On Integrating Graphical and Textual Editors for a UML Profile Based Domain Specific Language

An Industrial Experience

Salome Maro
Jan-Philipp Steghöfer
Anthony Anjorin
Chalmers | University of Gothenburg, Sweden
salome.maro@cse.gu.se
jan-philipp.steghofer@cse.gu.se
anjorin@chalmers.se

Matthias Tichy
University of Ulm, Germany
matthias.tichy@uni-ulm.de

Lars Gelin
Ericsson, Kista, Sweden
lars.gelin@ericsson.com

Abstract

Domain Specific Languages (DSLs) are an established means of reducing the gap between problem and solution domains. DSLs increase productivity and improve quality as they can be tailored to exactly fit the needs of the problem to be solved. A DSL can have multiple notations including textual and graphical notations. In some cases, one of these notations for a DSL is enough but there are many cases where a single notation does not suffice and there is a demand to support multiple notations for the same DSL. UML profile is one of several approaches used to define a DSL, however most UML tools only come with graphical editors. In this paper, we present our approach and industrial experience on integrating textual and graphical editors for a UML profile-based DSL. This work was conducted as part of an explorative study at Ericsson. The main aim of the study was to investigate how to introduce a textual editor to an already existing UML profile-based DSL in an Eclipse environment. We report on the challenges of integrating textual and graphical editors for UML profile-based DSLs in practice, our chosen approach, specific constraints and requirements of the study.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments - Integrated Environments

SLE'15, October 26–27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3686-4/15/10...$15.00
http://dx.doi.org/10.1145/2814251.2814253

Keywords UML profile, DSL, Graphical editor, Textual editor

1. Introduction

The term Domain Specific Language (DSL) refers to a language that is created for specific tasks [14]. A DSL captures design patterns that are common in a particular domain. Compared to General Purpose Languages (GPLs), a DSL focuses on the patterns that are used in a specific domain therefore avoiding all the general notations that are not needed within the domain. This makes creation of applications easier and designers can write less code (as little as 2%) than when using a GPL [22]. Applications developed using DSLs tend to be more concise, easier to maintain and reason about and above all can be developed quickly [16].

At Ericsson, DSLs are used to raise the abstraction level with which engineers create applications. This gives the engineers an opportunity to focus on the problem at hand and not worry about implementation details. Moreover, the fact that software at Ericsson is usually deployed on various hardware platforms is another main reason for the adoption of DSLs. It would be inefficient for engineers to write different code for every hardware platform. With DSLs they can reuse the logic models which are platform independent and this proves to be more efficient.

A DSL can have a textual notation or a graphical notation. When a DSL is large, covers a wide aspect and has different types of users like Ericsson's DSL, having one notation often does not suit the needs of all its users. This is because some cases are easier to specify when using a graphical notation while others can be conveniently specified using a textual notation [15]. On the one hand, a graphical notation for modelling applications has advantages like reducing the chances of errors, providing visualization and hence easing

understanding of the system being created. On the other hand, using text based modelling has advantages such as speed of creation and editing, speed of formatting and a wide availability of editors [15].

Unfortunately, when opting to use UML profile-based DSLs, only graphical editors are provided by most out-of-the-box UML tools. At the moment, Ericsson has a UML profile-based DSL for modelling applications for baseband switches. The company is using the Rational Software Architect (RSA) tool from IBM which only provides a graphical editor for UML models. Based on the cases that they model, some engineers have requested a textual editor to be provided because it would simplify their modelling tasks. At the same time, these engineers have to work and share models with other engineers who are using the graphical editor. Therefore, the company is investigating the possibility of adding a textual editor for the existing DSL.

Having a DSL with both textual and graphical editors being used simultaneously raises several problems. One of the main problems is how to maintain the two editors (textual and graphical) without adding substantial maintenance effort to the company. This means that when the DSL needs to be updated one should not have to do a lot of manual work. Another problem that arises is how to avoid information loss when users switch between graphical and textual views.

The main contribution of this paper is to present a practical approach on how to semi-automatically obtain a textual editor for a UML profile-based DSL and how to enable users to switch from one view of a model to another, i.e., to switch from the graphical view to the textual view and vice-versa without losing any information. Since the study was carried out in industry, we present the challenges that arise when having both textual and graphical editors for the same DSL.

The rest of the paper is organized as follows. Section 2 gives an overview of the DSL at Ericsson and Section 3 outlines the challenges of having textual and graphical editors for the same DSL. Section 4 describes our approach while Section 5 gives an evaluation of our tooling. Section 6 is a discussion on the approach in relation to the challenges. The paper concludes with Section 7 that discusses related work and Section 8, which gives a conclusion and proposes future work.

2. Industrial Case

Ericsson uses an in-house developed DSL to create applications for its baseband switches. This DSL is built using UML and UML profiles. Their UML profile is known as Hive profile and is divided into three major parts: Behaviour, Structure and Deployment. These serve different purposes when it comes to creating applications. The Hive DSL in total is made up of 17 stereotypes, 1 Enumeration and 1 class from the Hive Behaviour profile, 2 stereotypes from the Hive Structure profile and 14 stereotypes from the Hive Deployment profile.

At Ericsson, developers create models of applications using this DSL and later use model transformation tools to transform the models into C code (.c and .h files), which can be compiled into working applications. The transformations are done in three steps:

1. A Hive instance model is subjected to a model to model transformation that transforms it to a Dive model. Dive is another Ericsson in-house DSL which is XML-based. The Dive model is never manually edited by the user but always has to come from a corresponding Hive model using this model to model transformation.
2. From the Dive model, another model to model transformation is done that transforms the Dive model into a C instance model that conforms to the C Abstract Syntax Tree (AST).
3. From the C AST, a model to text transformation is performed to get .c and .h text files which can be compiled into working applications.

Figure 1 illustrates the above three steps.

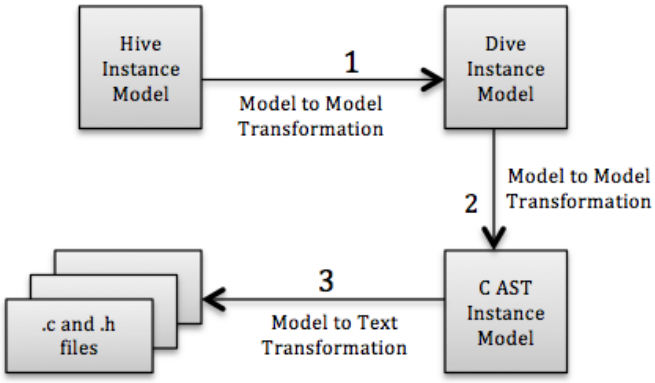

Figure 1: Transformation Tool Chain

The company uses RSA from IBM as their main editor which only provides a graphical editor for creating and editing UML models. As this does not suit the needs of all users, the company wants to add a textual editor so that their developers can have the option of modelling using text or graphics. For this the company has the following requirements:

1. Add a textual editor for the already existing DSL.
2. The company should not have to maintain the graphical and textual editors separately.
3. The engineers should be able to switch between textual and graphical views without losing any information.

One of the cases that we investigated at the company was how to model the behaviour part of the Hive profile using text. To model behaviour, the company uses activity diagrams extended with the Hive Behaviour profile. An activity diagram in UML is a behaviour diagram which shows flow of

control or object flow with emphasis on the sequence and conditions of the flow [31]. Figure 2 shows an extract of the Hive Behaviour profile with two stereotypes: `HiveAction` and `HiveVectorAction`. Both the stereotypes have two tagged values which are `taskPriority` and `operation`. Both stereotypes extend the `CallOperationAction` metaclass from the UML metamodel.

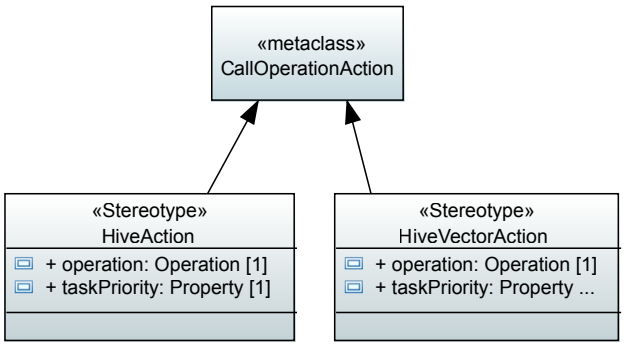

Figure 2: Extract from the Hive Behaviour Profile

3. Challenges

This section describes the challenges that we identified during the study. These are mainly general challenges related to having a textual and graphical editor used in a working environment and some specific challenges related to integrating a UML profile-based DSL with Xtext.

3.1 Storage and Versioning of Models in Repositories

In many companies it is common that more than one designer works on the same model and this is facilitated by using collaboration and versioning tools such as Git [12]. When some designers can edit the same model in text and some edit it graphically, the choice of which format (textual or graphical) to store in the repository is not a trivial one. Storing the models as graphical models means putting in place tools that can help designers to version, merge and resolve conflicts on a graphical model level. On the other hand storing them in text means using tools for versioning, merging and resolving conflicts that are on a textual level. Either way, deciding to store one version means that designers working in the other version need to first convert their models before committing their changes. This leads to other challenges such as synchronization and issues of maintaining the layout, which are discussed next. Storing both versions of the model is also an option that can be considered, but this means that one has to make sure that both models (textual and graphical) can be versioned and synchronised.

3.2 Synchronization of Models

When a user is editing a model in graphical format and decides to switch to textual notation, both the textual and graphical models should be in sync. This means that the tool should be aware of which text model represents which graphical model. This is a challenge if the graphical and textual views are not linked to each other as the source and target models can evolve separately and become out of sync. There is therefore a need to be able to link a transformed model to its source model in order to keep them synchronized.

3.3 Graphical Layout of the Model and Pretty Printing

When a user creates models in graphical format, they can arrange/format the diagram in a way that is suitable for them. For instance they can make the icons bigger, move the model elements to certain positions or arrange them in a particular way. When these models are transformed to textual models and then back to graphical models the user expects the layout of the original graphical model to be maintained. This is a challenge as the textual model does not store any information about the graphical layout of the model elements unless additional measures to facilitate this are put in place. As models get bigger, the layout that a user has modified becomes important and the auto-layout provided by the tool does not suit the user. Moreover, when a designer styles his or her text in a certain format using the text editor and then transforms the file to a graphical model and then back to text, the custom text formatting is also lost.

3.4 Model References

It is common that developers create several models that are linked to each other instead of creating one big model. This means that most models have model references to other models. This is a problem, as when switching from one view to another, one has to decide on whether to switch only the current model that is being worked on or also switch all models linked to the current model. Switching only the model being worked on means that we end up with a model containing a mixed grammar, i.e., a graphical model with links to a text model or vice versa. Switching the current worked on model, and all its referenced models may result to a bunch of models being transformed, which is not efficient when having many linked models.

3.5 Minimal DSL

The UML metamodel is a very huge one. For instance UML 2.0 has 265 model elements and 763 relationships [2]. This makes it quite complex to work with and in many cases, companies only use a small part of this metamodel. In order to come up with a small concise metamodel representing the UML profile, it is necessary to identify which parts of the metamodel the profile uses and which other classes the company uses from the UML metamodel. This is a difficult task because in many cases it is not clear which classes are actually used by the company and also because the UML metamodel contains many elements that are connected to each other and sometimes these connections are implicit.

3.6 Names in Model Elements

This challenge is mainly related to UML. The EMF implementation of UML uses special IDs called XMI IDs to identify model elements. This means that when a user creates an element in UML this element is assigned an ID by default. Even when a user does not give the element a name, UML can still identify that element with the ID it has. This has led to a habit that many designers do not give names to UML elements unless the names are necessary to them. For instance when modelling activity diagrams, most of the designers give names to the actions but not to the fork nodes or join nodes. However, when it comes to EMF textual editors, all elements are identified by using qualified names. If a designer creates a model element in text, the editor does not assign any ID to the element, it is the designer that needs to give unique names to the elements. This is a challenge as when transforming UML models to textual models, the produced text model elements are also unnamed. In text models we cannot have a reference to an element with no name so the resulting model breaks.

The main challenge here is that introducing the textual editor means that either the designers have to start giving names to all model elements in their models or the names should be automatically generated for elements that have no names when transforming to text models. Generating names also implies that when a user switches from text to graphics and back, the graphical model will have the generated names added. The XMI IDs from the UML models could also be considered for use as unique model element names in text but these have a format that is not user friendly.

3.7 Inconsistent Models

An inconsistent model is a model that does not adhere to the constraints of the metamodel, it has errors. When switching from a graphical model to a text model, if the transformation produces an inconsistent model then this model cannot be serialized in the Xtext syntax. The designer therefore needs to make sure that when transforming a graphical model, this model should not lead to an inconsistent text model. For example in the Hive case, the join node in textual metamodel has a constraint that it can have several inputs and only one output. So if a designer tries to transform a join node that is not connected to any input or any output then the result of this transformation cannot be serialized in Xtext textual syntax. The error must be fixed first for the serialization to work.

4. Approach

This section describes the approach that we used to address the problem of integrating textual and graphical editors in an Eclipse environment. It should be noted that the approach described here has been taken due to the fact that the company was already using a UML profile-based DSL. The approach also aims to address some of the challenges discussed in the previous section. This section is divided into two parts, the first part describes how to obtain the textual editor and the second part describes how to switch between graphical and textual views.

4.1 Obtaining the Text Editor

There are several EMF based plugins that can be used to generate textual editors with little effort. Since the company is already using RSA, which is also EMF based, going for one of these text editor generator plugins is a good solution. However, all these plugins need an Ecore model behind them in order to generate the textual editor. This Ecore model can be manually written but this means that every time the profile evolves, the Ecore model would need to be changed manually. To avoid this, we transformed the UML profile-based DSL into an Ecore model. The idea being that once the profile evolves, the Ecore model will be derived automatically.

The Ecore model obtained from the transformation could now be used with one of the text editor generator plugins to generate a textual grammar and editor for the DSL. For our case we used Xtext [6] to generate the grammar and the textual editor.

EMF comes with a functionality that can export UML models into Ecore models. Using this functionality was the first approach to transform the Hive profile to an Ecore metamodel for a DSL. This functionality worked well but it had one huge drawback since it also exports the whole UML metamodel to the exported Ecore model. This gives a very huge DSL with a lot of entities that are not needed. To overcome this we wrote our own ATL transformation and used a UML subset in our transformation instead of the whole UML metamodel.

To be able to obtain the UML subset, we need to know exactly which metaclasses are used. These metaclasses include those extended by the stereotypes in the profile and also those that are used without any stereotypes. For some DSLs this set of metaclasses is known and for some DSLs it may not be so obvious which metaclasses are used. This is especially true when users use part of UML that is not extended by any stereotype in the profile. In such cases this list of classes that are needed can be obtained by running a transformation that takes an instance model of the DSL and returns a collection of all UML metaclasses used on that instance model. This will give a correct list of classes needed only if the instance models cover 100% of the DSL.

In case no such instance models exist, one can identify the needed UML metaclasses manually and create a list of these classes either as an Ecore or as another UML profile that will only be used to identify these metaclasses. Once these classes have been identified a transformation can be written that copies only these classes from UML to create a subset UML metamodel. This UML subset can also be created manually as an Ecore model that contains all the classes of the subset and their attributes. However if the DSL changes frequently then this subset can be hard to maintain. For the case of Ericsson, we used a manual approach to get a

Table 1: UML to Ecore Mappings

UML	Ecore
Profile	EPackage
Stereotype	EClass
Metaclass	EClass
Property(Primitive Type)	EAttribute
Property	EReference
DataType	EClass
Enumeration	EEnum

list of classes used from the UML metamodel. This was done by examining various existing models and identifying classes that were implicitly used from the UML metamodel. This worked as a solution because their profile does not evolve frequently.

Since we created the UML subset manually, we had the flexibility to get rid of model elements that we considered unecessary in our final textual language or add some model elements. However this change needs to be noted so that when transforming back to UML models, we know how to re-create UML model elements from the changes made. The mapping for this are stored in a trace model [5] because the relationship of UML and the Ecore model will no longer be a direct one to one relationship. A trace model is a model that defines the relationship between a source model and a target model. In our case since one of the diagrams we modelled in text is activity diagrams, instead of having controlflows with source nodes and target nodes elements we added an attribute called "dependsOn" to activity nodes. This was done to make modelling of the flow from one node to another easier in text.

We wrote an ATL transformation that takes the UML profile and UML subset as input and produces an Ecore model as output. We used the produced Ecore model from the transformation as input to the Xtext plugin to generate the grammar and textual editor. This transformation could also be written using any transformation language. The mappings used to convert UML to Ecore follow the ones used in the UML to Ecore eclipse plugin [8] and also according to the relationship between UML and Ecore as described in [25]. The mappings are summarized in Table 1.

Figure 3 shows an extract of the Hive profile, part of the subset needed and the resulting Ecore model that was obtained from the transformation. In the figure, a stereotype called `HiveAction` is transformed to an EClass called `HiveAction`. The property of the `HiveAction` stereotype called `operation` which has a type of the UML Operation metaclass is transformed to an EReference called `operation` whose type is also an EClass called `Operation`. This `Operation` EClass comes from the `Operation` metaclass in the UML subset. The UML extension relationship that is represented by the property named `base_CallOperationAction` of type `CallOperationAction`, is also transformed to an EReference named `base_CallOperationAction` of type `CallOperationAction`. Similarly, the EClass called `CallOperationAction`, comes from the UML subset. This is done for all the stereotypes and their properties. From the UML subset, all the classes and their attributes are copied to the Hive Ecore model. So a class named Activity in the UML subset is transformed to a class named Activity in the Hive Ecore model.

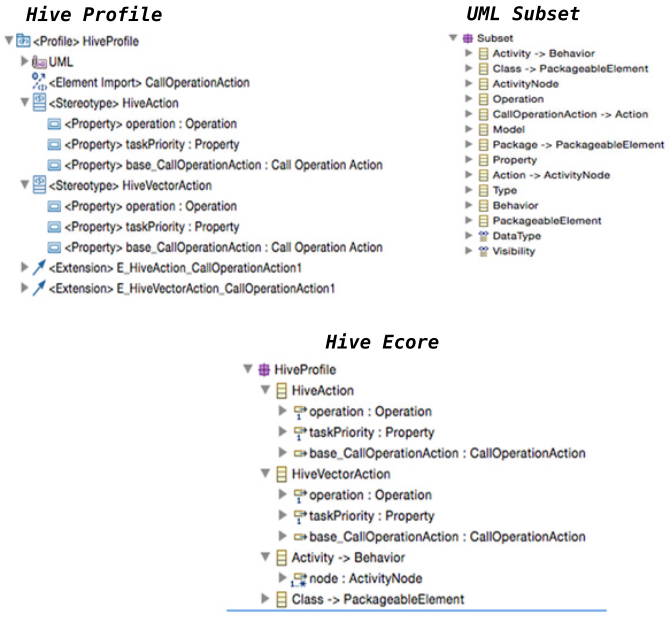

Figure 3: Hive Profile to Hive Ecore Metamodel

The grammar generated by Xtext from the Ecore metamodel of the DSL was not very usable because it had a lot of unnecessary syntactical structures and all the enumerations literals were missing. This is an Xtext specific issue for grammars that are automaticaly generated. Therefore, we manually edited the grammar in order to come up with something that is actually readable and usable.

Figure 4 shows our approach to obtain the textual editor from the Hive profile. It also shows which parts have to be done manually every time the Hive profile evolves and which parts have to be done only once.

With the setting shown in Figure 4, when the profile evolves, the following needs to be done to update the editor. First, one needs to check if the profile is using a metaclass that is not included in the UML subset and add all missing metaclasses and their attributes. Then the new profile and the UML subset will be used as input to the already existing ATL transformation to produce the new Hive metamodel. To update the grammar there are two choices, the first choice is to re-generate the whole grammar and the editor, but this means also re-doing the entire manual editing that was done to the grammar before. The second option is to replace the old Hive metamodel with the new one, this way when the editor

is compiled, the changes will be detected and one can update the grammar to match the new Hive metamodel manually.

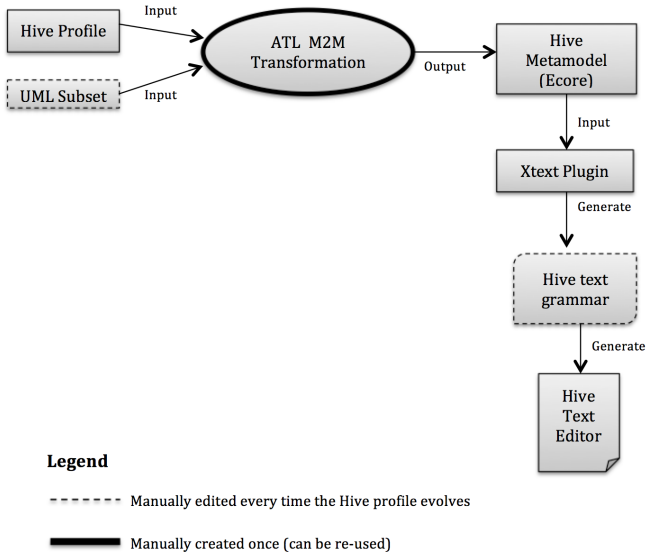

Figure 4: Generating the textual editor from the Hive profile

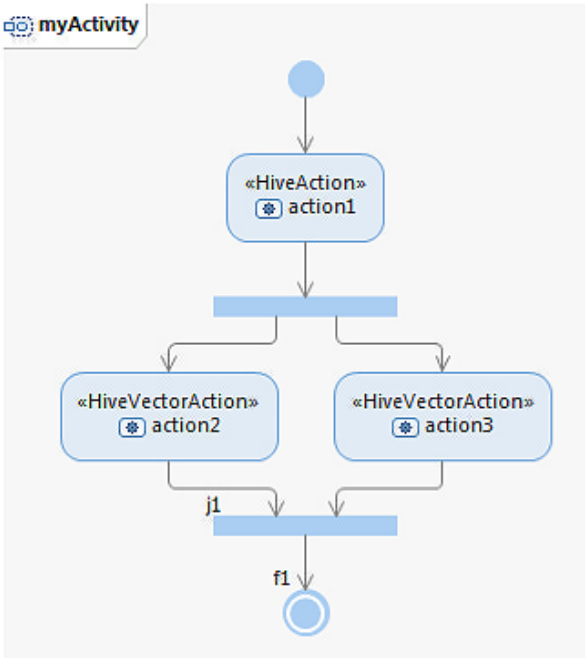

Figure 5: Graphical view of an activity diagram.

4.2 Switching between Graphical and Textual Views

Once the text editor was in place, it was now possible to create models using text. To switch from Xtext to UML and vice versa, two transformations are required: one is to transform graphical models to text models and the other is to transform text models to graphical models. Xtext also provides a mechanism to obtain an XMI version of the model written in text. This way the model could be used as an input to a transformation so that it can be transformed to its UML version. Figure 5 shows an example of a UML model in graphical format and Listing 1 shows its corresponding text model.

Listing 1: Textual view of the activity diagram shown in Figure 5.

```
1  Activity myActivity {
2   nodes{
3     Initial node init
4     CallOperationAction action1 dependsOn init
        testOperation HiveAction testVariable
5     CallOperationAction action2 dependsOn action1
        testOperation HiveVectorAction testVariable
6     CallOperationAction action3 dependsOn action1
        testOperation HiveVectorAction testVariable
7     MergeNode m1 merges (action2,action3)
8     ActivityFinalNode f1 dependsOn m1
9   }
10 }
```

Because we did not want to have any effort applied to these transformations when the Hive profile evolves, we used ATL HOT to generate these transformations instead of writing them manually.

Keeping in mind that the metamodel that was used for the Xtext language is generated from the UML profile and UML subset, most of the information needed for our instance model transformations is available in these two models (UML profile and UML subset). We therefore wrote a HOT that takes the UML profile and the UML subset as input and produces instance model transformations as output.

Generally, an ATL transformation is composed of matched rules. Matched rules define what the source element is and what target element it should be transformed to. Also the rule contains bindings, which define how attributes of a source model element should be transformed into attributes of a target model element. ATL transformations also have helpers, which are functions that can be called within ATL rules. Generating an ATL transformation means generating the rules, bindings in the rules and helpers as well. The code snippet in Figure 6 shows an example of the structure of an ATL transformation.

To generate the matched rules of our transformations, we use information from the Hive profile and UML subset. For example, when generating a transformation that will transform a UML model to an Xtext model the following is done.

- From the Hive profile, each stereotype that is not abstract is transformed to an ATL matched rule. This matched rule will have one source which is an instance of the metaclass extended by the stereotype and two outputs,

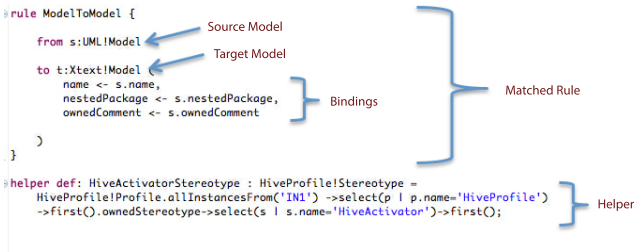

Figure 6: Structure of an ATL Transformation.

one corresponding to the extended metaclass and one corresponding to the stereotype itself.

For example Listing 2, shows an example of a generated rule from a stereotype called `HiveMapToFunction` which extends a metaclass called `Transition`. This rule (Listing 2) will transform an instance of a UML Transition which has the stereotype `HiveMapToFunction` applied to it (Line 3-5 in Listing 2) to two model elements in the Ecore model. The first element is an instance of a class called `Transition` (line 7 in Listing 2) which corresponds to the UML metaclass and an instance of class called `HiveMapToFunction` (line 14 in Listing 2) which corresponds to the stereotype.

- Each stereotype that is not abstract is also transformed to a helper that identifies the stereotype by name from the Hive profile in the transformation. An example of a helper function generated is shown in listing 3 and this helper function is generated from a stereotype called `HiveMapToFunction` and is used to identify this stereotype from the profile. This helper function is called in the generated matched rule in line 4 of Listing 2.
- Properties in the stereotype are transformed into ATL bindings. Examples of these ATL bindings can be seen in line 15 to 19 of Listing 2).

  For instance line 15 shows that the value stored in the property called `threadId` from the `HiveMapToFunction` stereotype will be stored in an attribute called `threadId` in the resulting Ecore model.
- Similarly, properties from the UML metaclasses are transformed into ATL bindings. Examples of these ATL bindings can be seen in line 8-11 of Listing 2.

  For instance line 8 shows that the value of the property named `kind` in UML will be stored in an attribute called `kind` in the resulting Ecore model.

Listing 2: Matched Rule created from a Stereotype

```
1  rule HiveMapToFunctionStereotypedClass {
2   from
3    s : UML!Transition (
4     s.isStereotypeApplied(thisModule.
        HiveMapToFunctionStereotype)
5    )
6   to
7    t : XTEXT!Transition (
8     kind <- s.kind,
9     source <- s.source,
10    target <- s.target,
11    name <- s.name,
12    extension_HiveBaseMapToBehavior <- t1
13   ),
14   t1 : XTEXT!HiveMapToFunction (
15    threadId <- s.getValue(thisModule.
        HiveMapToFunctionStereotype, 'threadId'),
16    newTask <- s.getValue(thisModule.
        HiveMapToFunctionStereotype, 'newTask'),
17    taskPriority <- s.getValue(thisModule.
        HiveMapToFunctionStereotype, 'taskPriority')
        ,
18    actionPackageFile <- s.getValue(thisModule.
        HiveMapToFunctionStereotype, '
        actionPackageFile'),
19    actionPackageName <- s.getValue(thisModule.
        HiveMapToFunctionStereotype, '
        actionPackageName')
20   )
21 }
```

- For the metaclasses in the UML subset, each metaclass that is not abstract is transformed into an ATL matched rule. For example in listing 4, the UML metaclass named Class from the subset generated a rule called ClassToClass. (line 1 in listing 4). This generated rule is used to transform UML classes that are not extended by any stereotype. This constraint can be seen in line 4 of Listing 4.
- Attributes and references of the classes in the UML subset are all transformed into ATL bindings. This works well as long as there is a direct one to one relationship of the attributes from UML to Xtext and vice versa (line 10 to 13 of listing 4).

Listing 3: ATL Helper generated from a Stereotype.

```
1 helper def: HiveMapToFunctionStereotype : PROFILE!
      Stereotype =
2   PROFILE!Profile.allInstancesFrom('IN1') ->select(p
      | p.name='HiveProfile')
3   ->first().ownedStereotype->select(s | s.name='
      HiveMapToFunction')->first();
```

Listing 4: ATL Matched rule generated from a Class in the UML subset.

```
1  rule ClassToClass {
2    from s: UML!Class (
3    s.getAppliedStereotypes() -> isEmpty()
4    )
5    to t:Xtext!Class (
6      name <- s.name,
7      ownedAttribute <- s.ownedAttribute,
8      ownedOperation <- s.ownedOperation,
9      ownedBehavior <- s.ownedBehavior
10   )
11 }
```

If the bindings from UML to Xtext do not have a direct one to one relationship (for example name to name), the HOT transformation needs more information in order to create these bindings. This extra information is related to the manual change that was made to the UML subset (discussed in Section 4.1) that affected the Hive metamodel (in Ecore). These mappings are stored in a trace model. The trace model we used is based on the ATL trace metamodel [5].

For example if we are transforming from a UML model to an Xtext model we know that the controlflow element in UML is not represented as a controlflow in text but as a `dependsOn` attribute. Therefore in our trace model, we create a trace rule called `ControlFlow` and add one link to it which has the source element as the source of the controlflow and the target element as the `dependsOn` attribute. Figure 7 shows an example of this trace model.

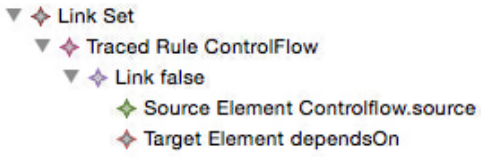

Figure 7: Example of a trace model.

Figure 8 summarizes how the two model transformations are obtained from HOT.

5. Evaluation

To evaluate our approach we applied it to two parts of the Hive DSL at Ericsson: the Hive Behaviour profile and the Hive Structure profile.

The transformation from UML profile to an Ecore model with the help of a UML subset (as described in Section 4) worked for both the profiles. To further test the solution, we had to analyse if there is any information that gets lost or is added during the switch. To achieve this, we used three demo models that are available at Ericsson which

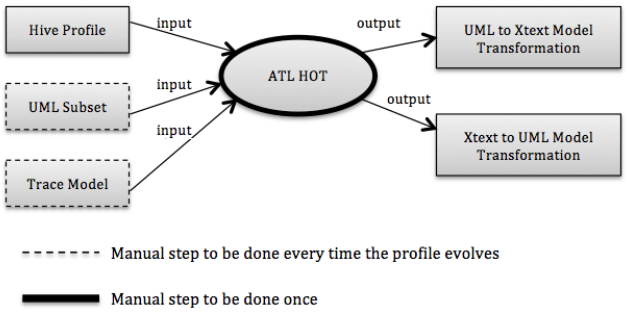

Figure 8: Generating Model Transformations.

are created using the Hive profile DSL. The demo models were provided as input to the transformation from UML to Xtext to obtain an Xtext model. We then converted the Xtext model back to UML using the transformation from Xtext to UML. The original UML model was compared with the one generated from Xtext using EMF Compare (see Figure 9). EMF Compare is an eclipse plugin that is used for comparison and merging of EMF models [4].

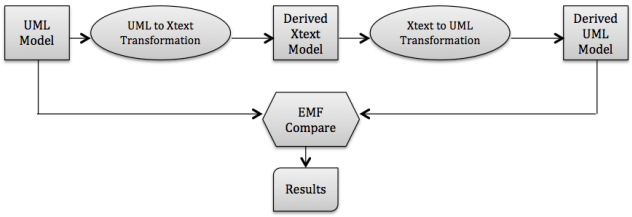

Figure 9: UML to Xtext evaluation process.

From the comparison done, we found out that even though the models were semantically equal, in some cases there were slight differences noted between the UML models and the ones obtained after switching to text and back to UML. On analyzing this we discovered that elements that had no names in UML were causing problems when switching to text. This has been discussed in details in Section 3.6. To overcome this we added an extra helper function to the transformations that generates arbitrary names for model elements in UML that were not named. This also implied that when switching back to UML from the generated Xtext model, these names also appeared in the UML model.

Another issue with our approach was the loss of graphical layout of the UML models. This is because we only transformed the semantic model and did not store the layout information anywhere. After a transformation the diagrams are lost and need to be re-generated. Regeneration uses the auto-layout that RSA provides. We did not implement a solution for this but propose that one can use incremental model transformations to solve the problem. Incremental model transformation is the kind of transformation where instead of the transformation creating a new target model every time, it checks which model elements are changed and updates only

those elements in the target model. The target model is not recreated but rather updated with changes from the source model. Section 6.2 provides a further discussion on this.

Another comparison was made when switching from instance models created using the Xtext editor to UML and again back to Xtext. Since there were no models available in text, new ones were created for the purpose of this comparison. The comparison was made using the textual quick diff functionality in Eclipse (see figure 10). This is a functionality that lets a user compare text files side by side.

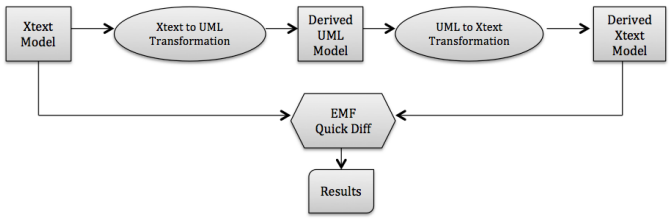

Figure 10: Xtext to UML evaluation process.

From this comparison, if a textual model had no comment inserted then the resulting model transformed to UML and back to Xtext was identical to the original text model. But when the text model had comments, these comments were lost and making the resulting model not identical to the original model during comparison. All the custom pretty printing that the user had made in the text is also lost after transforming to UML and back to Xtext. This challenge has been discussed in Section 3.3 and is due to the fact that the transformation created a new model every time it was run. Here we also suggest the use of incremental model transformations to solve this problem.

6. Discussion

This section discusses our approach in relation to the challenges that have been identified in Section 3.

6.1 Addressed Challenges

Our approach addresses four challenges so far: Challenge 3.2, 3.5, 3.6 and 3.7. Challenge 3.2 is about synchronization, how can we keep graphical and textual models in sync. With our current approach, for instance, when a user is editing a model in one graphical view and decides to switch to textual view, all the information that was in graphical view is transformed to a textual version of the model. The vice versa is also true. This ensures that the models are always in sync if the user does not edit the two different notations of the same model at the same time.

Challenge 3.5 is about how to achieve a metamodel for a textual DSL from the UML metamodel that has only the elements that are actually used by the company. This has been done by manual identification of metaclasses that are extended by the Ericsson DSL and metaclasses that are in use without being extended by any stereotypes. This worked because Ericsson's DSL rarely changes. However for companies where their DSLs change frequently automation of this step is a more suitable solution. There are approaches being researched on how to obtain this subset of UML automatically. For instance [2] suggested a way to split the UML metamodel into sub-metamodels according to the diagram type. But this still requires the users to identify key elements that are used in each diagram type. It also requires a user to build a tool that can do the splitting.

In [3], the authors propose a model shrinking approach that preserves the model elements' types. In this approach, an instance model that contains all the elements of the desired metamodel subset is needed. From this model, a selector tool is written that can extract the classes from the metamodel and create a metamodel subset. The metamodel subset is then shrunk to only the necessary elements needed. This method is usable if one has existing instance models that use all the classes that are needed in the subset of the UML.

Challenge 3.6 is about the fact that textual editors use qualified names to make references between elements. In our approach this has been solved by having a validation in the transformation that checks if model elements have names. If not we generate unique names for these elements. The names that we generate follow a pattern that corresponds to the type of model element so that a user reading the text model is able to understand it. For example if merge nodes are missing names we generate the names m1, m2, m3 and so on.

Challenge 3.7 is about how one can switch between models that have errors. In our approach this has been addressed by putting a layer of validation in the tool chain. Before transforming a model, one should validate the model first. If the model is inconsistent, then all errors have to be fixed before the model is switched. This is a prevention measure, thus future work is still needed on this.

6.2 Proposed Solutions for Non-Addressed Challenges

Our approach does not solve three challenges, which are challenge 3.1, 3.3 and 3.4 . However, during the study, developed a proposal on how these challenges can be addressed.

Challenge 3.1 which is about storage and versioning of models can be addressed in the following manner. First by the organisation putting in place a policy on which version of the model should be stored. For example they can decide to store the text or graphical version or both. Either way, all the designers should be aware of this policy. Second, with the policy in place then Version control software to support versioning of the stored models should be put in place. Lastly if only one version is stored, automation mechanisms should be implemented to make sure those working with the other version that is not stored can work as smoothly as possible.

Storing both versions of the models also automatically solves challenge 3.4, which is model references since all the references in graphical models will be to graphical model elements and the links in textual models will be to textual model elements.

We propose addressing challenge 3.3 which is about graphical layout of the model and pretty printing using incremental model transformations. Incremental model transformations will work if both versions of the models (graphical and textual) are stored in the repository. Since the graphical and textual models are related by transformations, making these transformations incremental means that every time a transformation is executed, the target model will only be updated on parts that have changed. This will keep the layout of existing model elements in place.

Since we did not implement this solution, we cannot give definite answers about its effectiveness. Moreover, we are aware of the limitations that incremental transformation has, for instance when adding new model elements. These elements do not have any layout information and they could be placed anywhere in the diagram. The developer will have to arrange them manually when the transformation is run. This problem can be addressed by using automatic layouts which can be applied to the new and deleted model elements and preserve the layout of the already existing model elements as much as possible. This kind of auto-layout expands the model to make room for new elements and shrinks it when elements are deleted from the model [24].

Also incremental transformations sometimes fail to maintain consistency between models due to reasons like erroneous models, or changes in the source model that have more than one way of being propagated in the target model. This problem has been discussed in detail in [26].

7. Related Work

Due to the nature of the problem we explored, our related work is divided into two parts. The first part discusses related work on using both graphical and textual editors for a UML profile-based DSL while the second part is on how to integrate/combine UML profiles with Ecore-based tools.

7.1 Graphical and Textual Editing for UML

There are various approaches that have been investigated for editing UML models using textual syntax. Tools such as PlantUML [21], TextUML [13] and PlantText [20] all aim at establishing a standard textual language for UML as a GPL but none of them mention the use of profiles.

Other prototypes have been implemented to represent parts of the UML metamodel as text. For instance in [19], a textual editor for the Action Language for Foundational UML (Alf) has been developed based on Xtext. The textual editor is implemented in such a way that it can be used to edit only parts of a UML model and not the whole UML model. A different approach is proposed in [23], where textual editors are embedded in graphical editors. This way when modelling using graphics, designers have an option to bring up a text editor as a pop-up box that they can use to edit model elements of graphical models. Our approach differs from these, as we want to be able to view and edit the whole model as a graphical or in textual model.

In [18] the authors describe an approach to use a textual editor to correct UML models that have errors. They implement a prototype using tUML, which is a textual concrete syntax for a specific UML subset. The subset supports class diagrams, state charts and composite structure diagrams. In this case the UML models are transformed to text and constraints are implemented in the textual editor. The developer fixes the errors in the textual editor and transforms the model back to graphical UML models. In this approach the transformations are not generated but written manually.

When it comes to switching between graphical and textual views, [7] proposes two approaches to facilitate the transformation of models that have both graphical and textual notation (i.e. one model containing parts written in UML and parts in text). The first approach is called Grammarware and refers to a text to text transformation of the models. With this approach the models are exported as text and transformation is done from text conforming to one metamodel to text conforming to another metamodel. The second approach is called Modelware and refers to a model to model transformation. In this approach, a model containing the graphical and textual content is transformed to a fully graphical model. This is done by converting the text part to its corresponding model element in the graphical metamodel. Our approach however proposes a way to have a DSL supporting both graphical and textual views and the possibility to switch between them but not combining text and graphics in the same model.

Projectional editing is another research area which investigates the use of various concrete notations for editing models. Projectional editing is a technology that displays the concrete syntax of models as projections. The concrete syntax of the model is not stored but only the abstract syntax of the model is persistent [27]. With projectional editing, the user edits the AST of the model directly and what the user sees on the screen is merely a projection. The main advantage of this technology is that the model can be projected in various notations depending on what the user prefers (textual, graphical, tabular or even a combination of these). However, this technology was not adopted in this study because it adds a lot of overhead to developers when it comes to learnability and familiarization [28]. Since projectional editing does not rely on parsers and hence has no grammar, it provides a different way of editing for which designers who are used to grammar based editors need time to get used to. Also, current reasonably mature projectional editors such as Jetbrains MPS [17] are also not yet integrated into Eclipse. For these reasons our approach uses Xtext which allows the use of grammar and provides functionality such as code completion and syntax highlighting out-of-the-box.

7.2 Bridging UML Profiles and Ecore DSLs

Due to the fact that one of the problems we had was how to combine UML profiles and Ecore, we also include related

work on how to bridge UML and Ecore. The state of the art in this area is described in the following.

Most research on the field of bridging UML and Ecore DSLs have focused on the automatic generation of UML profiles from an Ecore-based DSL metamodel. Some notable examples in this context is research done in [9], [11], [29]. In these papers the main idea is how to systematically derive a UML profile from an existing DSL metamodel.

There also exists research on how UML profiles and Ecore DSLs can be used together. In [1] the author proposes a way to bridge UML and Ecore using model to model transformations. They use ATL HOTs and Atlas Model Weaver (AMW) for the transformation of models from UML profiles to Ecore and vice versa. This approach is quite similar to our approach but they do not generate the Ecore metamodel but assume that it already exists. In [30] another similar approach has been presented to bridge UML profiles and Ecore DSLs. In this work, the authors use a dedicated bridging language which is based on the AMU metamodel. The bridging language defines a model with mappings (weaving model) from UML profiles to an Ecore DSL and transformations are generated from this weaving model. Our work is different from this because we do not generate the UML profile but start with an existing profile. We also do not use the AMW model but a trace model whenever the mappings from UML to the Ecore DSL are not one-to-one mappings.

In [10], the authors describe how to interchange DSL and UML models using UML profiles. They propose an approach of first automatically generating a UML profile from a DSL metamodel (which can be in Ecore) using an integration metamodel. The integration metamodel is used to create models that define the relationship between the DSL metamodel and UML. From such an integration model, a UML profile representing the DSL metamodel is generated. During the generation of the UML profile, mappings from the integration model to the profile are also generated. The transformation from UML models to models conforming to the integration metamodel is done using the mappings generated when generating the profile. The transformation from the intermediate model to a model conforming to the Ecore DSL metamodel is done using the mappings from the Ecore DSL to the intermediate model which were manually created. This approach has the advantage of generating mappings from UML to the intermediate metamodel automatically but since the profile is generated from the DSL, it also means that the users using UML can only use classes that are extended by the profile. Any un-extended class will lack its mapping back to the Ecore DSL metamodel. Our approach solves this by including a subset of the UML metaclasses that are used even when the metaclasses are not extended by any stereotype.

In conclusion, most of the research on integrating textual and graphical editors for UML profile-based DSL is still on prototype level. Even though there are several text based tools implemented for UML as a GPL, the adaptation of these tools has not been reported in an industrial context, making it hard to know to what extent they can be used and what challenges they bring. Furthermore, the work reported on bridging UML and Ecore is also on a theoretical level accompanied with toy examples. None of the approaches report an industrial application of the bridge except the work of Jouault and Delatour in [18].

8. Conclusion and Future Work

In this paper we have presented an approach to integrate textual and graphical editors for a UML profile-based DSL using model transformations. We have shown that it is possible to have the two editors working in the same Eclipse environment. Even though this study has been carried out in one company, the results can be generalized to any company using a UML profile-based DSL that wants to additionally have a textual editor for it. We have also presented challenges that are encountered in industry when combining a UML profile-based graphical editor with a textual editor.

For future work we propose further research on how to effectively and automatically obtain a UML subset from the UML metamodel. Also research should be done to investigate ways to maintain the layout of graphics and format of text when using multiple editors. As mentioned previously this information is lost once a designer switches from one view to another and then back. Another aspect that should be researched is how to store models and how to keep these models in sync when working with version control tools such as Git. As some designers can decide to model using text and others decide to model using graphics, there has to be a standard policy for storage of these models. Should they be stored as text or as graphics or should one store both versions of the model? The research here could also shed light on what are the challenges when storing textual models or graphical models or both.

In connection to storage of models, strategies should be developed for merging models written using the different notations. For instance if one designer is editing the model in text and the other one is editing the same model using the graphical format, how will they merge these changes.

Inter-model referencing is also another area that requires further research. Inter-model references here means that one model has references to one or more elements in other models. For instance a graphical model can have references to other graphical models. When it comes to switching between views this has to be considered. Investigations are required on whether only one model should be switched to textual syntax and keep its references to the graphical models, or whether the referenced models will need to be converted to text as well.

Our study also investigated an efficient way of updating the DSLs and textual editor using transformations. However we did not look into how the updated DSL will affect the already existing models and how these existing models can be

migrated to conform to the new DSL. This is a very important aspect and is also crucial future work.

References

[1] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, volume 5. Citeseer, 2005.

[2] J. H. Bae, K. Lee, and H. S. Chae. Modularization of the UML metamodel using model slicing. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1253–1254. IEEE, 2008.

[3] A. Bergmayr, M. Wimmer, W. Retschitzegger, and U. Zdun. Taking the pick out of the bunch-type-safe shrinking of metamodels. *Fachtagung des GI-Fachbereichs Softwaretechnik*, page 85, 2013.

[4] Eclipse.org. Emf compare - compare and merge your emf models, 2014. URL `http://www.eclipse.org/emf/compare/overview.html`. Accessed : 2014-06-30.

[5] Eclipse.org. File:ATL EMFTVM trace.png, 2015. URL `https://wiki.eclipse.org/File:ATL_EMFTVM_Trace.png`. Accessed : 2015-05-18.

[6] S. Efftinge. Xtext - language development made easy!, 2014. URL `http://www.eclipse.org/Xtext/`. Accessed: 2014-06-13.

[7] L. Engelen and M. van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, 253(7):105–120, 2010.

[8] Gentleware.com. UML-to-ecore plug-in, 2014. URL `http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/ecoreguide.html`. Accessed : 2014-06-30.

[9] G. Giachetti, B. Marín, and O. Pastor. Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles. In *Advanced Information Systems Engineering*, pages 110–124. Springer, 2009.

[10] G. Giachetti, B. Marín, and O. Pastor. Using UML profiles to interchange DSML and UML models. In *Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on*, pages 385–394. IEEE, 2009.

[11] G. Giachetti, M. Albert, B. Marín, and O. Pastor. Linking UML and MDD through UML profiles: a practical approach based on the UML association. *J. UCS*, 16(17):2353–2373, 2010.

[12] git scm.com. Git, 2014. URL `http://git-scm.com/`. Accessed : 2014-06-25.

[13] Github.io. Textuml toolkit, 2015. URL `http://abstratt.github.io/textuml/readme.html`. Accessed : 2015-06-01.

[14] R. C. Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009.

[15] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[16] P. Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[17] JetBrains. Jetbrains :: Meta programming system - language oriented programming environment and dsl creation tool, 2014. URL `http://www.jetbrains.com/mps/`. Accessed : 2014-06-15.

[18] F. Jouault and J. Delatour. Towards fixing sketchy uml models by leveraging textual notations: Application to real-time embedded systems. In *14th International Workshop on OCL and Textual Modeling: Applications and Case Studies*, pages 73–82. CEUR-WS, 2014.

[19] C.-L. Lazăr. Integrating alf editor with eclipse uml editors. *Studia Universitatis Babes-Bolyai, Informatica*, 56(3), 2011.

[20] Plaintext.com. Planttext, 2015. URL `http://www.planttext.com`. Accessed : 2015-06-01.

[21] Plantuml.sourceforge.net. Plantuml, 2015. URL `http://plantuml.sourceforge.net`. Accessed : 2015-06-01.

[22] D. S Wile. Supporting the DSL spectrum. *CIT. Journal of computing and information technology*, 9(4):263–287, 2001.

[23] M. Scheidgen. Textual modelling embedded into graphical modelling. In *Model Driven Architecture–Foundations and Applications*, pages 153–168. Springer, 2008.

[24] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett. An effective layout adaptation technique for a graphical modeling tool. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 826–827, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL `http://dl.acm.org/citation.cfm?id=776816.776971`.

[25] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[26] P. Stevens. Bidirectionally tolerating inconsistency: partial transformations. In *Fundamental Approaches to Software Engineering*, pages 32–46. Springer, 2014.

[27] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[28] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *Software Language Engineering*, pages 41–61. Springer, 2014.

[29] S. Walderhaug, E. Stav, and M. Mikalsen. Experiences from model-driven development of homecare services: UML profiles and domain models. In *Models in Software Engineering*, pages 199–212. Springer, 2009.

[30] M. Wimmer. A semi-automatic approach for bridging DSMLs with UML. *International Journal of Web Information Systems*, 5(3):372–404, 2009.

[31] www.uml diagrams.org. Activity diagrams, 2014. URL `http://www.uml-diagrams.org/activity-diagrams.html`. Accessed: 2015-04-13.