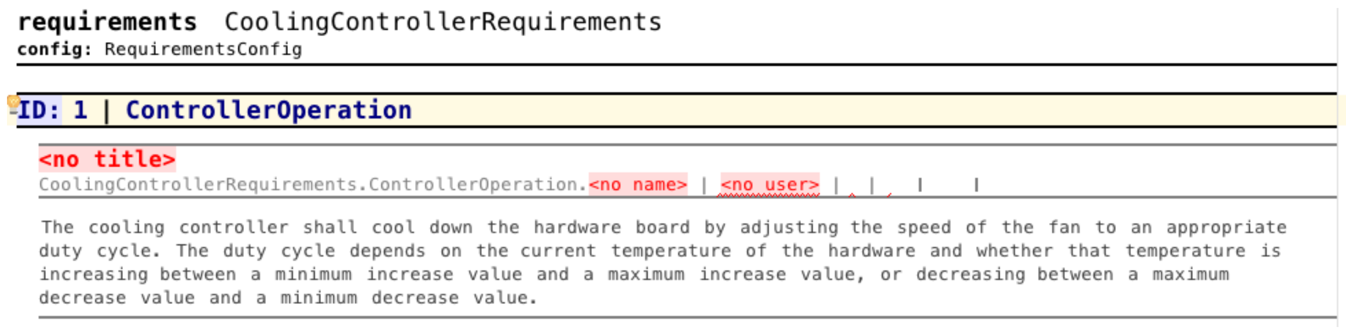
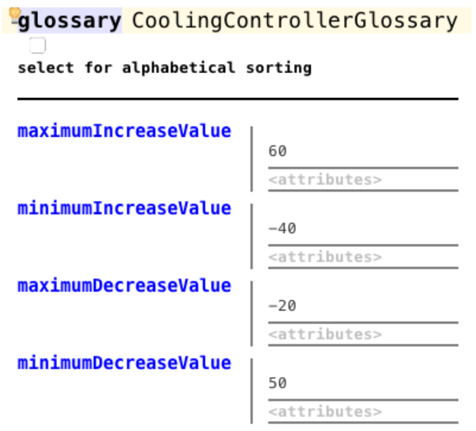

```
requirements CoolingControllerRequirements
config: RequirementsConfig

ID: 1 | ControllerOperation

<no title>
CoolingControllerRequirements.ControllerOperation.<no name> | <no user> | | | |

The cooling controller shall cool down the hardware board by adjusting the speed of the fan to an appropriate
duty cycle. The duty cycle depends on the current temperature of the hardware and whether that temperature is
increasing between a minimum increase value and a maximum increase value, or decreasing between a maximum
decrease value and a minimum decrease value.
```

Fig. 7. The requirements engineer adds a new requirement


glossary CoolingControllerGlossary

select for alphabetical sorting

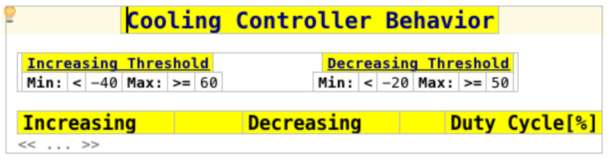
maximumIncreaseValue	60
minimumIncreaseValue	-40
maximumDecreaseValue	-20
minimumDecreaseValue	50

Fig. 8. The glossary for the cooling controller requirements

The next step is to actually 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. In figure 9 we depict the project's dashboard[2], which at this point proposes the creation of a table where such a function can be defined. Given the hint is creational, the requirements engineer can generate an empty table in the project model by pressing it. Such table is depicted in figure 10. Note that the minimum and maximum thresholds are already set in the table, as the process itself defines they should copied from the threshold values previously defined in the glossary.

[2]Note that in figure 9 we consider that all states that precede the current state have been visited. This is due to the assumption that forking processes achieve the same results by executing the same actions in alternative orders.

The requirements engineer can now proceed to the last refinement, which is to precisely 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.


Cooling Controller Behavior

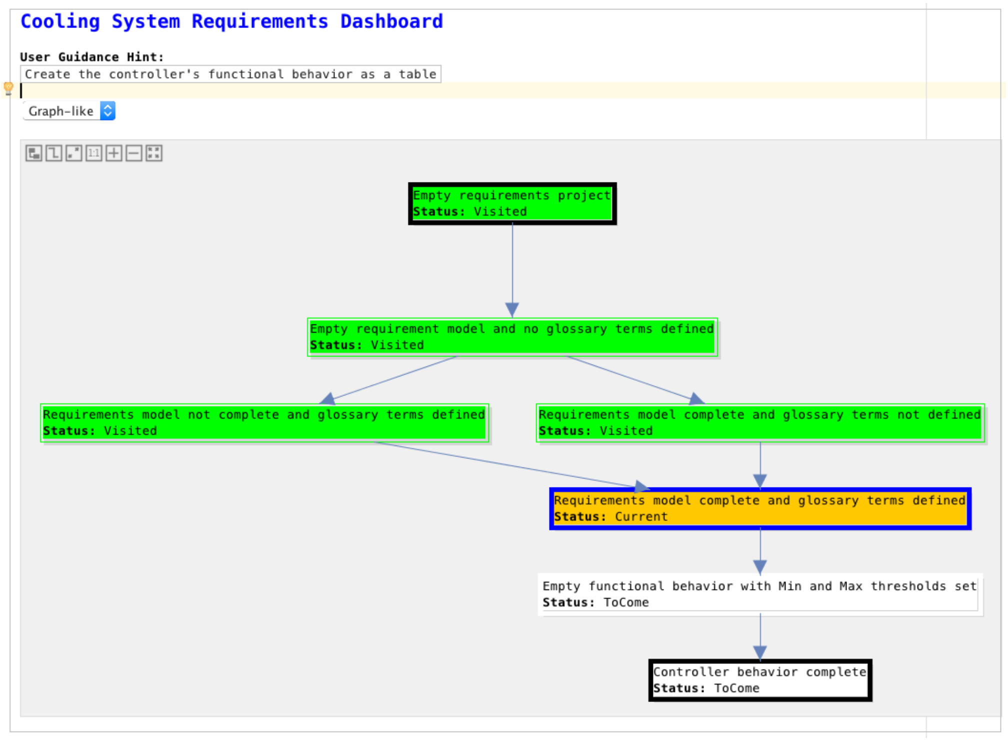
Increasing Threshold — Min: < -40 Max: >= 60

Decreasing Threshold — Min: < -20 Max: >= 50

Increasing	Decreasing	Duty Cycle[%]
<< ... >>		

Fig. 10. An empty table with filled in threshold values

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 11 represents a filled in table holding, for non-disclosure reasons, a fictitious complete behavior of the fan's controller. In the figure the 2D-graph representation of the table is also shown, and it can be produced by applying the Visualize Graph MPS intention to the table.



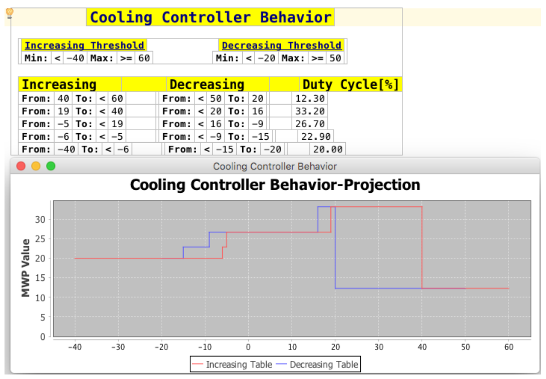
Cooling System Requirements Dashboard

User Guidance Hint:
Create the controller's functional behavior as a table

Graph-like

```
Empty requirements project
Status: Visited

Empty requirement model and no glossary terms defined
Status: Visited

Requirements model not complete and glossary terms defined
Status: Visited

Requirements model complete and glossary terms not defined
Status: Visited

Requirements model complete and glossary terms defined
Status: Current

Empty functional behavior with Min and Max thresholds set
Status: ToCome

Controller behavior complete
Status: ToCome
```

Fig. 9. The dashboard proposes creating a table to define the behavior of the fan controller



Cooling Controller Behavior

Increasing Threshold					
Min:	<	-40	Max:	>=	60

Decreasing Threshold					
Min:	<	-20	Max:	>=	50

Increasing					Decreasing						Duty Cycle[%]
From:	40	To:	<	60	From:	<	50	To:	20		12.30
From:	19	To:	<	40	From:	<	20	To:	16		33.20
From:	-5	To:	<	19	From:	<	16	To:	-9		26.70
From:	-6	To:	<	-5	From:	<	-9	To:	-15		22.90
From:	-40	To:	<	-6	From:	<	-15	To:	-20		20.00

Fig. 11. An filled in table and associated 2D visualization of the defined function

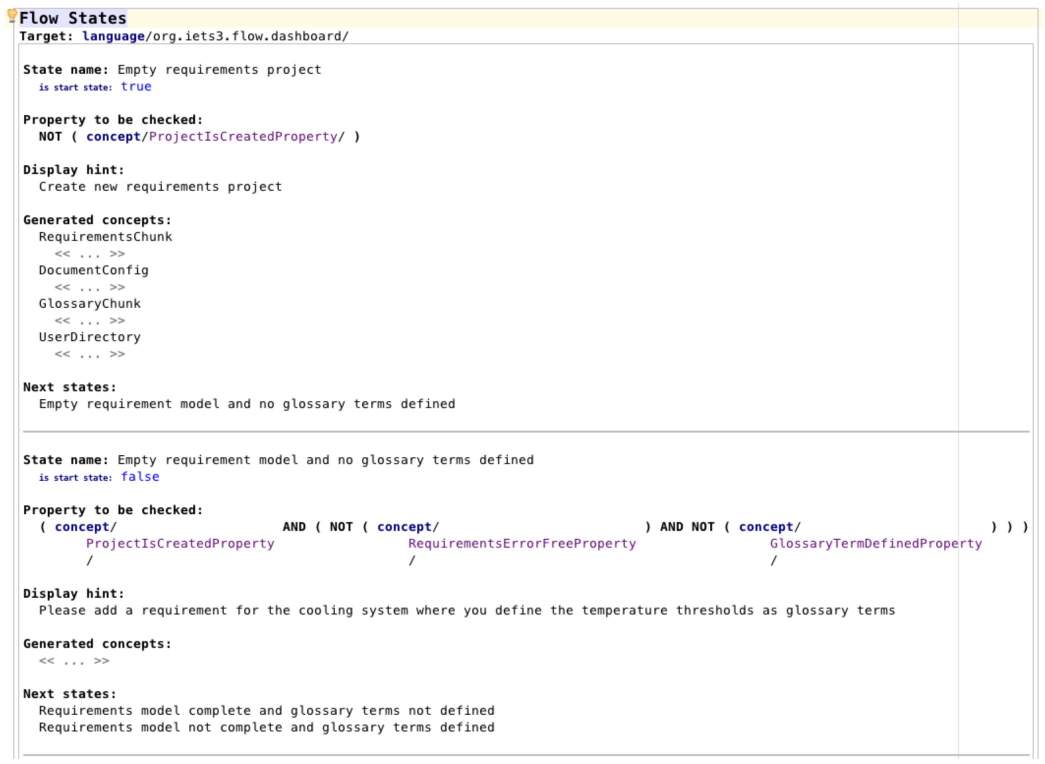
V. A Peek into the Implementation Details

As previously mentioned, the framework we present here relies on the one hand on DSL composition, and on the other hand on a process to guide the user through the construction of a model. The most important blocks for defining a process are *properties* of the composed model being defined. These properties extend an internal MPS Java BaseLanguage interface (defined at the metameta M3 level), called SpecificChecker. The SpecificChecker interface is implemented by all model checkers inside MPS. A few model checkers are already implemented inside MPS for analyzing models and returning error instances. Those model checkers search for example for typing errors or metamodel constraint violations. We extend the same SpecificChecker interface in order to build our custom checkers that can perform arbitrary checks on models, including interfacing with external analyzers. We then wrap such checkers using concepts of the Property language (see figure 2) to build the *properties* which are in turn used by the processes defined as instances of the Process language.

The *domain specific tool developer* is responsible for configuring the process that will be used by the dashboard to provide hints to the final user. This process is built as a instance of the Process language. An example of such an instance can be observed in figure 12, including the first two states of the process for gathering requirements for cooling controllers.

A process model contains states, where for each state the *domain specific tool developer* needs to define the type of state (*start*, *intermediate* or *final*), its name, the *condition* to be checked on the model, the text that should be displayed in the hint box on the dashboard, the new instances of concepts to be generated, if any, and the next states. The *condition* of a state is built as a propositional logic formula having as propositions concepts of the Property language, that are to be checked on the requirements model. Note that it is also possible to declare in a process state that data is copied from existing parts of the model onto the new instances that are created in case the hint is *creational*.

Once a process as been defined, the *domain specific tool developer* can then use an MPS intention to copy the process data in figure 12 to the Dashboard language declared as the *target* at the top of figure 12. This makes it such that the Dashboard language becomes aware of the process to be used for the model-driven development environment being built.



```
Flow States
Target: language/org.iets3.flow.dashboard/

State name: Empty requirements project
  is start state: true

Property to be checked:
  NOT ( concept/ProjectIsCreatedProperty/ )

Display hint:
  Create new requirements project

Generated concepts:
  RequirementsChunk
    << ... >>
  DocumentConfig
    << ... >>
  GlossaryChunk
    << ... >>
  UserDirectory
    << ... >>

Next states:
  Empty requirement model and no glossary terms defined

State name: Empty requirement model and no glossary terms defined
  is start state: false

Property to be checked:
  ( concept/                    AND ( NOT ( concept/                       ) AND NOT ( concept/                      ) ) )
      ProjectIsCreatedProperty              RequirementsErrorFreeProperty               GlossaryTermDefinedProperty
      /                                     /                                           /

Display hint:
  Please add a requirement for the cooling system where you define the temperature thresholds as glossary terms

Generated concepts:
  << ... >>

Next states:
  Requirements model complete and glossary terms not defined
  Requirements model not complete and glossary terms defined
```

Fig. 12. An sample of the instance of the Flow language for the cooling controller example

When a new dashboard is created in a project as an instance of the Dashboard language, an algorithm will continuously run in the background for calculating the current state of the process of creation of the composed model. This algorithm relies on the fact that the process graph has the following properties: it is a directed acyclic graph, having only one *start state* and one *end state*. Additionally, we also assume that forks imply different paths to achieve the same goal. This is enforced by mutually exclusive conditions on the states that follow a fork. As an example, in figure 3 it can be seen that the requirements model can be completed before the glossary terms are introduced, or the other way around. However, both must be finished before the "Requirements Model Complete and Glossary Defined" state. A corollary of this assumption is that only one state at a time can be the current state in the process graph flow.

The algorithm that calculates the current state operates by starting at the initial state and following the flow model graph until no more states can be found that satisfy the model. The last state that satisfies the model is returned as the current state. By skipping intermediate states that evaluate to *true* we avoid stopping the search at states of the process which will always be satisfied by the composed model after a given point, such as the "Requirement Model Complete and Glossary Terms Defined" state in our running example.

The main advantage of this algorithm we present here is that it allows for the current state to go backwards if parts of the model are deleted, as the process is always searched from the start. However, given our semantics of always returning for the latest state which property is satisfied by the model, care should be taken when designing the conditions on states. In particular, deletions in the model should not lead to inconsistencies between the current state as reported by the dashboard and the real current state.

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. This second case study is based on the work we have previously implemented on the EARS-CTRL tool [16] and it features properties in the process definition that are analysed by interfacing with the external autoCode4 [9] tool. Note that both [1], [2] are GitHub repositories that contain not only MPS projects, but also additional information about how to install those projects as well and pointers to videos demonstrating the usage of our framework.

VI. 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 [10], AutoFocus3 [6], MetaEdit+[23] or MPS [18] 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 [8] 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. Workflows are sequences of model transformations leading to analysis results that can be played (semi-)automatically, as well as recorded by the tool from a set of user actions.

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 [24], Guerra [11] 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 [15], [17], built on top of the AToM^3 [14] DSL workbench environment. 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 pre-defined 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.

Note although mechanisms like the one we propose in this paper do not exist in DSL workbenches such Sirius or AF3, it is possible to code them by using those workbenches' APIs and making usage of already existing analysis structures, as we have done for the work we present here. One of the differences that we are aware of regarding the AF3 workbench in particular, is that the guidance mechanism we present in this paper has been fully implemented using MPSs native mechanisms. Buiding the same mechanism in AF3 requires on the other hand extending the workbench's core framework using Java.

VII. 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 [22]; 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 framework. This process is based on a set of model analyses running in the background of the framework 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 potentially 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. Also, arbitrary deletions in the model may lead to inconsistencies in the current state of edition in the dashboard. A carefully designed process may avoid such inconsistencies, although automated help for the tool developer would also be important here. Finally, 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 [5].

As future work, beyond mitigating the shortcomings identified above, we will work continue working with Diehl Aerospace to further develop the case study we present in this paper into a usable requirements gathering system. Other partner companies have shown interest in applying the work we present to domains other than requirements engineering, which means in the future building or assembling different language stacks as well as different processes. Closer integration with MPS is necessary in order to perform analyses in the background only when strictly needed. Finally, mechanisms for the systematic creation of standalone editors that implement process-aware model-driven development environments is also in our future plans.

Acknowledgements

This work presented in this paper was developed in the context of the "IETS3" research project, funded by the German Federal Ministry of Education and Research under code 01IS15037A/B.

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] Vincent Aravantinos and Sudeep Kanav. Tool support for live formal verification. Submitted to the Practice and Innovation Track at MoDELS 2017.

[6] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In *ACES-MB (co-located with MoDELS)*, pages 19–26, 2015.

[7] Andreas Bayha, Levi Lúcio, Vincent Aravantinos, Kenji Miyamoto, and Georgeta Igna. Factory product lines: Tackling the compatibility problem. In Ina Schaefer, Vander Alves, and Eduardo Santana de Almeida, editors, *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016*, pages 57–64. ACM, 2016.

[8] 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 *Annual IEEE Systems Conference, SysCon 2016, Orlando, FL, USA, April 18-21, 2016*, pages 1–6. IEEE, 2016.

[9] Chih-Hong Cheng, Edward Lee, and Harald Ruess. autoCode4: Structural Reactive Synthesis. In TACAS'17, accepted for publication. Tool available at: `http://autocode4.sourceforge.net`.

[10] 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, *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 307–309. ACM, 2010.

[11] Esther Guerra, Juan Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar M. Santos. transML: A family of languages to model model transformations. In *MODELS*, volume 6394 of *LNCS*, pages 106–120. Springer, 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 Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 223–238. ACM, 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] Juan de Lara and Hans Vangheluwe. *AToM3: A Tool for Multi-formalism and Meta-modelling*, pages 174–188. Springer Berlin Heidelberg, 2002.

[15] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+PM: an integrated framework for investigating model transformation chains. In *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, pages 182–202, 2013.

[16] Levi Lúcio, Salman Rahman, Chih-Hong Cheng, and Alistair Mavin. Just formal enough? automated analysis of ears requirements. To appear in the proceedings of Nasa Formal Methods (NFM) 2017, 2017.

[17] Sadaf Mustafiz, Joachim Denil, Levi Lucio, and Hans Vangheluwe. The FTG+PM framework for multi-paradigm modelling: an automotive case study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012, Innsbruck, Austria, October 1-5, 2012*, pages 13–18, 2012.

[18] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains MPS as a tool for extending java. In Martin Plümicke and Walter Binder, editors, *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, pages 165–168. ACM, 2013.

[19] 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.

[20] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy, editors. *Model-Based Engineering of Embedded Systems, The SPES 2020 Methodology*. Springer, 2012.

[21] Ajitha Rajan and Thomas Wahl, editors. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer, 2013.

[22] S. Teufl, D. Mou, and D. Ratiu. Mira: A tooling-framework to experiment with model-based requirements engineering. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 330–331, July 2013.

[23] Juha-Pekka Tolvanen. Metaedit+ for collaborative language engineering and language use (tool demo). In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 41–45. ACM, 2016.

[24] D. Wagelaar. Blackbox composition of model transformations using domain-specific modelling languages. In *First European Workshop on Composition of Model Transformations-CMT 2006*, page 15, 2006.

[25] 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 Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.