Defining Domain Specific Transformations in Human-Computer Interfaces Development

Jean-Sébastien Sottet and Alain Vagner
Public Research Center Henri Tudor, 29 Avenue John F. Kennedy, Luxembourg, Luxembourg

Keywords: Human-Computer Interaction, Model-Driven Development, Model Transformation, Domain Specific Transformation Languages.

Abstract: Early model-based approaches for Human-Computer Interaction (HCI) clearly depicted models and frameworks for generating User Interfaces (UI) but considered model transformations as black-boxes. In the 2000's, these approaches were criticized due to the poor quality of the produced UI. One of the main reasons of this poor quality can be easily observed in state of the art UI transformations: they are the heart of designers' know-how but are maintained by a minority of specialists. Meanwhile, mainstream UI design methods have shown a growing number of heterogeneous stakeholders that collaborate to produce modern and qualitative UI. We claim that these stakeholders must comprehend and interact with transformations and thus we need to make the transformation language affordable to these stakeholders. Indeed, such a simplification should hide transformations complexity and burden for any stakeholder, finally focusing on a specific part of the design domain: a Domain Specific Language (DSL) for transformations or Domain Specific Transformation Language (DSTL). We provide in this paper a method and a supporting tool for systematizing and finally executing DSTL for model-driven UI development. We depict that framework on a proof of concept implementation for an HCI-specific stakeholder: the usability expert.

1 INTRODUCTION

In the end of the 80's, the development of Human-Computer Interfaces (HCI) was not considered as a particular task: it was done by any of the application developers. The complexity and effort needed for building User Interfaces (UI), like any other part of information systems, were considerably growing up during the last decades. In order to solve this complexity issue and reduce the consequent cost, some tried to generate the user interfaces. These approaches considered that an important part of the UI has already been made during the data and functional architecture design. Thus, early academic work in this area was focused on model-based approaches and frameworks (Szekely et al., 1992; Vanderdonckt, 1995; Puerta, 1997) for automatizing the creation of UI.

The focus of these first model-based frameworks was the establishment of (meta)models (e.g., expressive power, defining the concerns, etc.) whereas transformations were seen as black boxes. More recently, Model Driven Engineering (MDE) has influenced the model-based HCI approaches. Transformations have been more explicitly defined and specifically instrumented by tools such as ATL (Botterweck, 2011; Sottet et al., 2007), EMFText and Kermeta (Beaudoux et al., 2010) or graph transformations (Stanciulescu et al., 2005). They are thus an important artefact to be considered in the automation of HCI development.

It is admitted that recent UI design projects (as in any software part) involve collaboration between various stakeholders, e.g., graphical designers, usability specialists, analysts, etc. However several authors claimed that using traditional MDE approaches is a specialist work (e.g., transformation engineers) that requires an important training. For instance in (Coutaz, 2010; Panach et al., 2011), the authors consider that traditional transformation languages require specific skills and could not be written by any HCI stakeholder.

As a result, model transformations are very complex to understand for many stakeholders. They could not use or modify transformations in the same way they impact models. For instance, a graphical designer is traditionally using a mock-up tool, such as balsamiq[1] which is a domain specific modelling en-

[1] http://balsamiq.com/

Sottet J. and Vagner A..
Defining Domain Specific Transformations in Human-Computer Interfaces Development.
DOI: 10.5220/0004699702460253
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 246-253
ISBN: 978-989-758-007-9

vironment. Thus, it seems important that designers could also drive some of the transformation aspects related to graphical design if we want to build qualitative HCI.

In this paper we strive for a better stakeholder involvement (not only the developers) in Model-Driven Development (MDD) transformation processes for HCI. To reach this objective we must first (section 2) find which stakeholders are concerned and understand the need for building a Domain Specific Transformation Language (DSTL). We propose such a language in section 3. Finally, in the section 4, we propose a systematisation of DSTL construction, making it more specific to the problem to be addressed. In this section we also propose an approach to execute the transformations expressed in our DSTL.

2 HCI DESIGN STAKEHOLDERS AND THE TRANSFORMATION PARADIGM

2.1 Model-Based Framework for User Interfaces Design

HCI research in Model-based UI elicited a common reference framework: CAMELEON (Calvary et al., 2003). This framework is made of several metamodels (see Figure 1, the 4 core metamodels) in which different aspects of HCI engineering are represented. The CAMELEON models correspond to specific domains such as graphical design (*Concrete UI model, CUI*), functional analysis (*Task Model*), domain analysis (*Domain Model*), interaction design (*Abstract UI model, AUI*), all of which lead to the production of the UI code (*Final UI, FUI*).

Each stakeholder has his own background knowledge and expertise domain: usability expert, system analyst, graphical designer, developer, etc. All these stakeholders may be disappointed by standard transformation languages, like ATL. In practice they are not directly involved in the definition of transformations but, at best, give advice on the transformation result. For instance, the usability expert would recommend to use alphabetically ordered lists for the selection of a contact in an address book.

In Figure 1, we show the envisioned distribution of stakeholders on the design process and their potential interactions on transformations and models. The main stakeholders are the following:

Designers: have an interest on the concrete UI: layout, graphical elements, mock-ups, etc. Thus, designer could have an impact on the transformations proposed between CUI and FUI.

Developers: classical development process could start after CUI specification. Developers have an impact on the FUI (e.g., code). On the contrary, in a transformation-based approach, developers will have to dig CUI to FUI transformations and tune, if necessary, both transformations and FUI.

Analysts: on the contrary to the developer role, analysts have an early impact on forward engineering based on task and domain models. In addition, they can have an impact on transformations from domain to AUI.

Usability Experts: have a potential impact on all transformation steps since usability principles can be diluted all along the forward engineering steps.

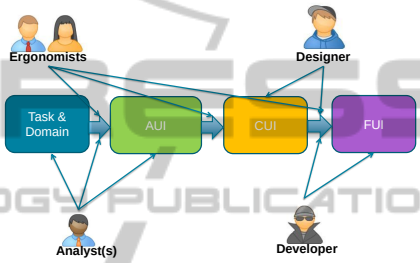

Figure 1: HCI Stakeholders interest on transformation processes in the CAMELEON Framework.

The CAMELEON framework depicts clearly (meta)models to be used at each step of the process (see Figure 1) and it is also quite straightforward to find which stakeholders will act on which specific aspects. We claim that every stakeholder could impact positively the transformation processes.

2.2 Transformation complexity

As mentioned in (Sendall and Kozaczynski, 2003) writing transformations requires a clear understanding of input and output semantics. By the way, who knows and understands the input and output domains? A relevant answer is certainly: the people who also participate in modelling activities.

The problem stands not only on (meta)model understanding but also on transformation languages complexity. Let's take for instance the *refImmediateComposite()* symbol, which refers to the immediate parent of an EMF composition in the ATL language (Jouault and Kurtev, 2005). It does not directly depend on the input or output models and metamodels (but on the supporting metametamodel). In addition, transformations embed some verbosity for technical (e.g., give name or id to modelling elements) or structural (e.g., graph search) handling of metamodels. As we can see on Listing 1, getting a simple label for a Concrete UI calls for a lot of "structural coding":

id, name and content of the label. Most of this code is certainly a burden for a non-technical stakeholder, who would rather simply say: "for this particular task I would like to produce a label which takes the task name". Indeed most of the stakeholders are usually reluctant to learn such "OCL-like" syntax.

Listing 1: Defining a label for a Graphical User Interface.

```
to out_label : CUI!Label (
name <-in_tsk.name, --copy the name of the task
id <-in_tsk.getID(), --get the id from task
value <- in_tsk.associations->first().
    ↪manipulatedConcepts.name, --set the
    ↪content of label
```

We claim that a collaborative building of transformations is the clue for more efficient MDD of HCI. Going in such direction could be an answer to the bad conclusion of the first generation of model-based UI approaches (Myers et al., 2000). In order to support this claim we need to tackle several drawbacks of classical transformation approaches:

- existing work rarely depicts what information is added in the transformation processes (e.g., which know-how to implement). Lot of heuristics are provided but not necessary implemented, e.g., UI adaptation (Florins and Vanderdonckt, 2004).
- transformations are either seen as black-box or expressed in a complex language for "transformation experts". Thus, domain experts are unable to express their "transformation heuristics".
- in order to facilitate the adoption and commitment/involvement of stakeholders, transformations have to be explained to each (collaborating) stakeholders.

In this paper, we concentrate our effort on involving usability experts in the design of transformations. Firstly because promising work tries to embed usability criteria (Sottet et al., 2007; García Frey et al., 2011) into transformations but without involving usability experts directly (e.g., defining criteria of transformations *a priori*). Secondly, because usability experts are certainly the less technical stakeholders. Their tasks are mainly dedicated to prototype (or end-product) usability assessments. We are convinced that they could have an interesting impact in early design phases and avoid cycles of redesign by re-writing the transformations.

It seems necessary, in order to involve domain specialists such as the usability expert to define a Domain Specific Transformation Language (DSTL). It seems to be a solution to cope with the complexity of standard transformation languages.

3 DOMAIN SPECIFIC TRANSFORMATION LANGUAGE

In the previous section, we have justified the need for defining a dedicated language, that should help stakeholders to participate in the transformation design process. Notably we want them to understand the transformation effects and involve them in the transformation design. The idea is not to create a fully functional language for defining transformations but to provide handlers (i.e., specific parts of the transformation that drive the transformation heuristics) to specific stakeholders in order to give them control only on aspects, that would be of interest to them.

We previously established that the usability expert is a key stakeholder of the AUI to CUI transformation (see Figure 1). The transformation vocabulary must be adapted to the usability expert and the structural aspects of transformations (syntactic overload) must be hidden. Most of the usability experts are more used to unstructured languages. In order to design such a language, we must address the following points:

- define the input vocabulary: usability experts understand the "interaction type" notion also called canonical abstract types (Constantine, 2003) that are used as a basis for model-based approaches. For instance, all tasks with type "choice of an element amongst n" reflect an interaction pattern known by usability experts.
- define the output vocabulary: an usability expert understands what the elements of a final user interface are, for instance indexed lists, scroll lists, etc.
- composing the transformation: the transformation must be easy to write (e.g., using auto-completion) and have limited choices (e.g., list of elements). This can considerably shorten the learning and design time.

The process[2] depicted in figure 2 encompasses the first important step for transformation simplification by simplifying input and output vocabularies, in other words getting a subset of these metamodels. Our input metamodel is named TDA (for *Task Domain Abstraction*) and bring together the concepts coming from CAMELEON task metamodel, domain metamodel and interaction types (*AUITypes*). One has to identify which portion of this input metamodel is really impacting the transformation. In a previous project (Sottet and Vagner, 2013), we designed

[2]This process could be generalized to other kinds of metamodels and with multiple input/ouput metamodels.

many TDA to CUI transformations with ATL. We have identified that the "interaction type" is the most used type in the transformation filter (e.g., the discriminating factor). We have done the same for the CUI, defining which elements are of importance according to the study of existing transformations (see Reduced M2 TDA/CUI on Figure 2).

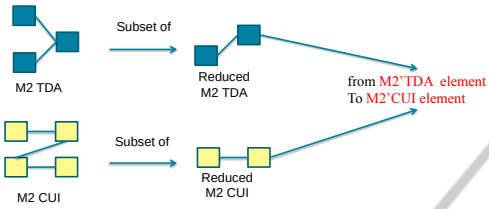

Figure 2: Process applied for specific transformation building for Task/Abstract UI model and CUI.

As an example, in natural language, usability experts usually provide some sentences (i.e., rules) like "It would be nice if all interactive lists could be displayed as indexed lists". Indeed, such form of sentence requires some advanced techniques in natural language processing in order to be able to interpret and execute this sentence. Moreover, some provided sentences can be contradictory, ambiguous, etc. In this paper we do not provide such advanced natural language processing tool but rather a controlled language for transformation rules specification by usability experts.

Such an approach (grammar definition and text editor) can be easily obtained by writing Xtext code[3] or using other DSL workbenches such as MetaEdit+[4]. One can summarise the basic specific HCI transformation rules (for obtaining simple form-based UI) as a set of 6 sentences/rules (see listing 2, one for each interaction type, and *APPENDIX 2* for an excerpt of the related Xtext grammar). In addition to this syntactical simplification and auto-completion, Xtext comes with interactive lists (see Figure 3) for limited choices which considerably reduces the error and time to write such transformations.

Listing 2: Set of rules for building form-based interfaces from TDA model to CUI model.

```
rule from Container interactive to Window with
    ↪All Concepts;
rule from Container notinteractive to Panel with
    ↪ No Concepts;
rule from Output interactive to Panel with All
    ↪Concepts;
rule from Input interactive to DataField;
rule from Choice 1/n interactive to
    ↪ListElementSelector with All Concepts;
rule from Choice n/n interactive to
    ↪ListElementSelector with All Concepts;
```

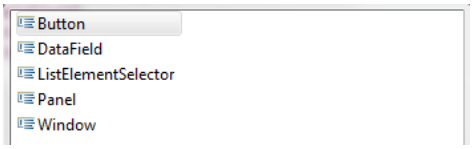

Figure 3: List of possible widgets (output metamodel) to bind with input Task elements.

We can extend this language to operators (more, less, equal) that can complete the first part (i.e., from) like for example: "I want all interactive lists containing more than 6 elements...". It makes the language more complete because some rules have a specific threshold which can be defined the usability specialist experience.

Obviously this solution considerably reduces the expression power of these transformations, but this helps the stakeholders to better grasp the effects/results of the transformation. In order to support the execution of these transformations, we have to complete them by generic code expressed in the underlying transformation language (see section 4).

4 SYSTEMATISATION AND EXECUTION

As mentioned in (Mernik et al., 2005) developing a DSL is not for free. A lot of inherent costs have been identified in the development of DSL like: (1) the training of the community and (2) the implementation of an interpreter/compiler for execution purposes. The cost of training the usability expert to our DSTL would certainly not exceed the training to standard transformation tools.

In order to simplify the development of such a transformation DSL we provide a way to systematise some of its construction aspects (See Section 4.1). It encompasses the automatic extraction of input and output patterns directly from metamodels. Finally we provide an Higher Order Transformation (Tisi et al., 2009) - HOT - (i.e., transformations which process other transformations) to support the execution of DSTL using existing transformation technologies (see Section 4.2).

4.1 Systematisation

Our DSTL, see Listing 2, is obtained by analysing both input and output metamodel elements (see Figure 2). This is a kind of HOT, which acts as

[3]http://www.eclipse.org/Xtext/
[4]http://www.metacase.com/mep/

a parametrization of a transformation (Tisi et al., 2009). As we already mentioned in the previous section, for the input metamodel (e.g. Task-Domain-Abstraction), we use the interaction type as the transformation rules key input (e.g., it is the discriminating factor that drives the transformations). The output metamodel contains a structured list of Concrete UI elements (e.g., set of description of widgets to be displayed such as a button, a list, a menu, etc.). These CUI elements are the choice that the usability expert will have to do when generating user interfaces.

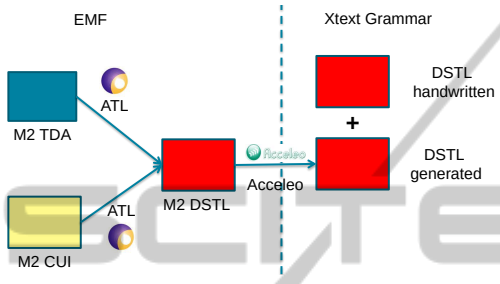

Figure 4: Automation of the creation of the DSTL grammar.

The idea is not write Xtext grammar code[5] but to partially generate the grammar from the two metamodels. The core idea (see figure 4) is to extract the information from each metamodel in order to obtain the input and output vocabularies for our DSTL (see Section 3).

To perform this task we firstly transform (using ATL transformations) the selected elements from two metamodels into a DSTL specification. This specification is an EMF model conform to our DSTL metamodel. From the TDA metamodel (see Listing 3), we select the "AUIType" (i.e., the canonical abstract type) as the DSTL input pattern based on the enumeration literals. From the CUI metamodel we extract each element (i.e., graphical widgets such as button, list, etc.). Once we got this DSTL (EMF) model built, we extract from it a partial Xtext grammar thanks to an Acceleo template. The two examples of generated excerpts are available on Figure 5 in the lower left and lower right boxes. These models are later combined with a generic core transformation grammar (see 5 middle part) into the final DSTL grammar.

Listing 3: Excerpt of ATL Code for exporting Task (TDA) model enumeration (EEnumliteral from EMF) of types into a DSTL input pattern.

```
rule TDAEnumerationTaskToInputElement {
  from
    inEnumLit : EMF!EEnumLiteral(inEnumLit.
        ↪refImmediateComposite().name='AUIType')
  to
    inPatt : DSTL!InputElement (
      name <- inEnumLit.name,
      elementConcerned <- #"Task")
}
```

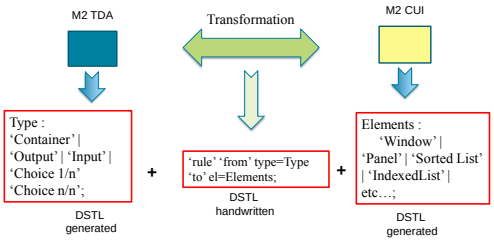

Figure 5: Automation of process from Figure 2.

4.2 Executing the Domain Specific Transformation Language

The DSTL specified in *APPENDIX 2* is a simplification of the ATL transformation "from TDA to CUI". If the expressive power of such a language is intentionally reduced, the technical and structural aspects remain hidden behind the scenes in ATL generic code. Indeed, we provide the stakeholders with a language that intentionally does not cover all the aspects of a transformation.

The execution semantics is provided by a general purpose transformation language (GPTL). Here we have chosen the ATL transformation language as target GPTL from our DSTL. In fact, the process that derives GPTL rules from DSTL rules is also a HOT. In practice, it consists in combining the DSTL model (established using the Xtext textual editor from the generated grammar) and existing core rules (i.e., written by transformation specialists).

We have chosen Acceleo[6], because (1) it is more efficient for writing code (i.e., Model to Text) than ATL itself and (2) it is easier to integrate in the template the existing technical ATL code[7]. We show an excerpt of the main rule template in the *APPENDIX 1* section. It provides, for a DSTL model, the ATL transformation input-pattern for the *AUItype* plus some hand-written filters in order to avoid rules overlapping (not accepted by the ATL compiler). For the out-pattern (i.e, after the *to*) another rule is making the link between DSTL and ATL-based CUI elements (*cOutPattern*).

Finally an ATL transformation is produced: ready to be compiled and executed. It integrates both previously implemented ATL transformations (made by a transformation expert) and ATL code produced from

[5]This is trivial in this example. However it may be very error-prone as one could forget to copy some elements.

[6]www.acceleo.org

[7]Generic structural manipulations of metamodel in ATL (e.g., helpers, call of libraries) are provided.

the HCI DSTL transformation. You can find an example in the listings 4 and 5.

Listing 4: Example of rule expressed according to the DSTL

```
rule from Choice 1/n interactive to
    ↪ListElementSelector with All Concepts;
```

Listing 5: Generated ATL code corresponding to the rule defined in listing 4.

```
rule selection_1_nToListElementSelector {
  from
    in_tsk : TDA!Task(
      in_tsk.auiType=#"selection_1_n"
      and in_tsk.isInteractivelyValid())
  to
    out : CUI!ListElementSelector(
      name <- in_tsk.name,
      id <-in_tsk.getID(),
      containedElements <- in_tsk.associations->
          ↪collect(e | thisModule.
          ↪DomainAttributeToInteractor(e.
          ↪manipulatedConcepts)))
}
```

5 RELATED WORK

Some related work in MDE and HCI engineering promotes the adaptation of transformation languages for specific purposes.

In (Agrawal et al., 2003), the authors propose a **UML visualisation** of graph transformations, that addresses people who are accustomed to UML syntax. In the same idea, UMLx (Willink, 2008), provides a **graphical representation** of transformations. Indeed, it is still not adapted to a wide variety of stakeholders (specifically in UI design), but promotes syntactical simplifications of transformations.

RubyTL (Cuadrado et al., 2006) is an implementation of a transformation language using a specific and extensible language. Extensibility is another way to provide more specific transformations (specialized regarding in/output metamodels). It provides the transformation designers with more specific rules.

In order to make transformations easier to non-transformation specialists, one can use natural language, giving "more sense" such as in (Störrle, 2013). For instance, in Natural MDA (Leal et al., 2006), the idea is to use a controlled natural language for expressing business rules. We can easily extend that to transformation rules. Natural MDA approach is similar to our transformation writing principles. However, here we propose mechanisms to semi-automatize the building process of such a language.

In (Sottet et al., 2009), authors propose the notion of trans-HCI: a specific UI on top of a transformation language. We implement that idea here using a textual editor for transformations. (Aquino et al., 2010) propose a framework for establishing transformation templates, that are easier to understand and manipulate than standard transformations. These transformations are edited thanks to the template and then executed within a standard transformation engine: proposing an upper-layer above transformations. If similar to our approach, transformation templates, do not hide some of technical aspects (e.g., necessity to provide types for values: real, string, boolean) which may confuse the stakeholders.

The paper of (Tisi et al., 2009) provides examples and a classification of Higher Order Transformation (HOT). We have been inspired by several kinds of HOT, notably the (Muliawan, 2008) approach. It is used to simplify and to make existing transformations more specific while keeping them executable. As in this paper, we transform input and output metamodels to pre-configure a transformation. Nevertheless, we go a step further defining a domain specific transformation language and thus we provide a specific syntax dedicated to particular stakeholders concerns. We use another kind of HOT in order to generate ATL code from our DSTL.

As a summary, related work address technologies to make transformations clearer, notably through adapted syntaxes (either textual or graphical) and by proposing extension and specialization mechanisms. We could indeed use these mechanisms to improve some parts of our work: graphical syntax, extension mechanisms, enhancing our HOT approach, etc. However, related approaches do not explicitly address human-centric aspects e.g., the stakeholders skills. For instance, the transformation template approach embed too much concerns into the transformation that could be a burden for stakeholders. On the contrary we deliberately give a simplified transformation model to avoid technical burden.

6 CONCLUSIONS

The integration of MDE in the HCI community is not uniform: although models found rapidly acceptance it is not the same for transformations. The poor quality of some generated UI and the opaque definition of transformations make them hardly accepted by some HCI stakeholders and researchers. People would prefer to use their standard design/implementation processes instead of something they do not understand clearly.

In this paper, we claim that stakeholders knowledge and know-how are first order citizen to build more efficient transformations for more qualitative UI. We have to aim not only at people that are trained to write transformations but to all kinds of HCI stakeholders. As a result, we will naturally refer to the definition of a stakeholder (i.e., domain) specific language for designing transformations. Domain specific transformations tend to be closer to the stakeholder vocabulary and to be enough focused on the transformation rationale. They thus provide targeted stakeholders with a support to express transformation heuristics and participate to the transformation design.

Our DSTL implementation (and related generation/extraction transformations) targets a specific stakeholder: the usability expert. To our experience, this is certainly the most representative case for HCI building: (1) the classic usability expert profile is oriented towards social science and (2) usability quality is crucial in HCI design. We provide the usability expert with a very simple language (controlled natural language), that helps in the generation of adapted (i.e., more usability regarding the situation) widgets.

We do not redesign a complete transformation language but rather provide some handlers for specific stakeholders. Such stakeholders act only on the specific transformation part while hiding the transformation inner mechanisms inside a traditional transformation in "back-end". Thus, there is still room for code transformation to be written by an expert (e.g., structural information).

On the HCI engineering side, our DSTL approach provides a way to: (1) capture specific know-how (this is usually done as assessment report after production of prototype) and make it executable, (2) involve more stakeholders in the MDE transformation design (3) by this involvement, provide a better acceptance of MDE approaches for HCI.

Technically, on the MDE side, we propose a generic approach for systematising the definition of DSTL based on HOT principles that: (1) provides a reduced set of input/output metamodels (that are specific to the stakeholder job), (2) derives (semi-automatically) a textual syntax to specify simplified transformation rules and (3) allows the execution of transformations.

This work is a first proof of concept in the HCI domain and more specifically addressing usability experts concerns. During this research process we did ask some usability experts their feeling about the standard transformation tools and they helped us designing the DSTL. However, we need more stakeholders to validate our approach regarding the HCI community. We try to present an initial instrumented methodology that can be generalised to other stakeholders and domains. Future work will concentrate on providing other DSL for other types of stakeholders and will try to address collaboration issues during transformation design.

ACKNOWLEDGEMENTS

This work has been partially supported by the Luxemburgish FNR MoDEL project (C12/IS/3977071).

REFERENCES

Agrawal, A., Karsai, G., and Shi, F. (2003). A uml-based graph transformation approach for implementing domain-specific model transformations. *International Journal on Software and Systems Modeling*.

Aquino, N., Vanderdonckt, J., and Pastor, O. (2010). Transformation templates: adding flexibility to model-driven engineering of user interfaces. In *Proceedings of the 2010 ACM Symposium on Applied Computing*.

Beaudoux, O., Blouin, A., and Jézéquel, J.-M. (2010). Using model driven engineering technologies for building authoring applications. In *DocEng '10: Proc. of the 2010 ACM symposium on Document engineering*.

Botterweck, G. (2011). Multi front-end engineering. In *Model-Driven Development of Advanced User Interfaces*, pages 27–42. Springer.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308.

Constantine, L. (2003). Canonical abstract prototypes for abstract visual and interaction design. In *Interactive Systems. Design, Specification, and Verification*.

Coutaz, J. (2010). User interface plasticity: Model driven engineering to the limit! In *Int. Conf. on Engineering Interactive Computing Systems (EICS 2010). Keynote paper.*

Cuadrado, J. S., Molina, J. G., and Tortosa, M. M. (2006). Rubytl: A practical, extensible transformation language. In Rensink, A. and Warmer, J., editors, *Model Driven Architecture Foundations and Applications*.

Florins, M. and Vanderdonckt, J. (2004). Graceful degradation of user interfaces as a design method for multiplatform systems. In *IUI*, volume 94, pages 13–16.

García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. (2011). QUIMERA: a quality metamodel to improve design rationale. In *Proc. of the third ACM SIGCHI Symp. on Engineering Interactive Computing Systems (EICS)*.

Jouault, F. and Kurtev, I. (2005). Transforming models with atl. In *Proceedings of the 2005 international conference on Satellite Events at the MoD-*

ELS, MoDELS'05, pages 128–138, Berlin, Heidelberg. Springer-Verlag.

Leal, L. N., Pires, P. F., Campos, M. L. M., and Delicato, F. C. (2006). Natural mda: controlled natural language for action specifications on model driven development. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Muliawan, O. (2008). Extending a model transformation language using higher order transformations. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 315–318. IEEE.

Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28.

Panach, I., Aquino, N., and Pastor, O. (2011). A model for dealing with usability in a holistic mdd method. *User Interface Description Language (UIDL)*.

Puerta, A. R. (1997). A model-based interface development environment. *Software, IEEE*, 14(4):40–47.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45.

Sottet, J.-S., Calvary, G., Coutaz, J., and Favre, J.-M. (2007). A model-driven engineering approach for the usability of plastic user interfaces. In *Engineering Interactive Systems*, pages 140–157. Springer.

Sottet, J.-S., Calvary, G., Favre, J.-M., and Coutaz, J. (2009). Megamodeling and metamodel-driven engineering for plastic user interfaces: mega-ui. In *Human-Centered Software Engineering*, pages 173–200. Springer London.

Sottet, J.-S. and Vagner, A. (2013). GENIUS: Automatically generating usable user interfaces. Technical report, PRC Henri Tudor, Luxembourg.

Stanciulescu, A., Limbourg, Q., Vanderdonckt, J., Michotte, B., and Montero, F. (2005). A transformational approach for multimodal web user interfaces based on usixml. In *Proc. of the 7th inter. conference on Multimodal interfaces*, ICMI '05, pages 259–266.

Störrle, H. (2013). Making sense to modelers: Presenting uml class model differences in prose. In *1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013)*.

Szekely, P., Luo, P., and Neches, R. (1992). Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *Proc. of the SIGCHI conf. on Human factors in computing systems*.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the use of higher-order model transformations. In *Model Driven Architecture-Foundations and Applications*, pages 18–33. Springer.

Vanderdonckt, J. (1995). Knowledge-based systems for automated user interface generation: the trident experience. In *Proceedings of the CHI*, volume 95. Citeseer.

Willink, E. D. (2008). On challenges for a graphical transformation notation and the umlx approach. *Electronic Notes in Theoretical Computer Science*, 211:171–179.

APPENDIXES

1. Acceleo template for ATL code generation from a DSTL model.

```
[template private gRule(rule : Rule)]
 rule [rule.auitype.concat('To').concat(nameString
     ↪(rule.uielement))/] {
    from
      in_tsk : TDA!Task([if (rule.auitype.
          ↪equalsIgnoreCase('Root'))]
            (maintsk.superOperator.oclIsUndefined
                ↪()) [else]
  in_tsk.auiType=#"[rule.auitype/]"
            [if (rule.interactivelyValid.
                ↪equalsIgnoreCase('interactive'))
                ↪]
  and in_tsk.isInteractivelyValid()
            [/if]
            [if (rule.interactivelyValid.
                ↪equalsIgnoreCase('notinteractive
                ↪'))]
  and in_tsk.type=#"abstract" and in_tsk.
      ↪isNotLeafNorRoot()
            [/if]
          [/if])
    to
       [rule.uielement.cOutPattern()/]
 }
[/template]
```

2. Excerpt of the DSTL Xtext grammar.

```
Rule:
    'rule from' auitype=Type ('and'
        ↪interactivelyValid=Valid )? 'to'
        ↪uielement=UIElement ';';
Type :
    'Choice n/n' | 'Choice 1/n' | 'Container' | '
        ↪Input' | 'Output' | 'Command' | '
        ↪Navigation' |'Root';
UIElement :
    Window | ListElementSelector | Button | Panel
        ↪| DataField | ImageField | TextField |
        ↪TextArea;
Window :
    'Window' label+=Label? 'with'
        ↪containedElements+=Concept;
Panel :
    'Panel' label+=Label? 'with' containedElements
        ↪+=Concept;
ListElementSelector :
    'ListElementSelector' label=Label? ('with'
        ↪containedElements+=Concept)?;
Concept :
    'All Concepts' | 'No Concepts' | selection+=
        ↪SelectedConcept*;
```