Integration of a Template System into Model-Based User Interface Development Workflows

Christopher Martin(✉) and Annerose Braune

Institute of Automation, Technische Universität Dresden, Dresden, Germany
{christopher.martin,annerose.braune}@tu-dresden.de

Abstract. In order to reduce the work effort necessary for the creation of user interfaces (UIs), Model-Based UI Development workflows have been introduced. They allow the automatic generation of UIs by means of model transformations. Transformation rules define how to query data from the source model and convert it into elements of the target model. By describing rules for the creation of the target model, it also defines the design of the resulting UI. A separation of these two types of information enables an easier adaptation of the resulting design, e.g. to satisfy design guidelines, without having to reprogram complex transformation rules.

We therefore introduce a generic model-based template system that can be integrated into model-based workflows to separate the design from the data query rules. A case study demonstrates the integration of our template system into a workflow that automatically generates Human-Machine Interfaces from engineering data of manufacturing plants.

Keywords: Human-Machine Interfaces · Template systems · Model-Based User Interface Development · Model-based workflows

1 Introduction

Human-Machine Interfaces (HMIs) are used for supervision and control of complex automation processes and systems. Their creation requires a major part of the plant engineering work effort (cf. [1]). As most of the plant engineering is done using computer-aided engineering software, the engineering data is usually digitally available. However, the creation of HMIs is still mostly done manually. First approaches to change this fact and allow for an integrated HMI development process have been developed only recently (e.g. [2,3]). These workflows make use of the *Model-Based User Interface Development* (MBUID) approach that uses models to describe User Interfaces (UIs) at different degrees of abstraction. Model transformations are used to change the level of abstraction, e.g., by converting engineering information of manufacturing plants into the navigation structure, layout, and panel setup of the HMI (cf. [2]). Furthermore, model elements representing UI widgets are created during this concretization process that

M. Kurosu (Ed.): HCI 2017, Part I, LNCS 10271, pp. 480–495, 2017.
DOI: 10.1007/978-3-319-58071-5_36

also have to be parameterized by the model transformation, e.g., the position or size of an UI element has to be defined.

A model transformation aggregates transformation rules that define how elements of a source model can be converted into elements of a target model (cf. [4]). For this, the model transformation rules have to define how to get (resp. query) data from the source model and how it should be displayed in the target model. Hence, in a MBUID workflow the design of a generated user interface is defined by model transformation rules. The final design of a user interface reflects the design decisions that are made by an expert based on domain knowledge, design guidelines, and information about the application's context of use, e.g., about the screen size of the target platform. So far, in MBUID workflows these decisions have to be hard-coded into model transformation rules. In order to enhance the reusability of this expert knowledge, the design decisions have to be formalized and separated from the transformation logic.

In software engineering, template systems merge selected data (usually queried from a database) and templates (representing the design) to create documents (cf. [5]). Thus, they allow a separation of the data and the design of a document. In this paper, we therefore introduce an approach to incorporate such a template system into a Model-Based UI Development workflow, i.e., especially into the transformations used in such workflows. As templates can be formulated in regular UI description languages, the user is also enabled to design the resulting HMI with the help of the respective editors that he is already used to rather than having to rewrite complex model transformation rules. Furthermore, the final design can be previewed and tested more easily as templates may be created as regular UI models rather than a mere set of rules.

After a short introduction of the MBUID concepts in Sect. 2, requirements for the realization of a template system for MBUID workflows will be deduced in Sect. 3. Based on those, a generic model-based template system is introduced in Sect. 4 and demonstrated by means of a case study in Sect. 5. Related work is then discussed in Sect. 6. Finally, in Sect. 7 we will draw conclusions about the applicability of a template system in MBUID workflows and discuss future works.

2 Model-Based User Interface Development

With the increasing number of available hardware platforms, such as smartphones, tablets, or even smartwatches, the work effort required for the creation of user interfaces has also increased in recent years as an UI has to be created specifically for each platform. In an effort to overcome this problem, the concept of Model-Based UI Development has been introduced. It uses formal models to describe UIs at different degrees of abstraction. In order to provide a systematic workflow in MBUID, the *CAMELEON Reference Framework* [6] (CRF) has been introduced. As shown in Fig. 1, it defines a workflow that comprises four levels of abstraction.

The Task and Concept (T&C) level describes the tasks that an user wants to perform with the objects (concepts) of the user interface and the order in which

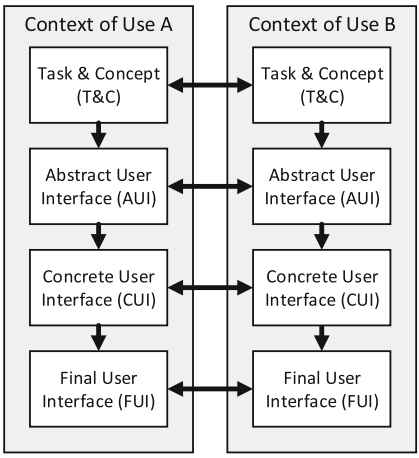

Fig. 1. Simplified representation of the CAMELEON reference framework (cf. [6])

the task are performed. The next level — the Abstract User Interface (AUI) — specifies the structure and elements of the UI in a platform- and modality-independent way. For example, it may define that a multiple choice selection is part of the UI but not if this selection should be represented by a checkbox or a vocal selection. The Concrete UI (CUI) adds information about the modality but not yet about the runtime platform. A CUI model may, e.g., describe a graphical user interface that has a panel which includes the checkbox mentioned before. Lastly, in the Final UI (FUI) the runtime platform is defined, i.e., it represents the actual executable user interface.

A change of the level of abstraction is usually realized by means of model transformations. In order to create a target model, a model transformation has to obtain information from the source model, the context of use, and about elements that have already been created (e.g., if a reference to an already existing element has to be created). Transformation rules specify how both the static and flexible elements of a target model are created. While the static parts are created without the need of data from the source model, the concrete realization of the flexible elements is determined based on data from the source model. Therefore, transformation rules have to define how this data can be queried from the source model and how to translate it into elements of the target model.

In order to convert an abstract UI model into a more concrete description of the UI, additional information has to be added that is not defined by the source model, e.g. about the kind of UI elements and their positioning when transforming an AUI to a CUI. This information is hard-coded into the transformation rules. Thus, different transformation rules are required to change the context of use (cf. Fig. 1), e.g., to create a vocal instead of a graphical user interface. However, this approach requires only one set of transformations that may be reused for every application with the same context of use, i.e., a CUI-to-FUI

transformation that creates an executable UI for a certain target technology can be reused for the generation of every UI with the same target technology.

UI models are created by means of UI modeling languages. Various UI modeling languages — such as MARIA [7], UsiXML [8] or Movisa [9] — have been established that allow the definition of UI models at different levels of the CRF. Although efforts for a standardization of a common UI modeling language have been expressed by a W3C working group [10], no modeling language has yet established itself as the standard above the others.

3 Requirements for Model-Based Template Systems

Template systems create documents by merging selected data with the predefined design of templates. To achieve this, a template engine has to query data — e.g., from a database or in case of a MBUID workflow from a source model — as defined by data query rules that are usually written in a normal programming language such as Java [11] or PHP [12]. This data is then used by the template engine to substitute placeholders (also called *template variables*) that have been specified in the templates. A template consists of static parts that do not change based on any data and flexible parts represented by template variables that require the querying of further data to determine their actual value. Thus, the combination of a template including template variables and a set of data query rules is comparable to a transformation rule as both specify rules for the generation of documents. Therefore, template systems can be used to replace regular rule-based model transformations in MBUID workflows.

A resulting workflow for the utilization of a template system in a MBUID workflow is depicted in Fig. 2. In order to allow the generation of a target model, the user first has to define his templates that resembles the desired design of the UI that shall be generated. By assigning template variables as placeholders, the user can declare the flexible elements of the template which shall usually be substituted by source model information or possibly even other sources of information. Once all templates are finished, the data query rules have to be defined, which describe how the data necessary for replacing the template variables can be obtained. In this paper, we will focus on the acquisition of such data only from a source model. While the data query rules should be created by a MBUID expert (as deep knowledge of the source model and the querying language is needed), the templates could be created by a design expert as only knowledge about the target UI modeling language is needed. Finally, the template engine (cf. TE in Fig. 2) replaces the template variables with data from the source model as defined by the data query rules and thus generates the target model.

Templates may be created using the same technology as the document that should be generated. For example, if a website should be created based on a database, a normal template engine such as [11] might allow the specification of the data query rules using Java and the templates could be defined using HTML. As the templates are normal HTML pages — except for the template variables — a designer may create it using his standard web development workflow.

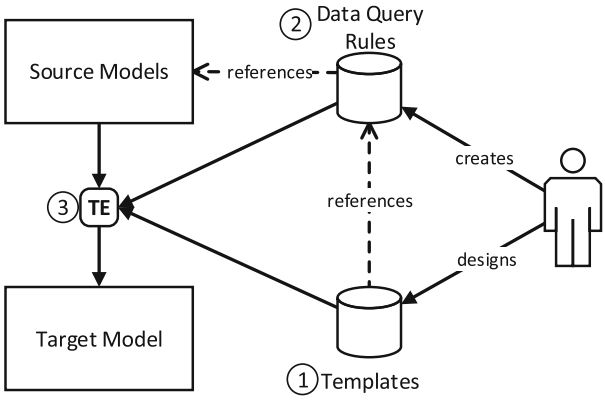

Fig. 2. The workflow for the application of a template system in a MBUID workflow

However, the placeholders have to be specified using a specific syntax to allow the template engine to link the acquired data to its respective template variable. In order to keep this beneficial characteristic of a template system intact when integrating it into a MBUID workflow, templates have to be definable in a regular UI modeling language.

Usually each step of a MBUID workflow is realized using a modeling language that is specific to the level of abstraction of the respective step. The set of modeling languages used in a MBUID workflow can vary widely between different workflows. As described in Sect. 2, no modeling language has established itself as a standard thus far. Hence, if a template system is to be used in a MBUID workflow, it has to be generic in respect to the potential modeling languages of the workflow, i.e., the template system must not be created for a specific UI modeling language but allow the use of any language.

As the creation of model transformations require a large part of the work effort necessary for the creation of a workflow, model transformation languages have been developed — e.g., *ATL* [13] or *Epsilon* [14] — that extend standard programming languages by offering functionalities specific to model editing such as an easier referencing of model objects and values. Hence, when integrating a template system into a MBUID workflow, transformation languages should be used for data querying rather than standard programming languages.

In summary, three components have to be realized in order to integrate a template system into a MBUID workflow. These can be seen in the workflow shown in Fig. 2: ① templates including template variables, ② data query rules required to acquire data from a source model that shall be used for replacing the template variable values, and ③ a template engine that scans the templates, substitutes the template variables based on the data query rules, and thus creates the target model. As also discussed in this section, the definition of all information should be realized generically in order to not restrict the user in his choice of UI modeling or model transformation languages.

4 Generic Model-Based Template System

As defined in the last section, templates should be created as normal UI models using any UI modeling language. The structure of a modeling language is defined by its meta-model. Using the elements that are offered by the meta-model of an UI modeling language, the static part of a template can easily be described. As these meta-models normally do not include elements that allow the specification of template variables, the declaration of the flexible part must be done externally in order to not require any changes to their existing meta-models. Therefore, a new meta-model for the generic annotation of template variables to elements of an UI model has to be created. This can be done using different technologies, as modeling languages can be represented in multiple ways, such as XML or as part of the *Eclipse Modeling Framework*[1] (EMF). While models that are described in XML are based on meta-models that are defined using the *XML Schema Definition* (XSD), meta-models in EMF are created based on the *Ecore meta-model*. However, EMF is also able to load XSD-based models and thus also allows the processing of modeling language that are represented in XML. Furthermore, it provides a large number of tooling for the convenient processing of models such as multiple model transformation languages and generic editors. Consequently, we decided to base our approach on EMF as it offers the most flexibility when working with varying UI modeling languages.

In EMF, a model consists of a hierarchy of objects (*EObject*) that are linked to each other via references (*EReference*). In contrast to XSD-based models, values are only stored in attributes (*EAttribute*) and not as content of an object. Hence, usually only attribute values should be replaced by a template engine as the object structure is already statically defined in the template, i.e., template variables may only be annotated to attributes.

However, certain situations can also require a change of the object hierarchy, e.g., if based on data from a source model an alternative UI widget should be used or if a widget needs to be used repeatedly, because the corresponding source model element has a cardinality greater than one. Thus, an annotation of objects is also necessary but in a different context: the repeating resp. the omitting of an object structure or the selection of alternative model objects is required rather than the substitution of a value. To allow for the specification of this, the template meta-model has to support the annotation of repeatable and optional model objects as well as the definition of the rules that describe how often an object should be instantiated or when an object should be displayed or omitted.

A simplified version of the resulting meta-model is shown in Fig. 3. It does not include any actual UI description elements as the template should be defined as a regular stand-alone UI model. Therefore, only a path to the template is given in the *TemplateModel*. It allows the annotation of template variables by allowing to reference any EAttribute of a template and the assignment of an unique *identifier* which is used for referencing of specific template variables.

[1] http://eclipse.org/modeling/emf/.

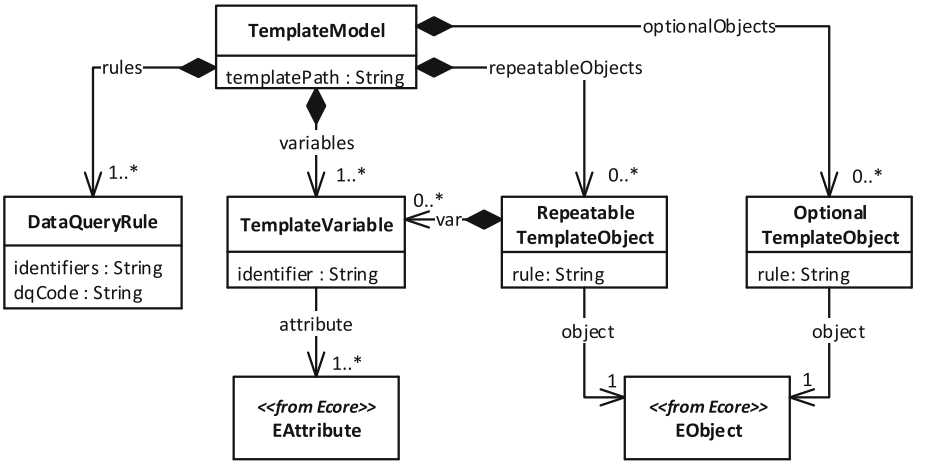

Fig. 3. Simplified subset of the template system meta-model

This enables the linking of a variable to a data query rule. By defining the link using identifier rather than references, a loose coupling of the data query rule and the template variable is achieved which allows for an easy change of the template (and hence the template variables) without the need of recreating any references to data query rules.

Furthermore, template objects can be annotated to allow for the definition of repeatable and optional elements. By allowing the reference of any EObject in a template, the root object of an UI widget can be tagged. The tagging then also applies to all child elements as they are part of the widget that shall be repeated or not displayed. While template variables are occurring in both optional and repeatable objects, only in case of a repeatable object the association of the template variable with the object is relevant as information about the cardinality of the object may be required, e.g., to calculate the size and position of UI elements if aligned in a list. If an object must be repeated, it will be initialized multiple times and its template variables will be reprocessed for every new object. The resulting object can then be added to the containment tree of the target model.

With the template annotations done, data query rules have to be defined. These rules are described as model transformation rule fragments in a transformation language of choice and stored in the *rule* resp. *dqCode* attributes. These fragments have to be called by the template engine when a template variable is applied. However, as the rules can be written in different transformation languages, a transformation language adapter has to be created specifically for each transformation language that should be supported by the template engine. The adapter realizes the calling of the fragments, the handover of parameters such as the source model that shall be queried, and the handling of the return value. When starting the processing of a template, the template engine creates a

copy of the template as the new target model. It then iterates the template and processes the template variables and objects when found. The creation of the final model can thus be viewed as an in-place model transformation that works on a copy of the template.

5 Case Study: Template System for AutoProBe

In order to demonstrate our approach, we present a case study that shows the application of the template system in a workflow that allows the (semi-) automatic generation of Human-Machine Interfaces as introduced in the project *AutoProBe* [2]. The workflow used in AutoProBe consists of multiple steps that are depicted in Fig. 4a. Based on a plant model that aggregates data from the engineering process of a manufacturing plant, the plant hierarchy is used to generate a navigation structure (step ①). Next the basic panel setup (step ②) is performed during which most of the UI elements are created and placeholders for the plant-specific elements are placed on the different panels. The positioning is done based on the plant geometry and a predefined fixed layout that can be seen in Fig. 5. The placeholders are then replaced by parametrized library elements (cf. [15]) to create the concrete HMI model (step ③). Information about the actual components and sub-stations are used for the library item initialization. In a last step, code for an executable HMI in HTML/JavaScript is generated. However, an export to industrial HMI tools, such as *SIMATIC WinCC*[2] by Siemens, is also possible if appropriate import interface are offered by the tool.

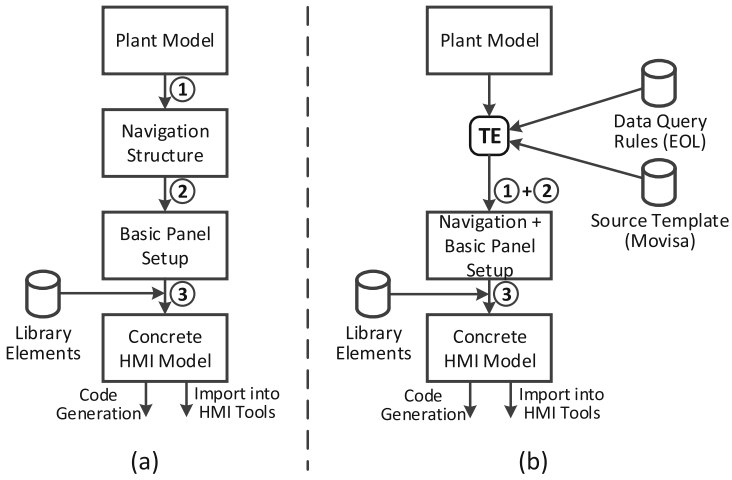

Fig. 4. (a) The AutoProBe workflow as introduced in [2] and (b) the new workflow after integration of a template system

[2] http://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/scada/simatic-wincc/Pages/default.aspx.

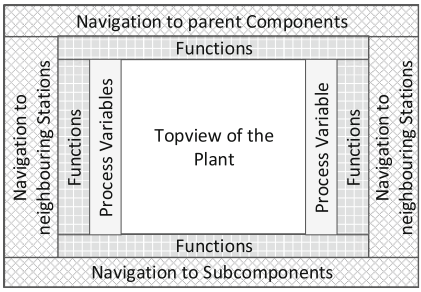

Fig. 5. Default layout used for UI generation in the AutoProBe workflow (cf. [2])

As illustrated in Fig. 4b, we have integrated the template system into the first and second step of the workflow. Those two steps can be merged as a separate transformation step is not necessary anymore, because the creation of the navigation structure can be handled with the help of repeating template objects and the positioning, initializing, and parameterizing of UI elements can now be done in the same step by using template variables. Therefore, in the new workflow steps ① and ② have been merged. The capability to initialize concrete library entries — as normally done during step ③ — has not yet been integrated into the template engine but might also be added to it in the future, i.e., all three intermediary steps could then be processed by the template system. Some of the library elements could in the future even be realized as part of the template and would thus not be needed in the library anymore.

In AutoProbe, the UI modeling language Movisa [9] was used for the definition of the intermediary models, i.e., for all models except for the plant model. Movisa was created to allow the definition of HMIs specifically for the domain of industrial automation and is situated at the CUI level of the CRF. It was also used for our case study and thus the templates have been defined in Movisa. As the transformations in AutoProBe have been implemented using the *Epsilon Object Language* (EOL) [14], the data query rules are also defined in this language. To allow for this, an EOL data query adapter has been implemented for our template engine. It can call EOL operations that query data from a given source model and return a value that is used for the template variable substitution. In order to support the user, an interactive wizard was implemented that allows the annotation of *TemplateVariables* and *TemplateObjects* from the regular Movisa editor. It allows the assigning of identifier and if a *DataQueryRule* element is created, an EOL operation stub for the data query rule is also created automatically. Furthermore, an attribute was introduced that allows the value of template variables that are part of a repeatable object to be made unique upon initialization. As Movisa requires every element to have an unique ID, this avoids redundant data query rules that only checks if an ID is unique and if not appends an unique number to the original value of an attribute.

For our case study, we created two templates that realize two different layouts: the default layout as shown in Fig. 5 and a layout for smaller screens as illustrated in Fig. 6 that uses the *Off Canvas* UI pattern for the navigation menu. It is a very common UI pattern in mobile design (e.g., as recommended by Google [16]) and displays content in a panel that is not shown on screen by default but slides in from the side when activated by hitting the menu button in the top left corner of the UI. Furthermore, functions and process variables are displayed in a scrollable list. The topview of the plant is replaced by a scrollable list of the plant's components.

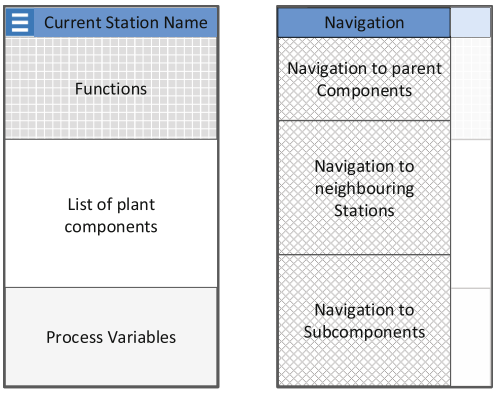

Fig. 6. Alternative layout for mobile devices with hidden (left) and activated (right) Off Canvas navigation menu

An example of the Movisa template can be seen in Fig. 7 which shows the section of the template that is used to generate the top navigation bar in the default layout (cf. Fig. 5). It consists of only a single *Button* that shall be repeated for every available navigation target of a higher level in the hierarchy of the plant. By annotating the element as a *RepeatableTemplateObject* a rule can be defined that specifies for which elements of the source model a new navigation button has to be created. The IDs of all elements are marked to be made unique, because they may be initialized multiple times as they are part of an object that may be repeated. The Button is placed on the screen by defining the x-position (*XScaledProperty*) and y-position (*YScaledProperty*). While the latter is fixed, the x-position depends on the number of buttons in the navigation bar. Therefore, a *TemplateVariable* and a *DataQueryRule* are defined that realize the calculation of the x-position. This way, the buttons are equally spread and center-aligned horizontally in the top navigation bar. For reasons of clarity only this template variable and data query rule as well as only the button's position are displayed in Fig. 7. Similar rules are used for the other navigation bars, functions, and process variables. The topview is positioned on both axes based on a topview of the plant as defined by the plant topology in the plant model.

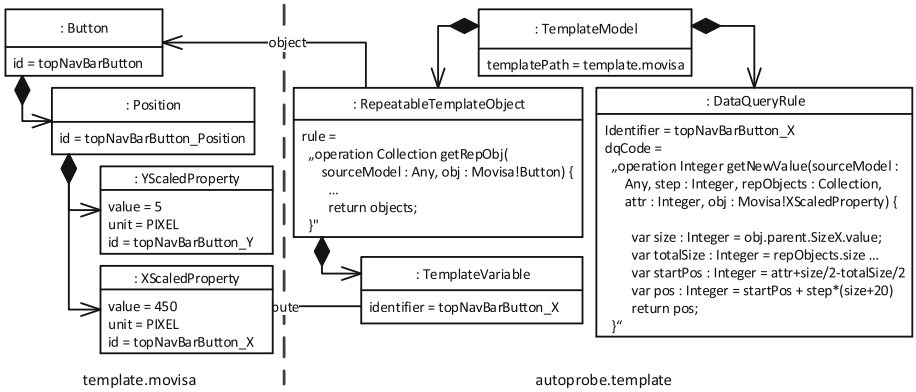

Fig. 7. Simplified representation of the subset of the Movisa template that defines the top navigation bar (left) and the respective section of the *TemplateModel* (right) for the default AutoProBe layout

The Off Canvas layout uses a comparable principal by aligning elements vertically instead of horizontally. Thus, the resulting TemplateModel is quite similar as it also defines TemplateVariables for the y-position of the various RepeatableObjects. However, the topview of the plant has been altered for this design as it becomes too sophisticated to be still comprehensible on the small screen of a mobile device. Therefore, the elements are also aligned vertically in a scrollable list.

Since templates are defined as regular Movisa models, executable code can be generated from them without the need of already having template variables and data query rules specified. This allows the early testing of the template's function already during design phase. Figure 8 shows the generated FUI for the Off Canvas template. The function of the menu activation can already be tested even though the template has not yet been annotated with template variables and data query rules. Furthermore, it enables the designer to get a first impression of the resulting design of the final application at an early phase of the workflow creation process.

Examples of the resulting executable UIs are shown in Figs. 9 and 10 that show the detail page of a quality testing station's PLC and list important process variables of that station. The station and its process variables were described in a plant model that was the input for the workflow. By using a template system, we enable the user to switch the final design of the generated user interface by just switching the template. A change in the static parts of the final user interface — e.g. the background colors or a header logo — does not require reprogramming of transformation rules anymore, but can be realized using the regular Movisa editors. However, if changes to attributes are required that have been defined by a template variable, the respective data query rule has to be reprogrammed as well.

Furthermore, we were able to define templates for the navigation structure, the basic panel setup, and some elements of the concrete HMI model.

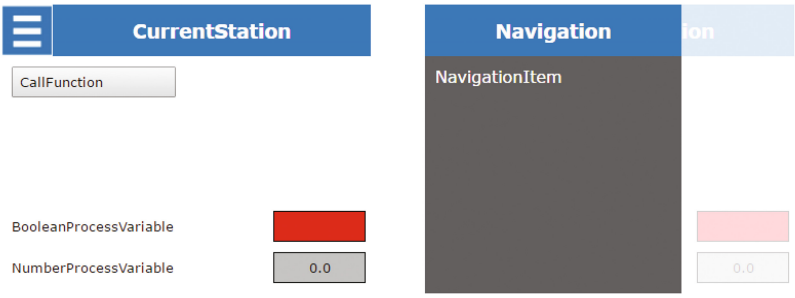

Fig. 8. Generated HTML/JavaScript version of the Movisa template for the Off Canvas layout as introduced in Fig. 6 (empty bottom half was cropped)

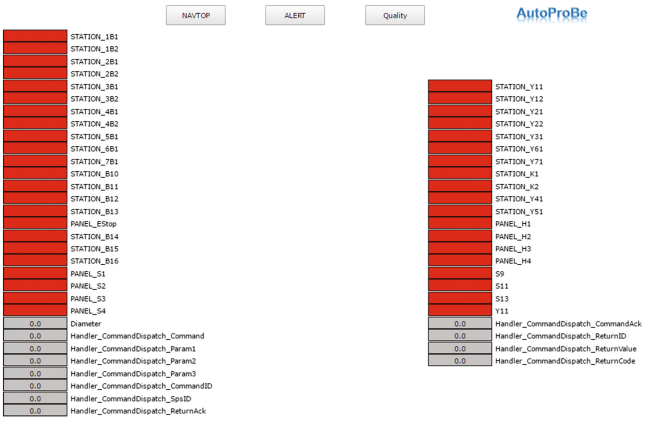

Fig. 9. FUI generated by the AutoProBe workflow for the detail view of a quality testing station using the default AutoProBe layout template

Fig. 10. Detail view of a quality testing station (cf. Fig. 9) generated using the Off Canvas layout template (cf. Fig. 8)

Therefore, we reduced the number of transformations by merging steps ① and ② as well as the number of library elements necessary by integrating some of them into the Movisa template.

6 Related Work

Templates are often used in model-to-text transformations, i.e., transformations that generate text from a source model. One example for this is the standardized XML transformation language *XSLT* [17]. It allows the specification of template rules for XML nodes. If such node is found in the source XML document, the template rule is executed and the results are inserted into the final document. Such rule-based template definition is very common in MBUID workflows as most model-to-text transformations follow the same template-based principle. For example, the *Epsilon Generation Language* (EGL) [18] allows the definition of text snippets as templates that shall be inserted into the target model if a certain element is found in the source model. Model-to-model transformations (e.g. ATL [13]) follow a similar principle by defining transformation rules for source model elements that create target model elements. However, when using these transformation languages, it is still necessary to define rules for the creation of static contents by means of a specific rule definition syntax. Therefore, the design of the generated UI can only be changed by reprogramming complex transformation rules. By default it is not possible to reference templates that are defined in a regular UI modeling language as it is allowed by our approach.

In the domain of web-based user interfaces, template systems are already very common. Content management system, such as *Joomla*[3] or *WordPress*[4], use templates to define the design of the site independent of the actual contents. These templates are usually created as HTML or PHP pages that contain predefined placeholders which are then replaced at runtime by contents from a database. The data query rules used by the template engine are hard-coded into the system though. The principle of this approach is comparable to our solution as it allows the definition of the templates in the target technology. However, as the data query rules are integrated into the system and rely on data from a database, it cannot be applied to MBUID workflows.

Various general purpose template engines exist that allow the definition of templates and data query rules in multiple technologies. Some of them — such as [11,19] — also support the specification of templates in regular XML, i.e., every template is still valid XML. As most modeling languages are defined in XML or XMI, these template engines can also be used to create templates in most UI modeling languages and could thus be integrated into a MBUID workflow. While this enables the separation of the design from the data query rules, the rules have to be implemented in a regular programming language specific to each template engine. As no template engine supports a model transformation language by default yet, it is not possible to use the model accessing and editing

[3] https://www.joomla.org/.
[4] https://wordpress.org/.

capabilities of common model transformation languages for the data query rule definition. In the future, we will continue to analyze more template engines in regard to their applicability to MBUID workflows and in regard to the features offered by the systems in order to further enhance our template system.

In the domain of MBUID, Sinnig et al. [20] describe an approach that uses the template engine *Velocity* for the instantiation of UI pattern in an UI model. The pattern are saved as UI model snippets including template variables that are replaced during initialization. As this solution was created specifically for the XUL modeling language, it is limited to this UI modeling language and thus lacks the required flexibility. In contrast to our approach that uses a template engine for the transformation of a source model to a target model (cf. Sect. 3), the Velocity engine is only used to manually add elements into a preexisting UI model. Thus, it does not allow the specification of data query rules as the template variables are resolved manually.

Aquino et al. [21] present *Transformation Templates* that separate the specification of the structure, layout, and style of UI elements from the model transformation rules by providing them as *Parameters* for the model transformation. However, the definition of such Parameters is done on the meta-model level, i.e., the Parameter is applied for every element of the specified type. This may be limited by use of selectors, but the changes are only applied after the element has been created in the target model. The creation of the target model is not part of this approach and has to be performed separately. Thus, the transformation templates can be seen as a library of target creation rules that describe how the layout of the target model should look like or how an element should be styled. It does not allow the definition of data query rules and can therefore only be used in conjunction with a dedicated model transformation. Furthermore, the definition of the Parameters has to be done in a specific modeling language, i.e., the template may not be created in a regular UI modeling language.

7 Conclusions and Future Work

In this paper, we have presented an approach for the separation of the design from the model transformation rules of MBUID workflows by integration of a template system. To meet the requirements of a template system in a MBUID workflow, we created a generic *TemplateModel* that allows the definition of data query rules and the formal annotation of template variables independent of the UI modeling language. The model transformation language used for data querying may also be changed by implementing a data query rule adapter for the template engine. While this still requires a lot of work, the implementation is only necessary once per model transformation language and can then be used for all template models. Our proposed template system is able to create a target model based on source model data, templates, and the template model.

We have demonstrated the applicability of this approach by means of a case study that showed that a template system promotes the separation of the design and data querying in MBUID workflows. As templates are created as regular UI models, the resulting UI of a workflow can be designed using the regular tools

associated with the UI modeling language rather than integrating the design into transformation rules. Furthermore, as templates are valid UI models, they can already be transformed into FUI without the need to execute the whole workflow. This enables the testing of the UI design and some of its mechanics (e.g., the expanding and collapsing of an off canvas panel) at a very early stage of the design process.

Furthermore, we have shown that a change of the design of a generated UI can be realized by just switching the template if only static content should be changed. If attribute values should be changed that were defined by a template variable, the respective data query rule might also need to be changed. As the definition of these rules is still a very complex task, we will examine how to improve this task in future works. One possible solution could be the creation of an explicit model querying language, e.g., based on the source section definition as introduced by *PAMTraM* [22].

As already shown in our case study, UI pattern support the user in the development of user interfaces by offering proven solutions to reoccurring design problems. By providing the user a library of UI pattern, the work effort necessary for the creation of templates can be reduced. In future work, we therefore want to integrate the possibility to store and reuse UI pattern as template fragments.

Additionally, our approach has to be evaluated by means of further case studies. We are currently considering the application of the template system in the context of plug-and-produce scenarios as introduces by the *Industry 4.0* initiative that proposes reconfigurable process and manufacturing plants. At the moment, there is a lack of concepts on how to adapt the HMI to a change in the plant configuration. A template system could eventually be used to overcome this challenge.

Acknowledgments. The IGF proposal 16606 BG of the research association "Gesellschaft zur Förderung angewandter Informatik e.V." (GFaI) is funded via the AiF within the scope of the "Program for the promotion of industrial cooperational research" (IGF) by the German Federal Ministry of Economics and Technology (BMWi) according to a resolution of the German Bundestag.

We gratefully acknowledge funding of the project "KonTrans" – BR4107/2-1 by the "Deutsche Forschungsgemeinschaft" (DFG).

References

1. Myers, B.A., Rosson, M.B.: Survey on user interface programming. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 1992, pp. 195–202. ACM, New York (1992)
2. Martin, C., Freund, M., Braune, A., Ebert, R.E., Pleßow, M., Severin, S., Stern, O.: Integrated design of human-machine interfaces for production plants. In: Proceedings of 20th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2015), pp. 1–6. IEEE, September 2015
3. Schleipen, M., Okon, M., Enzmann, T., Wei, J.: IDA - Interoperable, semantische Datenfusion zur automatisierten Bereitstellung von sichtenbasierten Prozessführungsbildern. In: VDI-Berichte, vol. 2143, pp. 83–86. VDI-Verlag, Düsseldorf (2011)

4. Gruhn, V., Pieper, D., Röttgers, C.: MDA - Effektives Software Engineering Mit UML2 Und Eclipse. Springer, Heidelberg (2006)
5. Niemeyer, P., Knudsen, J.: Learning Java, 3rd edn. O'Reilly, Sebastopol (2005)
6. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interacting with Computers **15**(3), 289–308 (2003)
7. Paternò, F., Santoro, C., Spano, L.D.: MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput. Hum. Interact. **16**(4), 1–30 (2009)
8. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: a language supporting multi-path development of user interfaces. In: Bastide, R., Palanque, P., Roth, J. (eds.) DSV-IS 2004. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005). doi:10.1007/11431879_12
9. Hennig, S.: Design of Sustainable Solutions for Process Visualization in Industrial Automation with Model-Driven Software Development, 1st edn. Jörg Vogt Verlag, Dresden (2012)
10. Fonseca, J.M.C., Calleros, J.M.G., Meixner, G., Paternò, F., Pullmann, J., Raggett, D., Schwabe, D., Vanderdonckt, J.: Model-based UI XG final report. Technical report, W3C (May 2010)
11. The Thymeleaf Team: Thymeleaf (2017). http://www.thymeleaf.org/
12. New Digital Group, Inc.: PHP Template Engine—Smarty (2016). http://www.smarty.net/
13. Eclipse: ATL - a model transformation technology (2016). http://www.eclipse.org/atl/
14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006). doi:10.1007/11787044_11
15. Freund, M., Martin, C., Braune, A.: A library system to support model-based user interface development in industrial automation. In: Kurosu, M. (ed.) HCI 2016. LNCS, vol. 9731, pp. 476–487. Springer, Cham (2016). doi:10.1007/978-3-319-39510-4_44
16. Pete LePage: Responsive Web Design Patterns—Web (2017). https://developers.google.com/web/fundamentals/design-and-ui/responsive/patterns
17. W3C: XSL Transformations (XSLT), November 1999. https://www.w3.org/TR/xslt
18. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The epsilon generation language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69100-6_1
19. Wanstrath, C.: Mustache(5) - Logic-less templates (2009). http://mustache.github.io/mustache.5.html
20. Sinnig, D., Gaffar, A., Reichart, D., Forbrig, P., Seffah, A.: Patterns in model-based engineering. In: Jacob, R.J.K., Limbourg, Q., Vanderdonckt, J. (eds.) Computer-Aided Design of User Interfaces IV: Proceedings of CADUI 2004, pp. 197–210. Springer, Dordrecht (2005)
21. 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. ACM, New York (2010)
22. Freund, M., Braune, A.: A generic transformation algorithm to simplify the development of mapping models. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, pp. 284–294. ACM, New York (2016)