

A Multi-level Approach for Model-Based User Interface Development

Björn Benner

Information Systems and Enterprise Modeling,
University of Duisburg-Essen, Essen, Germany
Email: bjoern.benner@uni-due.de

Abstract—Multi-level modeling is considered to provide advancements over the traditional modeling approaches. It is applicable, among others, to scenarios where more than two classification levels are required and a rigid dichotomy between classes and objects needs to be relaxed. As the Model-Based User Interface Development (MBUID) inherently encompasses more than two classification levels, there are challenges which cannot be accounted for with the traditional approaches (e.g., the propagation of changes or the extensibility of modeling languages). Therefore, it seems promising to apply multi-level modeling for overcoming these challenges. In this paper, we show the applicability of the multi-level modeling language FMML^x and the language execution engine XModeler for the MBUID-field. As a proof of concept, a corresponding prototype based on FMML^x and XModeler is presented which illustrates the benefits of multi-level modeling.

Index Terms—Multi-level Modeling, MBUID, FMML^x, XModeler

I. INTRODUCTION

Multi-level modeling aims at providing benefits over traditional conceptual modeling, e.g., by supporting an arbitrary number of classification levels, increased capabilities of reuse and increased capabilities of integration [1]. Recently, the interest in multi-level modeling has increased [1]–[4] and various multi-level modeling approaches have been successfully applied in different areas, e.g., the management of IT-infrastructure [5] or the process improvement in the automotive domain [6].

Taking into account, on the one hand, characteristics of traditional conceptual modeling, on the other hand, unfulfilled requirements (e.g., the propagation of changes or the extensibility of modeling languages), an additional area which seems to be promising as a further field for the application of multi-level modeling is the *Model-Based User Interface Development* (MBUID) [7], [8]. MBUID utilizes different types of models (e.g., *abstract user interface*, *concrete user interface* or *final user interface*) for the development of user interfaces (UIs) and aims at reducing the amount of time consumed by the manual UI creation: about 48% of the source code and about 50% of the implementation time is spent on the UI development [9].

MBUID utilizes models on different levels of abstraction for developing the UI, thus, it would benefit from an arbitrary number of classification levels. As those models are interdependent, changes in one model have to be synchronized

into the dependent models. Therefore, the capabilities for integration can support the mutual synchronization of models, thus, support the overall integrity. Furthermore, MBUID would benefit from the increased capabilities of reuse, as it is not desirable to create each UI from scratch.

Therefore, our goal is to investigate the applicability of the multi-level paradigm for the model-based development of UIs. To the best of our knowledge, only one approach exists that applied multi-level modeling for model-based development of UIs [10]. However, this approach targets the development of concrete syntaxes and is limited by a predefined set of visualization types. Therefore, this approach investigates only a small subset of the MBUID area. In contrast, we do not only focus on the design of UIs, but consider also the other aspects of the UI's entire lifecycle, i.e., its execution, its modification as well as its adaption. Therefore, we investigated not only the applicability of multi-level modeling but also the applicability of multi-level programming for MBUID approaches. Henceforth, the combination of multi-level modeling with multi-level programming is understood as the multi-level paradigm. Accordingly, the traditional paradigm is understood as the traditional conceptual modeling in combination with prevalent programming languages.

In order to accomplish the above-mentioned goal, the remaining of the paper is structured as follows: First, generic requirements towards the model-based development of UIs are identified (Section II). Afterwards, the approaches based on the traditional paradigm are evaluated against those requirements (Section III). Following, a multi-level approach is selected and the suitability of it for satisfying these requirements is discussed (Section IV). Afterwards, a corresponding prototype is developed (Section V) and an evaluation takes place (Section VI). Finally, conclusions and an outlook on future work are given (Section VII).

II. REQUIREMENTS TOWARDS MBUID APPROACHES

Requirements towards MBUID approaches can be categorized into core and additional requirements. Core requirements focus on the core functionalities of any MBUID approach, e.g., the possibility to design models for the UI creation. As all MBUID approaches deliver those, following, we focus on the additional requirements, which target advanced features enabling effective and efficient design of UIs. Following, the

most relevant additional requirements, taking into account the goal of this paper, are presented.

R1 Foster Productivity: Considering all of the aspects that should be accounted for while building a UI model (e.g., widgets, colors or arrangement), building models is considered difficult [11]. Especially if the modeling language only provides primitive (generic) modeling concepts, the creation of such rich models is a cumbersome endeavor as everything has to be recreated from scratch. Therefore, in order to support productivity of modeling, a modeling language should provide semantically rich concepts which account for specific characteristics of user interfaces [7], [8].

R2 Foster Range of Reuse: Beyond this, MBUID approaches should be applicable in a wide range of different scenarios (economies of scale) [12]. Therefore, MBUID approaches should provide generic concepts which can be reused in different situations. However, this claim stays in opposition to **R1**: In order to foster productivity, an approach requires semantically rich concepts. However, such specific concepts can presumably not be reused in a wide range of scenarios. In contrast, in order to be reusable in different scenarios, an approach has to provide generic concepts, which hampers the productivity.

R3 Transparent Propagation of Modifications: Due to the omnipresence of change, developed UIs have to be adapted continuously to changed requirements during their lifecycle. For this reason, it is necessary to modify the corresponding models in order to satisfy those changed requirements. Subsequently, those changes should be reflected in the corresponding source code. However, it should not be necessary to recompile the source code, but the UI should be adapted during runtime. Therefore, it is desirable to have a transparent propagation of change which applies the changes in the model in an automatic manner to the source code and maintains the overall integrity [8], [13].

R4 Extensibility of Language and Adaptability of Tool: In literature, the omnipresence of change is mostly discussed in terms of its implications for the created user interface, e.g., [13]. However, literature mostly neglects changes which go beyond the *language application* and require to adapt the *language specification*. For example, it might be desirable to add new concepts to the language specification by refining existing ones (e.g., a refined Button which adheres to a corporate design). In that context, it should not only be possible to enhance the language specification on demand with a new concept, but also to adapt the corresponding tool.

III. SHORTCOMINGS OF THE TRADITIONAL PARADIGM

We argue that MBUID approaches which rely on the traditional paradigm cannot entirely satisfy the additional requirements due to the limitations imposed by the traditional conceptual modeling and corresponding tools implemented using prevalent object-oriented programming languages.

In order to illustrate the implications of the prevalent paradigm on current MBUID approaches, it is discussed using a prototypical MBUID approach named Cameleon Reference

Framework (CRF) [14]. This framework presents an overall structure for MBUID approaches which is on the one hand valid for the majority of existing approaches [14] and on the other hand used as a foundation for new MBUID approaches [15]–[21].

The CRF constitutes a four-layered framework for structuring the development and the adaption of UIs [14]. The four layers of the CRF are the *Final UI* (i.e., the actual UI implemented in a programming language), the *Concrete Interface* (i.e., a non-executable platform- and interactor-specific model of the UI), the *Abstract Interface* (i.e., an abstract specification of the UI) and *Concepts and Task Models* (i.e., the specification of relevant tasks and domain concepts). The layers are based on three types of so-called ‘ontological models’ [14]: domain-models “support the description of the concepts and user tasks” [14, p. 294], context-models “characterize the context of use in terms of user, platform and environment” [14, p. 294] and adaption-models “specify both the reaction in case of change of context of use and a smoothly way for commuting” [14, p. 294].

In order to transition from one layer to another, model transformation is applied [16]. Model transformation is defined as the application of transformation rules in order to translate a model into either text or another model (which adheres to another metamodel) [22]. The translation from *Concepts and Task Models* to *Abstract Interface* constitutes a model transformation, as both models have different intentions, thus, they adhere to different metamodels.

In contrast, the transformation from the *Abstract Interface* to *Concrete Interface* is unsatisfactory. As an *Abstract Interface* is interactor-independent and the *Concrete Interface* is interactor-dependent, several *Concrete Interfaces* can be transformed or derived from one *Abstract Interface* [14]. Therefore, a number of *Concrete Interfaces* can be characterized or rather *classified* by one *Abstract Interface*. Thus, the *Abstract Interface* rather represents a kind of metamodel for the corresponding *Concrete Interfaces*. Therefore, this relation can be regarded as a model transformation, because both models have different metamodels. However, an instantiation-relation between *Concrete Interface* and *Abstract Interface* would be more appropriate because an *Abstract Interface* can be considered as a class for a number of *Concrete Interfaces*.

The transformation from the *Concrete Interface* to the *Final UI* is also a model transformation, as source code (i.e., text) is generated based on a model. However, the transformation itself is not satisfying, as it is afflicted by well-known synchronization problems. If there are changes to the model, those changes have to be transferred to the source code and vice versa.

The lack of an instantiation-relation and the necessity of code generation is not caused by a misconception of the CRF itself. However, they are a consequence of the traditional conceptual modeling in combination with the predominant programming languages.

The “prevalent programming languages are restricted to the dichotomy of objects and classes” [23, p. 32]. Therefore, there is only one classification level, i.e., an entity is either a class

or a corresponding instance. This dichotomy in programming languages influences the tool support. Software tools which support the development of MBUID models, implement the required modeling concepts in a programming language. Thus, the modeling concepts are on the class-level. Therefore, the created models are instances of the modeling concepts, i.e., in the realm of those software tools, the created MBUID models are on the object level [23]. However, those MBUID models are models of the generated UI, thus, they are conceptually (at least) on the level of classes. As consequence, it is necessary to generate corresponding classes from the instances in order to allow a further instantiation. Nevertheless, the usage of corresponding software tools implies a mismatch between the conceptual and the technical level of a MBUID model. Thus, the current MBUID modeling tools cannot adequately support the MBUID approaches due to the limitations imposed by the prevalent programming languages.

Due to these limitations, current MBUID approaches are not capable of propagating modifications transparently (**R3**). During design time, the existing models are transformed into source code, which can be executed afterwards. If there are changes to the model, the corresponding source code is simply recreated, i.e., these are changes on the class level. During runtime, it is not possible to recreate the source code in that way, but it is necessary to adapt the existing objects. For this reason, the models are interpreted first, followed by invoking adaption mechanisms on the objects. Such adaption mechanisms are hazardous, because an inappropriate adaption might lead to runtime failures. As traditional conceptual modeling and predominant programming languages are not tightly integrated, it is necessary to implement complex transformation mechanisms in order to connect both. Therefore, the combination of traditional conceptual modeling and predominant programming languages is not capable of propagating modifications transparently.

Furthermore, the application of traditional conceptual modeling implies a conflict between productivity (**R1**) and range of reuse (**R2**) as discussed in the previous section. Traditional conceptual modeling does only allow satisfying one demand while neglecting the other [23]: An approach can contain semantically rich concepts (in the sense of domain specific modeling languages), thus, foster the productivity. However, such concepts can presumably not be applied in a wide range of scenarios. In contrast, an approach can contain generic concepts (in the sense of general purpose modeling languages), which can be reused in several scenarios. However, the concepts do not provide semantic richness, thus, the productivity gained by its usage is rather low. Hence, the traditional paradigm is only capable of satisfying either **R1** or **R2** at a time.

Beyond this, the traditional paradigm limits the extensibility of MBUID approaches (**R4**). It enforces a strict separation of language specification and language application. Hence, adding further language concepts requires modifying the language specification, first, and adapting the corresponding tool afterwards. After recompilation, the enhancements can be used

in the tool. Thus, the traditional paradigm does not allow enhancing an approach during runtime.

In consequence, the prevalent paradigm consisting of traditional conceptual modeling and predominant programming languages is not able to satisfy the identified requirements. Therefore, in the next section, we investigate an alternative paradigm comprising a multi-level modeling and programming approach.

IV. MULTI-LEVEL MODELING AND LANGUAGE EXECUTION ENGINE

In order to satisfy the identified requirements, an approach is required which supports multiple levels of classification as well as a corresponding execution engine. Although different multi-level modeling approaches (e.g., [24], [25]) as well as execution engines (e.g., [26], [27]) exist, to the best of our knowledge there is no other pair of multi-level modeling approach and corresponding language execution engine besides FMML^x (*Flexible Meta-Modeling and Execution Language*) [1] and XModeler [28], [29] which allows modelling as well as executing behavior. Therefore, it becomes our approach of choice.

FMML^x builds on XCore (the XModeler metamodel), thus, it incorporates XCore’s recursive, self-reflexive architecture. This architecture is inspired by Hofstadter’s praise of recursion called “golden braid” [30]. In this architecture, the instance-of-relation between class and object is enriched by a specialization-relation. Thus, the concept *class* is a specialization of the concept *object*. Therefore, a class can be instantiated to another class, i.e., it enables a recursive language architecture. Hence, FMML^x allows for an arbitrary number of classification levels. Furthermore, as all classifications levels are in the same model, there is no strict separation of language levels. It is of note that FMML^x also provides the concept *inheritance* in addition to the *instantiation*.

Moreover, FMML^x contains the concept of *intrinsicness* which allows a deferred instantiation of attributes, operations and associations. *Intrinsicness* assigns an *instantiation level* to those entities [1], i.e., they are instantiated on the corresponding *classification level*. Therefore, the dichotomy between instantiation and specialization is relaxed, as attributes, operations and associations can be instantiated as well as inherited from one classification level to the next lower level. A similar concept is the *potency* of *Deep Instantiation* [31].

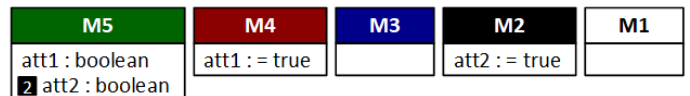


Fig. 1. Notation FMML^x

The notation of FMML^x is similar to the notation of UML class diagrams (cf. Figure 1). Beyond this, the classification level is indicated by the background-color of the name-box: M₅ is visualized by a green background of the name-box, M₄ by red, M₃ by blue, M₂ by black and M₁ by white (cf.

Figure 1). *Intrinsicness* is presented as a black box with a white number (i.e., the *instantiation level*) in front of the feature’s name.

XModeler (also known as XMF) is a language execution engine [1], [28], [29]. As explained, the XModeler’s metamodel is XCore, thus, XModeler allows for an arbitrary number of classification levels as well.

XModeler does not only support the creation of models on several classification levels, but it also allows for model execution. Each entity of the system is not only a model but also source code. A modification of the model implies a modification of the source code, and vice versa. Due to this shared representation, each entity can be executed. Thereby, execution is not limited to querying model elements, but each entity can be enhanced by algorithms for an execution in the sense of a complete programming language [28], [29]. For this reason, XModeler provides XOCL (eXecutable OCL) which is an enhanced version of OCL (Object Constraint Languages) that comprises imperative features [28].

The usage of FMML^x in combination with XModeler’s capabilities for modeling and programming constitutes an integrated environment. The UI design and the UI execution is performed in the same system. Hence, there are no issues related to the synchronization of a design and an execution environment. Therefore, a UI can directly be executed without further effort after designing or modifying it.

Furthermore, the integrated environment can also serve as a live-editor. This means, that such a multi-level approach is not limited to modifying a model and executing it afterwards, but it is capable of modifying a model and adapting corresponding UIs during runtime automatically. Those capabilities enable a tight interweaving of the development and the test of a UI.

V. A PROTOTYPICAL MULTI-LEVEL MBUID APPROACH

Following, a prototype is presented which constitutes a rudimentary multi-level MBUID approach. As already discussed, the selected multi-level paradigm is the configuration of FMML^x and XModeler. This prototype implements selected aspects only in order to illustrate the capabilities of a multi-level approach for MBUID. Therefore, this approach neglects features of prevalent approaches (e.g., the consideration of task models or an explicit visual notation). However, this approach is intended to illustrate a potential solution for the discussed requirements.

As the application of FMML^x and XModeler constitutes an integrated environment, it is not sufficient to focus only on the UI design, but it is necessary to consider also the execution. Therefore, an adequate language architecture has been developed which supports both (Figure 2).

The language architecture is inspired by the MVC-Pattern [32]. In this pattern, a *Controller* is in charge of synchronizing a *Model* with a *View*. Consequently, if there are two *Views* for one *Model*, there are also two *Controllers*. As those controllers relate to the same model, there is a high chance, that both controllers access the same aspect, thus, there is presumably redundant source code.

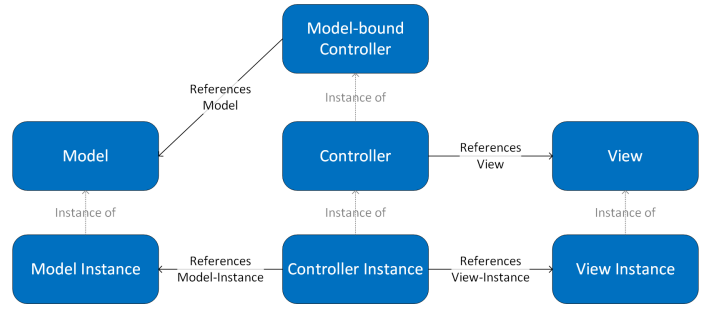


Fig. 2. Language Architecture

The language architecture presented in Figure 2, utilizes the multi-level modeling in order to overcome this threat of redundancy. The concept *Controller* is divided into two concepts: the *Model-bound Controller* and the *Controller*. While a *Model-bound Controller* implements the methods necessary to access a *Model*, a *Controller* is an instance of a *Model-bound Controller* and implements the access methods for the *View*. Due to the instantiation relation between *Model-bound Controller* and *Controller*, the access methods for the models can be implemented in the *Model-bound Controller* once and reused by each *Controller*.

It is of note that the arrangement of the language architecture should not be mistaken for a dedicated assignment to classification levels, e.g., the position of the concept *Controller* in Figure 2 does not necessarily assign it to M1. On the contrary, our current research indicates that the concept *Controller* should also be differentiated over several classification levels.

However, for the purpose of this paper, a small subset of the domain of discourse is selected, which allows to satisfy the identified requirements. Therefore, the *implemented* language architecture comprises only three classification levels: M_2 , M_1 and M_0 . The levels M_2 and M_1 are for designing a UI for an existing model; the level M_0 represents the execution of the designed UI. That is, the elements of the running UI are actual instances of the *Controller* respectively of the *View*. Following, the specified language concepts for the *View* and for the *Controller* are presented.

The *View* is based on the language concepts presented in Figure 3. These concepts provide a basis for developing a library of all supported *VisualizationElements*. Furthermore, Figure 3 contains selected instantiations, i.e., elements of such a library. *VisualizationElements* are differentiated in *Widgets* and *VisualizationFeatures*. *Widgets* constitute those elements which are for interaction (e.g. Buttons, Radiobuttons or Checkboxes). In contrast, *VisualizationFeatures* serve to characterize the *Widgets* (e.g., the widget’s background-color or foreground-color).

The selected instantiations of the language concepts contain an abstract conception of the *FormularWidget*, which is the supertype for all form-widgets. As an exemplary specialization, the already mentioned *Widgets* Button, Radiobutton and Checkbox are presented. For each *FormularWidget*, both the foreground-color and the background-color have to be

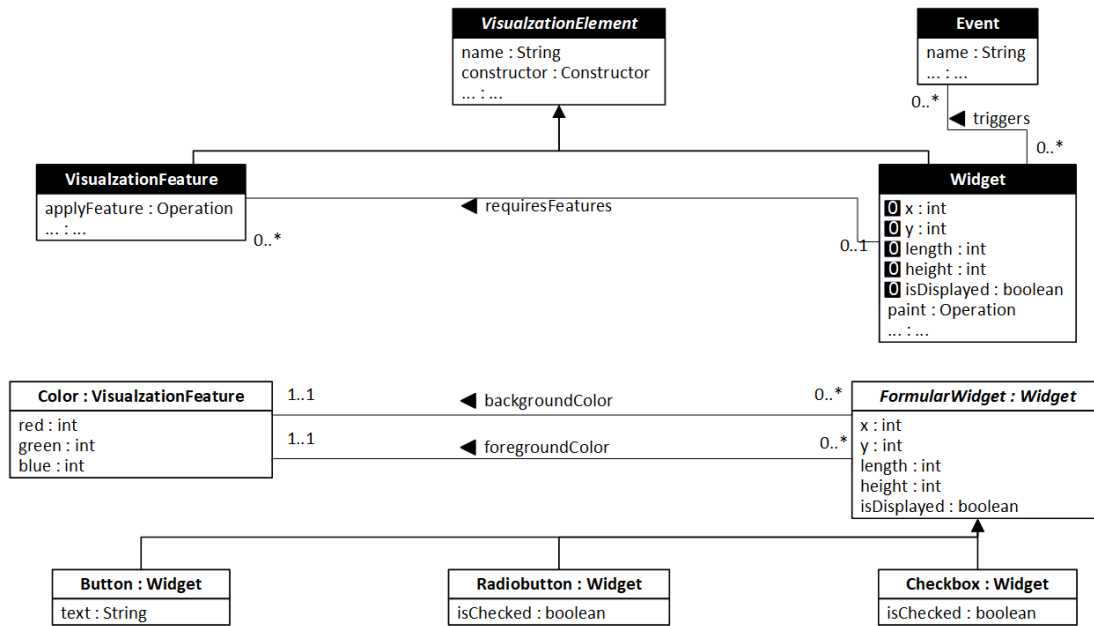


Fig. 3. Language Concepts for the View and Selected Instantiations

specified. Thus, *FormularWidget* has two associations to the feature *Color*.

The language concepts which are specified for implementing the *Model-bound Controller*, the *Controller* and the *Controller-Instance* are presented in Figure 4. As the model is mainly intended to illustrate the associations between the concepts, most of the attributes are faded out. The included language concepts refer to elements which support the development of both the *View* and the *Model*. Elements of the *View* (cf. Figure 3) are marked with a green box with ‘VM’. The *Model* is created with FMML^x, thus, those concepts are part of XCore. Hence, those elements are marked by a yellow box with ‘XCore’. It is of note that this model does only contain classes which are required for instantiating *Model-bound Controllers*, *Controller* as well as *Controller-Instances*; exemplary instances are presented in Figure 5 ② and ③).

The superclass for controllers is the *ElementXWidget* which describes that there is a *Controller* which connects a *Model*-element with a *View*-element. The *ElementXWidget* is specialized into *AttributeXWidget* and *MetaclassXWidget*. Those concepts describe the specific relation of an attribute respectively a metaclass to any kind of *Widget*. *AttributeXWidget* and *MetaclassXWidget* are connected via three relations: *hasAttributeType* on M_2 , *displayAttributeType* on M_1 and *displayAttribute* on M_0 . Those three relations are necessary concepts as they describe different circumstances: *hasAttributeType* comprises all attributes possessed by a class; *displayAttributeType* describes those attributes which are part of a UI, *displayAttribute* relates to those attributes which are actually shown by a running UI, i.e., it relate to potential customization.

ElementXWidget features a relation *ViewRelation* on M_1 for associating the *View*. Such a relation can either be associated

with a *Widget* (*WidgetRelation*) or a *VisualizationFeature* (*FeatureRelation*). The *WidgetRelation* contains a reference to a *Widget* as well as a number of *Configurations*. A *Configuration* refer to a number of *VisualizationFeatures* and is valid in a specific context, e.g., for a given device, a given role or a given user. Thus, it is possible to customize the appearance of a *View* to a specific device, a group of persons (i.e., to roles) or even to a specific user. Furthermore, it is also possible to connect a *Controller* to a *VisualizationFeature*, i.e., it is possible to adapt a *VisualizationFeature* – and therefore, a *View* – according to information stored in the *Model*.

Moreover, each *Controller* might be the root-element of a *Screen* and may have *EventHandlers*. *Screens* logically bundle *Controllers* which *View*’s constitute one UI. Therefore, *Screen* refers to one root-element which might contain further controllers. Thus, the elements of one UI are ordered hierarchically.

EventHandlers are in charge of dealing with events triggered by the *Model* or a *View*. Therefore, handlers allow for detecting user interactions and serve to notify the controller about changes in the *Model* or the *View* respectively. Furthermore, some events might require displaying another *View*. For example, after sending a search query, a *View* has to be displayed which presents the results. Therefore, an *EventHandler* might be connected to a *Screen* via a *FollowingScreenRelation*. A corresponding *Screen* is displayed if the *guard*-condition (i.e., a XOCLExpression) is satisfied.

The defined *Controllers* and *View*-elements on M_1 can be executed by instantiating those elements to M_0 . Thus, a running UI consists of corresponding instances.

Figure 5 presents an application of the implemented prototype. It aims at illustrating the development of a form for

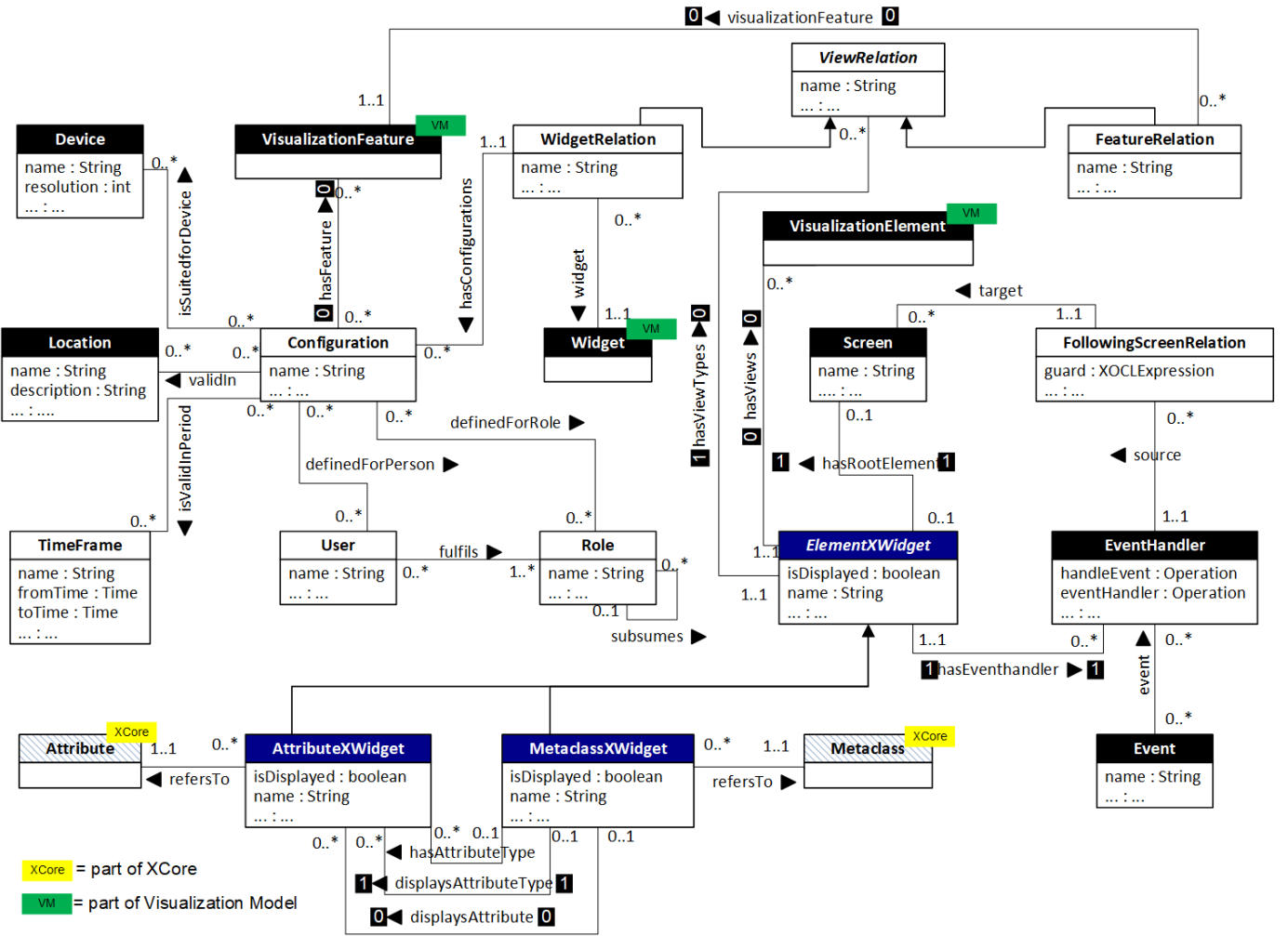


Fig. 4. Multi-level MBUID Metamodel

creating new customers. In this example, the *Customer* is characterized by the attributes *name*, *firstname* and *isVIP* ①. For the UI creation, a corresponding *Model-bound Controller* is created, first ②. Following, it is instantiated to a *Controller* and a corresponding *View* ③. By a further instantiation, the UI is executed, i.e., the form is shown ④.

VI. EVALUATION

Following, it is elaborated, how FMML^x and XModeler in general, and the presented prototype in particular, are able to satisfy the identified requirements for MBUID.

R1 Foster Productivity & R2 Foster Reuse: The FMML^x allows for an arbitrary number of classifications levels, thus, concepts can be defined on a high level of abstraction and be refined on lower levels. Due to *Intrinsicness* it is furthermore possible to specify knowledge regarding low levels of abstraction already on a high level of abstraction. This hierarchy of concepts allow to relax the tension between productivity and reuse [1]. Concepts on a high level of abstraction are generic, thus, they can be reused in a wide range of scenarios. Furthermore, concepts on a low

level of abstraction are semantically rich, thus, they foster productivity. Hence, multi-level modeling enables developing MBUID approaches, which satisfy both **R1** and **R2**, e.g., in the presented prototype the relaxation is visible in two concept hierarchies: First, the MVC-concept *Controller* is refined over three abstraction levels (*Model-bound Controller*, *Controller*, *Controller-Instance*). Second, the abstract concept *Widget* is instantiated into the *Button*, *Radiobutton* and *Checkbox*, which are instantiated into corresponding instances in a UI.

R3 Transparent Propagation of Modifications:

FMML^x and XModeler maintain a shared conceptualization of model and source code. Due to this conceptualization, changes in a model imply changes in the source code. Hence, there is no necessity to implement error-prone mechanisms for model transformations. In terms of MBUID approach, changes in a model imply a direct adaption of the corresponding UI during runtime. Thus, FMML^x and XModeler allow to transparently propagate changes during runtime. Looking at the prototype, changes in the underlying *Controller-Instances* and the *View-Instances* lead immediately to an adaption of

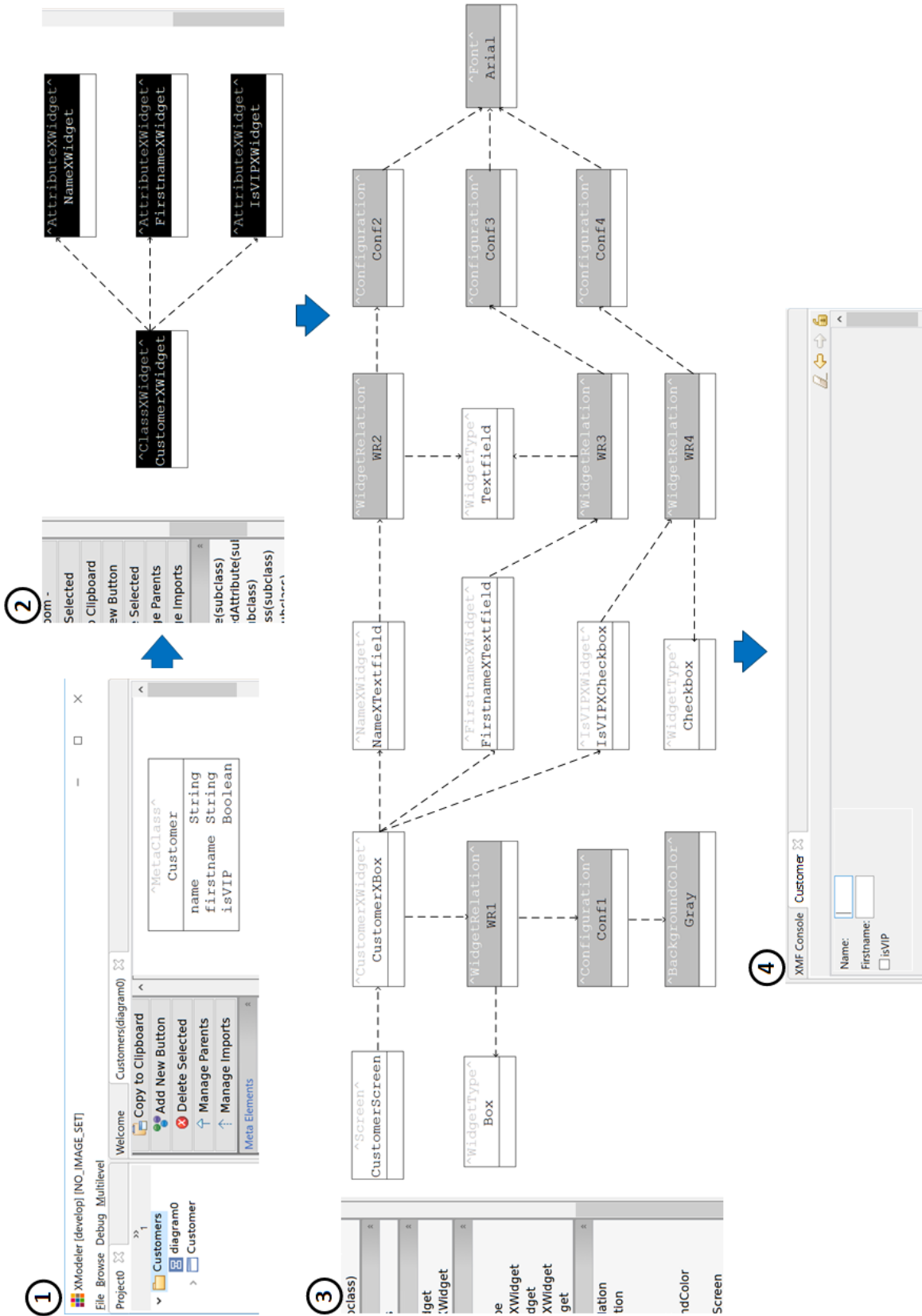


Fig. 5. Screenshots of the UI-creation using the XModeler

the UI. Furthermore, changes to the state of the *Controller*, the *View* or the *Model-bound Controller* influence also the resulting UI.

R4 Extensibility of Language and Adaptability of Tool: By applying FMML^x and XModeler, the definition and the application of a MBUID approach are not strictly separated as both are contained in one multi-level model. Thus, enhancements of the approach are directly usable due to the **R3 Transparent Propagation of Modifications**. Thus, FMML^x and XModeler enable approaches which are extensible and provide an adaptable tool (e.g., the presented prototype).

VII. CONCLUSION

In this paper, we applied a multi-level paradigm to the MBUID-field with the aim to show its advantages for the creation of UIs. For this reason, requirements related to the development UIs were identified which support an effective and efficient model-based development process. Against those requirements, the limitations of the traditional paradigm have been shown. In order to overcome these limitations, a multi-level approach consisting of the multi-level modeling approach FMML^x and language execution engine XModeler has been suggested. In that context, it is discussed, how such a multi-level approach can satisfy the requirements for a model-based UI development. As a proof of concept, a rudimentary prototype has been designed and implemented, which illustrates the potential of multi-level modeling and a corresponding language execution engine for MBUID.

As the developed approach presents only a rudimentary prototype, future work targets the development of a comprehensive multi-level MBUID approach. For this purpose, it is not sufficient to transfer features of existing MBUID approaches to a multi-level MBUID approach, but it is necessary to identify potential for improvement concerning prevalent approaches. Furthermore, it would also be desirable, to not restrict the UI creation to a primary manual task, but to support an automatic generation of UIs. Moreover, it would also be promising to incorporate further interaction technologies in order to incorporate further senses in the human-computer-interaction. Those investigations constitute our future work.

REFERENCES

- [1] U. Frank, "Multilevel Modeling," *Business & Information Systems Engineering*, vol. 6, no. 6, pp. 319–337, Nov. 2014.
- [2] T. Clark, C. Gonzalez-Perez, and B. Henderson-Sellers, "A foundation for multi-level modelling," in *MULTI@ MoDELS*. Tilburg University, Sep. 2014, pp. 43–52.
- [3] C. Atkinson and T. Kühne, "In defence of deep modelling," *Information and Software Technology*, vol. 64, pp. 36–51, Aug. 2015.
- [4] V. A. Carvalho and J. P. A. Almeida, "Toward a well-founded theory for multi-level conceptual modeling," *Software & Systems Modeling*, pp. 1–27, 2016.
- [5] U. Frank, "Designing Models and Systems to Support IT Management: A Case for Multilevel Modeling," in *MULTI@ MoDELS*, 2016, pp. 3–24.
- [6] S. Al-Hilank, M. Jung, D. Kips, D. Husemann, and M. Philippsen, "Using multi level-modeling techniques for managing mapping information," in *MULTI@ MoDELS*, 2014, pp. 103–112.
- [7] G. Meixner, F. Paternò, and J. Vanderdonckt, "Past, Present, and Future of Model-Based User Interface Development," *i-com*, vol. 10, no. 3, pp. 2–11, Nov. 2011.

- [8] J. Vanderdonckt, "Model-driven engineering of user interfaces: Promises, successes, failures, and challenges," *Proceedings of ROCHI*, vol. 8, 2008.
- [9] B. A. Myers and M. B. Rosson, "Survey on User Interface Programming," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1992, pp. 195–202.
- [10] R. Gerbig, "Deep, Seamless, Multi-format, Multi-notation Definition and Use of Domain-specific Languages," Ph.D. dissertation, Verlag Dr. Hut, 2017.
- [11] A. R. Puerta and P. Szkeley, "Model-based Interface Development," in *Conference Companion on Human Factors in Computing Systems*. ACM, 1994, pp. 389–390.
- [12] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML: An appliance-independent XML user interface language," *Computer Networks*, vol. 31, no. 11, pp. 1695–1708, 1999.
- [13] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.
- [14] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A Unifying Reference Framework for multi-target user interfaces," *Interacting with Computers*, vol. 15, no. 3, pp. 289–308, Jun. 2003.
- [15] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "USIXML: A language supporting multi-path development of user interfaces," *EHCI/DS-VIS*, vol. 3425, pp. 200–220, 2004.
- [16] F. Paterno, C. Santoro, and L. D. Spano, "MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, p. 19, 2009.
- [17] P. Akiki, "Engineering Adaptive Model-Driven User Interfaces for Enterprise Applications," phd, The Open University, Sep. 2014.
- [18] I. Marin, F. Ortin, G. Pedrosa, and J. Rodriguez, "Generating native user interfaces for multiple devices by means of model transformation," *Frontiers of Information Technology & Electronic Engineering*, vol. 16, no. 12, pp. 995–1017, Dec. 2015.
- [19] C. Wiehr, N. Aquino, K. Breiner, M. Seissler, and G. Meixner, "Improving the Flexibility of Model Transformations in the Model-Based Development of Interactive Systems," *UsiXML*, p. 46, 2011.
- [20] B. Collignon, J. Vanderdonckt, and G. Calvary, "Model-Driven Engineering of Multi-target Plastic User Interfaces," in *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, Mar. 2008, pp. 7–14.
- [21] M. Blumendorf, S. Feuerstack, and S. Albayrak, "Multimodal User Interfaces for Smart Environments: The Multi-access Service Platform," in *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM, 2008, pp. 478–479.
- [22] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.
- [23] U. Frank, "Enterprise modelling: The next steps," *Enterprise Modelling and Information Systems Architectures*, vol. 9, no. 1, pp. 22–37, 2015.
- [24] C. Atkinson, M. Gutheil, and B. Kennel, "A Flexible Infrastructure for Multilevel Language Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 742–755, Nov. 2009.
- [25] B. Neumayr, K. Grün, and M. Schrefl, "Multi-level Domain Modeling with M-objects and M-relationships," in *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, 2009, pp. 107–116.
- [26] J. De Lara and E. Guerra, "Deep meta-modelling with metadepth," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2010, pp. 1–20.
- [27] C. Atkinson and R. Gerbig, "Flexible Deep Modeling with Melanee," in *Modellierung (Workshops)*, vol. 255, 2016, pp. 117–122.
- [28] T. Clark, P. Sammut, and J. Willans, *Applied Metamodeling A Foundation For Language Driven Development: Second Edition*. Ceteva, 2008.
- [29] —, *Superlanguages: Developing Languages and Applications with XMF*. Ceteva, 2008.
- [30] D. H. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books, 1979.
- [31] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, no. 3, pp. 345–359, 2008.
- [32] T. M. H. Reenskaug, "The original MVC reports," Dept. of Informatics, University of Oslo, Oslo, Tech. Rep., 1979.