Conception of Ambiguous Mapping and Transformation Models

Christopher Martin[1], Matthias Freund[1], Henning Hager[2], Annerose Braune[1]

Institute of Automation, Technische Universität Dresden, 01062 Dresden
[1] {christopher.martin, matthias.freund, annerose.braune}@tu-dresden.de
[2] henning.hager@monkey-works.de

Abstract. Model transformations are the linking element between the different levels of abstraction in the model-based user interface development. They map source elements onto target elements and define rules for the execution of these mappings. Approaches for the reuse of transformation rules use formal transformation models, which only specify the mappings and abstract from the implementation. Current solutions are usually only able to describe unambiguous (1-on-1) mappings. In general, however, there are ambiguous (1-on-n) mappings from which the unambiguous mappings are only chosen during the design process. The knowledge which source element can be mapped onto which target elements is to date not being formalized. This paper therefore presents a proposal for an ambiguous mapping and transformation model and describes its usage in an iterative development process.
Keywords: Model-based User Interface Design, Model Transformations, Mapping Model, Transformation Model, Iterative Development

1 Introduction

Developing user interfaces is a cost- and time-consuming task, especially if multiple platforms in terms of hard- and software have to be supported. Model-driven development processes for user interfaces (UIs) try to handle this problem by allowing a designer to specify the functionality of a user interface by means of models. These models abstract from implementation details and can ideally be used to generate executable user interfaces for different target platforms. In general, the models used in such model-driven development processes are defined on different layers of abstraction. The generally accepted *Cameleon Reference Framework* (*CRF*) constitutes four layers of abstraction [1, 2]: The *Task & Concepts* (*T&C*) level allows defining the tasks that the user shall execute with the user interface. An *Abstract User Interface* (*AUI*) model describes the UI in a modality-independent way. In contrast, a *Concrete User Interface* (*CUI*) model is already bound to a specific modality but still independent of an underlying computing platform. A *Final User Interface* (*FUI*) is an executable UI dependent on a specific (class of) computing platform(s).

Model transformations either convert more abstract user interface models into increasingly concrete models (cf. the vertical transformations in Fig. 1) or they transfer models of a certain level of abstraction to another context of use (cf. the horizontal transformations in Fig. 1). In any case, transformations constitute the connection between the different layers. Therefore, their quality and reusability is a key factor for the efficiency of the entire model-based approach.

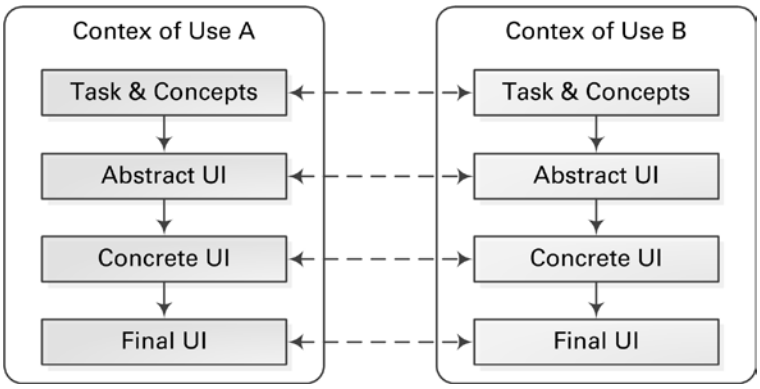

Fig. 1. Simplified version of the Cameleon Reference Framework [1] as introduced in [2]

Model transformation rules incorporate two basic tasks (comp. [3]): (1) the description of mappings from source elements onto target elements and (2) the description of the execution of these mappings including the generation of the target model(s). While standard (hard coded) model transformations usually do not separate the mappings from their execution but combine them in the transformation code, there are already some approaches that favor the definition of formal mapping respectively transformation models. These models describe which model element or meta-model element of the source language can or shall be mapped onto which element of the target language. Thereby, they are independent of the technology that is used afterwards for the implementation of the transformation execution. Consequently, as shown in Fig. 2, they separate the functionality of the transformation (the description of mappings) from the implementation details (the execution of these mappings). The advantage of mapping models is that the correlations between elements of the source and the target (meta-)model can be configured instead of having to be programmed. In this case, the creation of a model transformation only requires knowledge about the application domain and not about implementation language details.

Existing mapping model specifications essentially only allow unambiguous (1-on-1) mappings that map one source element onto exactly one target element. For example, a mapping model from AUI onto CUI elements would describe that – in case of a graphical user interface – every modeled AUI-level *"selection element"* has to be mapped to e.g. the drop-down list element at CUI-level.

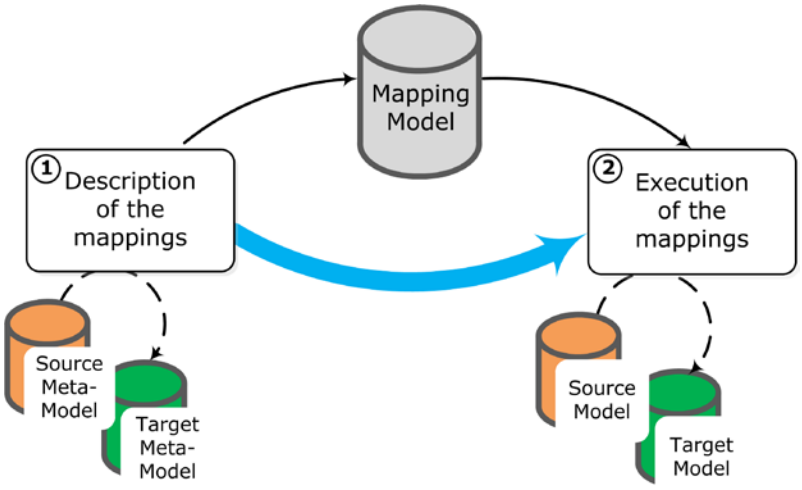

Fig. 2. Separation between description and execution of mappings by means of a mapping model

However, Puerta and Eisenstein show that not unambiguous (1-on-1) but ambiguous (1-on-n) mappings are frequent for the model-driven development of user interfaces [4]. That means, that the ”selection element” instruction at AUI-level in general could be visualized by means of a drop-down list as well as by a set of radio buttons or a vocal selection. Consequently, mapping models for user interface design require multiple instead of unique target model elements. During the design process these ambiguities have to be resolved to allow the transformation of the model. Which of the (ambiguous) mappings is used depends, among others, on the target platform, the developer's preferences and on company guidelines. It is important to note that the resolving of the ambiguities has to be performed for each element of the source model as, in general, not every instance of a source meta-model element shall be mapped to an instance of the same element of the target meta-model. For example, depending on the number of the possible choices of a “selection element”, either a drop-down list or radio buttons could be reasonable.

The model-based user interface development has to support iterative development processes to be widely accepted by UI designers because user interfaces are usually not developed in a single linear development process but rather in multiple development cycles. Therefore, the proposed mapping model should not only be able to specify ambiguous mappings but also save the information about the chosen mappings for each element. This allows for an easy reuse of transformations and the chosen mappings for every iteration step of the development process.

Hence, this paper presents a meta-model that (a) supports the definition of ambiguous mappings for the transformation of UI models and (b) stores the chosen mappings of the executed transformation.

2 Related Work

In regular transformations, both the mapping of a source element onto a target element as well as the generation of the target model are hard coded for one specific pair of UI description languages in an implementation technology such as ATL [5], QVT [6] or XSLT [7]. The occurring possible (ambiguous) mappings are usually either not defined so that only one of the possible target elements is used or only programmed directly into the transformation in an informal way [8].

Bézivin et al. show in a fundamental work that, in contrast to the hard coded transformations, it is also possible to formally specify the mappings in so called *Transformation Models* [9]. Therefore, the mappings of the meta-model elements are described by formally assigning source and target elements to each other. This approach is used by the UI description language UsiXML, which specifies a transformation model that allows the description of a model transformation based on graph grammar [10]. The MARIA[1] description language family [11] also allows the definition of unambiguous mappings with the help of the graphical editor *MARIAE* [12]. However, in both cases the unambiguous mappings apply to every element of the same type, i.e. every instance of the same meta-model element. This means that e.g. every "*selection element*" can only be mapped to a drop-down list for the entire user interface. An alternative mapping to a radio button or a vocal selection is only possible in another transformation for a different context or by manually changing the target model. Consequently, there is no reusable information about the ambiguous mappings.

First approaches for ambiguous mapping models are made by Hager et al. [8] and by Aquino et al. [13]. Hager addresses this problem by implicitly describing all possible ambiguous mappings for one pair of description languages, namely *DISL*[2] [14] at the AUI level and *Movisa* [15] at the CUI level, in (hard coded) transformation rules. The UI designer can choose an unambiguous mapping in an interactive process. With this approach, a UI designer can map a DISL element type (AUI level) onto various Movisa element types (CUI level) even in a single user interface. The choices made are then stored in a *Persistent Transformation Mapping Model* (*Petmap*) and are available for further iteration steps. However, the ambiguous mappings for this pair of UI description languages are also not formalized as they are implicitly included in (hard-coded into) the transformation rules.

In [13], Aquino et al. propose *Transformation Templates* that extend the UsiXML Transformation Model defined in [10]. They allow defining (possibly ambiguous) mappings by the specification of a selector that is applied to the target meta-model. Furthermore, weighting factors can be specified that reference the UsiXML context model and that are used if multiple target elements for one source element are possible. However, both [8] and [13] specify their transformation models imperatively: They include implementation details of the transformation. Because of the imperative description of the mappings in this approach it is possible to use variables, conditions and loop constructs. For example, the specification of the selectors requires deep

[1] Modelbased lAnguage foR Interactive Applications
[2] Dialog and Interface Specification Language

knowledge of the syntax and semantics of the transformation templates and thus is not applicable by UI designers but only by transformation language experts.

In contrast to imperative modeling, declarative modeling of transformations simply states which elements shall be mapped onto each other but not how these mappings are implemented by the transformations: It describes the “what” and not the “how”. The advantage of a declarative modeling is the simple and clear notation suitable also for a UI designer without detailed knowledge of implementation technologies. The disadvantage of this approach is the limited language scope that e.g. does not allow expressing dependencies between different mappings.

Nevertheless, the creation of imperative transformation models requires advanced knowledge about the syntax and semantics of the implementation technology and therefore a transformation expert. Consequently, the approach presented in the following makes use of a declarative mapping model.

3 Conception of a Meta-Model for the Description of Ambiguous Mappings

This section introduces the concepts used to realize the requirements for ambiguous mappings for transformations of UI models and for storing chosen concrete mappings derived in Section 1. The realization of these concepts as the *Persistent Ambiguous Mapping and Transformation Model* (*PAMTraM*) will also be described. The workflow of this solution including all involved models and meta-models as well as the role of the PAMTraM is shown in Fig. 3.

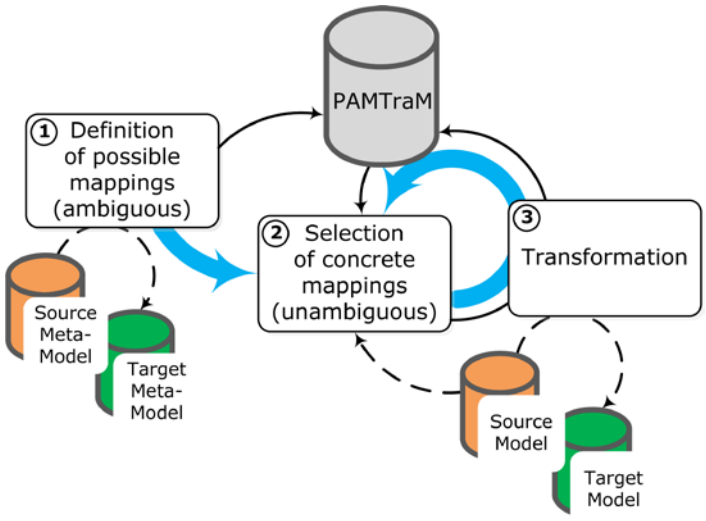

Fig. 3. Proposed workflow and involvement of the PAMTraM

During step 1, all ambiguous mappings from elements of the source meta-model onto elements of the target meta-model have to be defined. These mappings are stored in the PAMTraM. This step has only to be performed once for every language pair.

Afterwards, step 2 of the workflow consists of choosing the concrete mapping for every element of the source model. Therefore, the ambiguous mappings defined in step 1 are read from the PAMTraM and evaluated by a selector. This selector can e.g. be a UI designer or additional constraints (e.g. platform constraints) that allow an automatic evaluation.

Finally, step 3 constitutes the actual execution of the transformation. The execution uses the unambiguous mappings defined in step 2 to create the target model from the source model. Furthermore, the performed transformation including the selected unambiguous mappings is stored in the PAMTraM so that it can be reused for the next iteration cycles. In addition, the information about chosen mappings can e.g. be evaluated if statistical selection mechanisms are to be applied during step 2.

Steps 2 and 3 can be repeated to allow an iterative development process. The following sections describe the realization of the two parts of the PAMTraM by means of its meta-model.

3.1 Definition of Ambiguous Mappings

The mapping rules are defined on the meta-model level [3] because the allowed mappings of elements are mainly based on the UI description languages. They specify mappings from a source to one or more target meta-model elements. The mapping model must therefore allow the definition of those mappings and thus reference the source and the target meta-model. To ensure a comfortable creation of the mapping models, a declarative description will be implemented, such as e.g. already used in MARIAE's transformation editor for the definition of the unambiguous mappings. This approach allows the definition of mappings at a high level of abstraction. This way, a domain expert can create the mapping model and a transformation expert is not required.

The declarative description of the mappings, compared to an imperative approach, requires additional effort for the implementation of the mapping model into an executable transformation rule. But this additional effort is put into perspective, since such an implementation is only required once per UI description language pair. Another positive effect of the declarative description is the capability to easily define ambiguous mappings that assign one source element to *multiple* target elements. Fig. 4(a) shows the structure of a declarative, ambiguous mapping (*Mapping*). It enables the assignment of a source meta-model element to all eligible target meta-model elements.

As it is not reasonable to define multiple *Mapping* elements (each in turn defining multiple target meta-model elements) for one source meta-model element, additional constraints or validation steps should ensure that only one mapping is defined for every element of the source meta-model.

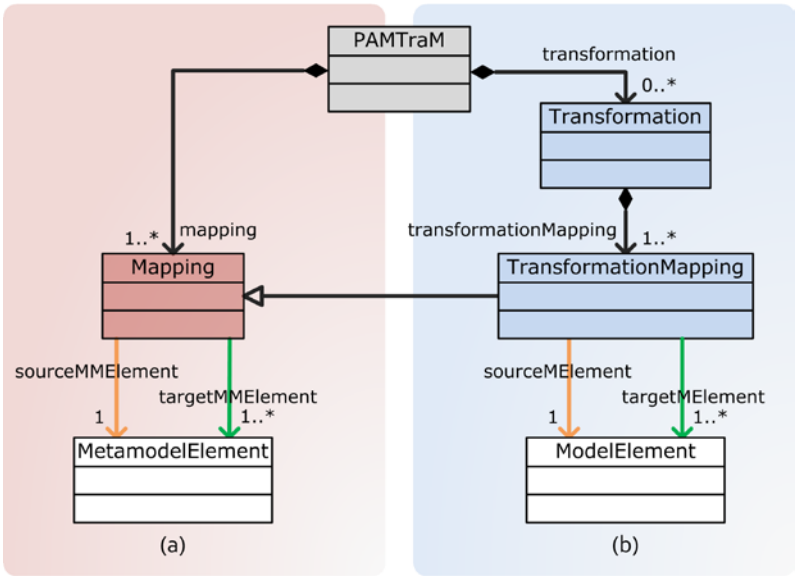

Fig. 4. Resulting implementation of the PAMTraM for the storage of ambiguous mappings (a) and performed transformations (b)

It is important to note that the element *MetamodelElement* can be *any* element of any meta-model – it is not limited to classes (if a UML-like structure is used for the meta-models) but can also be a class attribute. This is necessary because – depending on the meta-models used – it cannot be ensured that merely classes are mapped onto each other. This is best explained by a simple example: In DISL (see Section 2), the different widgets that can be used to realize a user interface are not represented by different classes of the meta-model but by different enumeration literals of the enumeration *GenericWidgetType*. Consequently, DISL uses a simple meta-model class *Widget* with an attribute *genericWidget* of the type *GenericWidgetType*. Fig. 5 shows the relevant excerpt of the DISL meta-model.

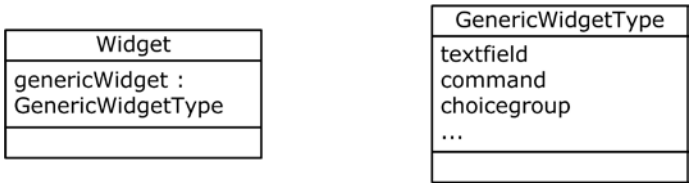

Fig. 5. Excerpt of the DISL meta-model: realization of widget types

Consequently, when using DISL as source or target meta-model, the mappings do not have to be specified for the class *Widget* but for the different enumeration literals like *textfield* or *command*.

3.2 Storage of Concrete Mappings

Modeling ambiguous mappings allows the explicit definition of all possible mappings of source to target element types. Before a concrete transformation (as an instance of the ambiguous mappings) can be executed, the ambiguities need to be resolved. This resolution depends on numerous factors such as the specific context of a user interface or industry- or company-specific guidelines. Furthermore, the ambiguities usually have to be resolved for every element individually, because it cannot be assumed that a source element type should always be mapped to the same target element type (cf. [16]). This process may be time-consuming – especially if a user is involved in the resolving of ambiguities – and has to be repeated for every iteration cycle even if only minor changes have been applied to the UI model. Consequently, the design decisions, which are made during a transformation, should be saved, so that they can be reused for iterative improvements of the user interface: In that case, an unambiguous mapping has only to be determined for the model elements that have been added or modified during the last iteration cycle whereas the stored concrete mappings of a previous transformation can be reused for the unaltered elements.

For this purpose, the meta-model element *Transformation* (see Fig. 4(b)) represents a concrete transformation that was executed for a specific user interface in a specific context, which can be saved in the PAMTraM. Gruhn et al. define a transformation as an instance of a mapping on the model level [3]. Therefore the unambiguous mappings that are determined for each model element during the transformation are saved as *TransformationMappings* (see Fig. 4(b)). Analogous to the *Mappings*, a declarative structure is used again. In comparison to the *Mappings*, *TransformationMappings*, however, map concrete (model) elements onto each other. Otherwise they have similar characteristics like the involved element types. Hence, an inheritance relationship between *TransformationMapping* and *Mapping* was used.

It should be noted that not necessarily only a single target element must be selected: Especially in multimodal user interfaces, it is often useful to map an AUI element to multiple CUI target elements, so as to combine a graphical output of an alarm with an auditory output of the same alarm. This relationship is represented by the *TransformationMapping* that allows the mapping onto multiple target model elements.

Those two (basically independent) requirements – modeling of ambiguous mappings and saving of performed transformations – were jointly implemented in a meta-model, since they are both specific for each pair of meta-models. In addition, when re-transforming a user interface, both, the basic mappings as well as the specific performed transformations, are needed. A common storage of all performed transformations in one model also allows for a good starting point for further developments on the reduction of the ambiguities like the statistical analysis of the determined mappings.

The resulting meta-model PAMTraM (cf. Fig. 4) is applicable for any combination of target and source meta-models, because there are no restrictions or statements made about their structure and only general concepts were implemented. Beyond that, it allows the use of different methods to resolve the ambiguities as it only stores the resulting mappings.

3.3 Tool Support for the Application of the PAMTraM

In order to make the workflow (depicted in Fig. 3) – and therewith the concepts introduced in Sections 3.1 and 3.2 – applicable, tools supporting the UI designer in performing the three steps of the workflow should be created. Therefore, we have implemented the PAMTraM by using the *Eclipse Modeling Framework* (*EMF*) [17].

During the first step of the workflow, the UI designer has to be supported in the definition of the ambiguous mappings. Although the EMF automatically creates simple, generic editors (e.g. a tree editor), the usability of these editors is not very high as they are not tailored towards the needs of the user. Instead, the design of a specialized *concrete syntax*[3] supporting the modeler to think in his domain is necessary [18].

In general, concrete syntaxes can be divided into *textual* and *graphical* representations, both with different advantages and disadvantages. For example, whereas a textual concrete syntax may enable a faster specification of models, a graphical concrete syntax may help in visualizing relationships. As multiple concrete syntaxes can be used simultaneously, the advantages of both types can be combined. As a first prototype, we have implemented a textual concrete syntax for the PAMTraM by means of the tool *EMFText* [19]. In order to assist the modeler, a content assistant and code completion are integrated.

Regarding the second step of the workflow, the necessary tool support can be divided into two parts: the implementation of the selection mechanism and the reuse of previous transformations in case of an iterative development.

As our approach does not dictate the used mechanism for the selection of concrete mappings during the second step of the workflow, the required tool support cannot be generalized. For example, if the UI designer shall choose the concrete mappings, a tool supporting this user interaction has to be implemented. This could e.g. be a dialogue that presents the different possibilities to the user and asks him to select one or more of the alternatives. If, however, statistical algorithms or platform constraints shall be applied, these mechanisms have to be implemented.

In case of an iterative development, a previous version of the user interface may already have been transformed once or repeatedly. In this case, the chosen (unambiguous) mappings for those parts of the source model that have not been altered during the iteration should usually be reused to keep the resulting parts of the target model that are not involved in the iterative changes consistent. Consequently, the selection mechanism should only be applied to those parts of the source model that have been changed during the iteration step. The described behavior has been implemented by means of a tool that reads previous transformations from the PAMTraM. Afterwards, the user is asked if one of the previous transformations (and with those the results of the used selection mechanism) shall be reused or if the selection mechanism shall be applied for every model element.

The tool support necessary for the third step of the workflow consists of the transformation executing the selected concrete mappings (and thereby generating the target model) and storing them in the PAMTraM for further iteration cycles (cf. Sec-

[3] The term *concrete syntax* denotes the concepts used for the representation of models.

tion 3.2). However, as this transformation is dependent on the used source and target meta-model, its implementation cannot be generalized either.

An exemplary implementation of the tools supporting the workflow is presented in the following case study.

4 Case Study

This section demonstrates the application of the presented approach based on the development of a user interface for the process plant shown in Fig. 6. The process plant consists of three tanks that are connected via pipes. For this case study, the UI shall enable the user to select one of the three tanks resulting in the illustration of the fill level of the selected tank. Two possible final user interfaces are presented in Fig. 7: The UI on the left makes use of a set of radio buttons to select the tank and of an animated image to display its fill level. In contrast, the UI on the right uses a drop-down list for the selection of a tank and a textual representation of its fill level.

The starting point of the case study is an AUI model expressed in DISL. By application of the PAMTraM, a concrete model of the UI, based on MARIA, shall be generated. Therefore, the XML schemata of both languages have been converted to EMF-based meta-models for easier processing in the EMF environment used for the implementation of the tools described in Section 3.3. The (model-to-text) transformation generating the final UI has only been implemented prototypically to demonstrate possible representations of the resulting UI.

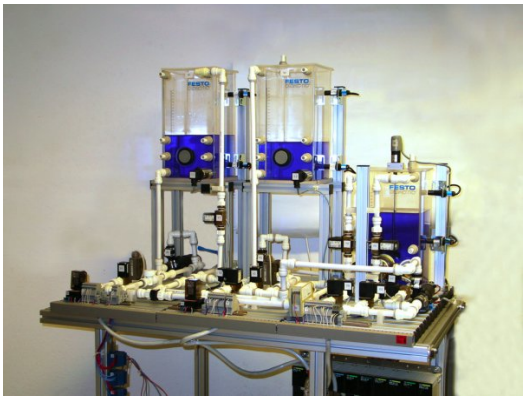

Fig. 6. Depiction of the process plant used in the case study

DISL is a dialogue descriptions language, which is situated on the AUI level of the Cameleon Reference Framework [14]. DISL offers ten *Widgets*, represented by *EnumerationLiterals* (cf. Fig. 5), for the description of user interfaces. In the DISL model (AUI model) of the case study, the structure of the realized process visualization is described. For this purpose, a widget to display the currently shown tank and its level (*textfield*), a selection element for the three tanks (*choicegroup*) and a widget to confirm that choice (*command*) are modeled.

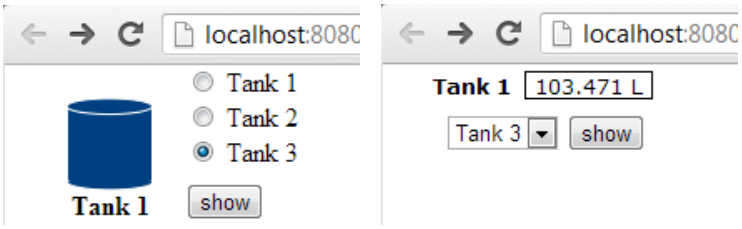

Fig. 7. Two versions of the final UI based on different results of the selection of unambiguous mappings (cf. Fig. 3)

MARIA is a description language for graphical, vocal and multimodal user interfaces, which covers the top three abstraction levels of the CRF [11]. For this case study, only the CUI level is used, in which MARIA, among others, defines the *MultimodalDesktopCUI* that is used hereinafter. Table 1 defines possible mappings of the *Widgets* that are used in the case study onto elements (*Class*es) of the MultimodalDesktopCUI of MARIA. These constitute the basis for the formulation of the *Mappings* of the *PAMTraM*.

Table 1. Excerpt of the possible mappings from DISL elements onto elements of the MARIA MultimodalDesktopCUI

		DISL – AUI		
		textfield	command	choicegroup
MARIA – MultimodalDesktopCUI	audio	x		
	button		x	
	check_box			x
	choice_element	x		
	drop_down_list			x
	image	x		
	link		x	
	list_box	x		x
	radio_button			x
	text	x		
	text_area	x		
	text_field	x		
	video	x		
	vocal_activator		x	
	vocal_output	x		
	vocal_selection			x

Table 1 shows (in excerpts) the occurrence of ambiguities, for example for the possibilities to define an output variable (*textfield*). For further steps, these (ambiguous) mappings were stored in an instance of the PAMTraM that was presented in Section 3.

Fig. 8 displays the resulting code for the definition of the mappings that has been created with the textual model editor (comp. Section 3.3). By means of the editor capabilities (e.g. the content assistant), a UI designer can define the mapping model without deep knowledge about the underlying meta-model of the PAMTraM.

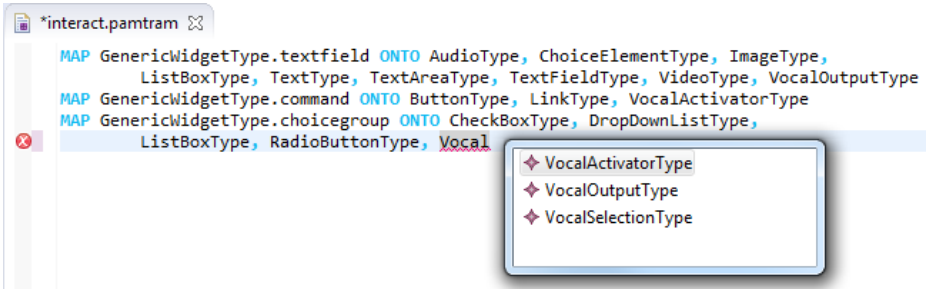

Fig. 8. Definition of the mappings with the textual editor

An interactive transformation was used in this case study to solve the occurring ambiguities. For each AUI element, it presents all possible target elements to the user in a graphical interface. The user can then choose the mapping to be used in the transformation by selecting one or more target elements. This transformation was implemented with help of the Epsilon transformation framework [20] and EMF.

Fig. 9 shows the prompt for user interaction during the initial transformation of the *textfield*-Widget that displays the fill level. The possible target elements correspond to those that were specified in Table 1 and defined in the ambiguous mapping model. It should be noted that the alternative(s) that was/were chosen by the user are stored as *TransformationMappings* as described in Section 3.2.

Fig. 10 shows the corresponding part of an instance of the PAMTraM (cf. Fig. 4(b)): The upper half displays a stored transformation consisting of several transformation mappings for the various UI elements. In the bottom half, one of the stored transformation mappings is presented in more detail showing that the source model element “fill_level” as instance of the meta-model element “textfield” has been mapped to an instance of the target meta-model element “image”.

When re-transforming the UI model, e.g. in case of an iterative development, the user can select one of the stored, previously performed transformations. In this way, the corresponding unambiguous mappings are reused and a user interaction is only necessary for the transformation of elements that were added or changed during the last iteration step (comp. Sections 3.2 and 3.3).

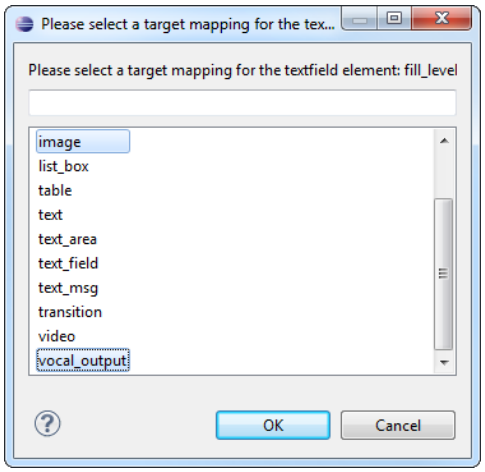

Fig. 9. User interaction for selecting the desired mapping

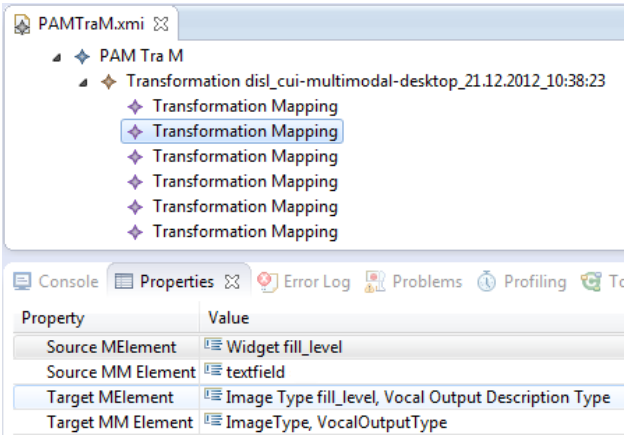

Fig. 10. Automatic storing of the selected mappings as TransformationMappings in the PAMTraM

As a designer can make different choices when selecting the mappings, different UIs can be the result of the transformation. Two possible user interfaces for the process plant that were created using the PAMTraM approach have already been shown in Fig. 7. They are the result of a model-to-text transformation applied to the CUI model resulting from the previous steps. In the UI on the left, an image was chosen to represent the fill level of the tank, while textual output was chosen on the right. Furthermore, a group of radio buttons respectively a drop-down list was chosen to select the tank, whose fill level should be shown. These decisions could e.g. be made based on platform restrictions as the second UI is more compact and thus fits better onto smaller screens.

5 Conclusions

The proposed *Persistent Ambiguous Mapping and Transformation Model* meets the requirements that were developed in Section 1. It implements (a) the definition of ambiguous mappings and (b) the storing of executed transformations in one combined meta-model. By making use of a declarative structure, domain experts without any special knowledge about the execution of a transformation should be enabled to define ambiguous mappings. Indeed, this has yet to be proven by an empirical evaluation In order to hide the structure of the underlying meta-model from the domain expert, a textual editor has been created that supports the user in the definition of mappings. However, a graphical editor, like the one MARIAE offers, could be useful to handle the often very large models and to support the user's actions. Furthermore, it has to be elaborated if mappings between groups of elements (instead of single elements) need to be supported and how this support could be realized.

The structure of the defined meta-model allows the integration of different methods to resolve ambiguities: In the case study, we have demonstrated an interactive transformation. Furthermore, the utilization of the context of use or an integration of statistical algorithms (e.g. the evaluation of previous user choices based on stored transformations) is possible.

PAMTraM supports the iterative development of user interfaces by storing the chosen, unambiguous mappings. By reusing these stored mappings, the selection mechanism (e.g. a user interaction) only has to be applied to elements that have changed since the last iteration cycle.

Moreover, as PAMTraM uses universal structures, it is not limited to transformations between AUI and CUI (as demonstrated in the case study) but can be applied to all model(-to-model) transformations within the CRF. This extends to a transformation that makes only use of unambiguous mappings. Thus, a continuous methodology including a continuous tool support can be applied. However, as PAMTraM makes use of models respectively meta-models of the source and the target language, it cannot be applied to languages that are not represented by a formal meta-model but e.g. by a description in natural language.

The transformation execution that evaluates the ambiguous mappings defined in the PAMTraM and stores the chosen, unambiguous mappings has to be implemented for every language pair. However, the achieved separation between the domain knowledge stored in the PAMTraM and the actual transformation reduces the necessary effort for this implementation: The transformation can be implemented by an expert with in-depth knowledge of the creation of transformations who does not need to be a domain expert. Furthermore, the once implemented transformation can be used for every user interface that is described using the same UI description language pair.

Future work will cover the integration of the context of use into the decision process to reduce ambiguities. Therefore, possibilities to define conditions have to be created. Besides that, the user has to be able to attach these conditions to ambiguous mappings. By combining this with a subsequent user interaction, it can be ensured that the user may only choose between alternatives that are reasonable (for a certain

context of use and a certain user interface). Compared to the exclusive user interaction realized in Section 4 the user is relieved of useless target elements.

Finally, an extensive empirical study has to be conducted in order to prove and classify the advantage of our approach. Therefore, the specification of transformations for various pairs of UI description languages as well as the development of specific user interfaces by means of the specified transformation has to be evaluated and compared to existing approaches concerning its effectiveness and usability.

References

1. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. In: Interacting with Computers, vol. 15, pp. 289–308 (2003)
2. Vanderdonckt, J.: A MDA-compliant environment for developing user interfaces of information systems. In: Advanced Information Systems Engineering, pp. 16–31 (2005)
3. Gruhn, V., Pieper, D., Röttgers, C.: MDA - Effektives Softwareengineering mit UML2 und Eclipse. Springer, Heidelberg (2006)
4. Puerta, A., Eisenstein, J.: Towards a general computational framework for model-based interface development systems. In: Knowledge-Based Systems, vol. 12, no. 8, pp. 433–442 (1999)
5. The Eclipse Foundation: ATL – a model transformation technology, http://www.eclipse.org/atl/
6. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.1, http://www.omg.org/spec/QVT/1.1/ (2011)
7. W3C: XSL Transformations (XSLT), Version 1.0, http://www.w3.org/TR/xslt (1999)
8. Hager, H., Hennig, S., Seißler, M., Braune, A.: Modelltransformationen in nutzerzentrierten Entwurfsprozessen der Automation. i-com, vol. 10(3), pp. 19–25 (2011)
9. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems, pp. 440–453. Springer, Heidelberg (2006)
10. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: a language supporting multi-path development of user interfaces. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) Engineering Human Computer Interaction and Interactive Systems. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005)
11. Paternò, F., Santoro, C., Spano, L. D.: MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. In: ACM Transactions on Computer-Human Interaction, vol. 16(4), pp. 1–30 (2009)
12. Lisai, M., Paternò, F., Santoro, C., Spano, L. D.: Supporting transformations across user interface descriptions at various abstraction levels. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) Proceedings of the 13th IFIP TC 13 international conference on Human-computer interaction - Volume Part IV. LNCS, vol. 6949, pp. 608–611. Springer, Heidelberg (2011)

13. Aquino, N., Vanderdonckt, J., Pastor, O.: Transformation templates: adding flexibility to model-driven engineering of user interfaces. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1195–1202 (2010)
14. Schäfer, R.: Model-based Development of Multimodal and Multi-device User Interfaces in Context-aware Environments. In: C-LAB Publication. Dissertation, Universität Paderborn (2007)
15. Hennig, S., Braune, A.: Sustainable Visualization Solutions in Industrial Automation with Movisa – a Case Study. In: 9th IEEE International Conference on Industrial Informatics, pp. 634–639 (2011)
16. Hennig, S., Van den Bergh, J., Luyten, K., Braune, A.: User Driven Evolution of User Interface Models – The FLEPR Approach. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) Proceedings of the 13th IFIP TC 13 international conference on Human-computer interaction - Volume Part III. LNCS, vol. 6948, pp. 610–627. Springer, Heidelberg (2011)
17. Eclipse Modeling Framework Project, http://www.eclipse.org/modeling/emf/
18. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., von Stockfleth, B.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
19. DevBoost: EMFText, http://www.emftext.org
20. Epsilon, http://www.eclipse.org/epsilon/