A Library System to Support Model-Based User Interface Development in Industrial Automation

Matthias Freund(✉), Christopher Martin, and Annerose Braune

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

Abstract. Conventional visualization systems in industrial automation are equipped with powerful library systems that offer aggregated reusable visualization elements. These represent important domain-specific knowledge and promote the necessary consistent look-and-feel of visualizations. In contrast, modeling languages used in model-based workflows do not offer library systems. This hinders the acceptance of such approaches in industrial automation.

Hence, this paper presents an approach towards a library system for the model-based development of visualizations. We present a generic approach that can be applied to a variety of modeling languages. A case study demonstrates this for one specific modeling language and illustrates the application of the library system.

Keywords: Human-machine interfaces · Library system · Model-based user interface design · Model transformation

1 Introduction

In industrial automation, human-machine interfaces (HMIs) are used to supervise and control technical processes and machines. Developing these HMIs is in general performed manually by an HMI engineer with industrial visualization systems like *Simatic WinCC*[1] or *Wonderware InTouch*[2]. Such visualization systems offer powerful tools and editors, one key asset being their very extensive library systems that offer different domain- and/or company-specific libraries. As illustrated in Fig. 1, such libraries are created by domain experts and contain reusable parameterized elements (called *widgets*, *symbols*, or *faceplates*). These library entries contribute to the supply of built-in elements that can be used by an HMI engineer for the creation of an HMI. Thus, they represent important domain-specific knowledge and promote a consistent look-and-feel of visualizations that is especially important in industrial automation. In order to browse the entries that are available inside a library as well as to retrieve and insert a selected entry into a target HMI, library systems also offer a set of management algorithms (cf. Fig. 1).

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

[2] http://global.wonderware.com/DE/Pages/WonderwareInTouchHMI.aspx.

M. Kurosu (Ed.): HCI 2016, Part I, LNCS 9731, pp. 476–487, 2016.
DOI: 10.1007/978-3-319-39510-4_44

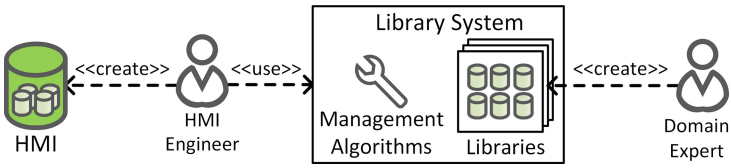

Fig. 1. The role of library systems during the development of HMIs

However, the available industrial visualization systems are committed to a single target technology resp. to a fixed set of target platforms. *Model-based User Interface Development* (MBUID) approaches like *UsiXML* [1] or *MARIA* [2] try to solve this problem by allowing an HMI engineer to specify the functionality of an HMI via declarative models [3] that abstract from implementation details and can ideally be used to generate executable HMIs for different hard- or software platforms. Whereas MBUID has been a hot topic in the research community for many years, acceptance by the industry is according to [3] still a major issue. While most approaches offer more or less powerful graphical editors and limited tool support, one of the main problems compared to industrial visualization systems is usually the missing availability of library systems.

Hence, this paper presents an approach for a model-based library system that supports HMI engineers in the creation of HMI models. As there is a variety of modeling languages currently used in MBUID, a generic solution that can easily be adopted for a specific modeling language or workflow is proposed. Therefore, general requirements are discussed in Sect. 2. Our generic approach towards a model-based library system is presented in Sect. 3 and a specific adaptation for the modeling language *Movisa* is demonstrated in Sect. 4. Section 5 discusses related work in the field of library systems for model-based approaches. Finally, conclusions and future work are presented in Sect. 6.

2 Requirements for Model-Based Library Systems

As library entries usually aggregate multiple HMI elements, their complexity can range from simple (e.g., symbolic representations of switches, valves, or lamps according to [4]) to complex (e.g., faceplates for the visualization of complete silos including multiple sensors). Consequently, a library system has to enable the persisting of such aggregated HMI objects as library entries. In contrast to a *normal* aggregated HMI object, the structure of the child objects inside a *library entry* is treated as immutable: If changes to the structure are to be made, a new library entry has to be created. The same applies to most attribute values, e.g., for the position of a contained object relative to the enclosing entry. Instead, a library entry offers only a dedicated interface that is made up of a set of *parameters*. The types of elements to be stored as library entries as well as the types of parameters that are necessary/useful depend on the concrete structure, capabilities, and semantics of the used HMI meta-model and have

to be predefined by the library system to support the definition of new library entries. For example, the library system may state that every entry must define exactly one parameter that allows the customization of the entry's size upon instantiation in an HMI model. Furthermore, every library entry usually can be equipped with additional metadata like a name or an ID, a version, a description, or a thumbnail in order to be identifiable by the user.

Currently, different modeling languages are used for MBUID depending, e.g., on the application domain, on prior knowledge, or on suitable tools [3]. Most of these languages base upon similar concepts and use similar modeling technologies, though. Thus, it is reasonable to define a generic data structure for the persisting of library entries that can easily be adapted to a specific modeling language or workflow.

A library system is also responsible for organizing the otherwise loose collection of various library entries into different (e.g. domain-specific) libraries and enable their integration with HMI tools and editors. Hence, library systems require management functions that allow to browse, add, remove, and instantiate library entries. In summary, a model-based library system has to comply with the following requirements:

1. A data structure has to be provided that allows to define and persist the different parts of a library entry:
   (a) the immutable structure of aggregated HMI objects and attribute values,
   (b) a dedicated parameter interface indicating modifiable elements,
   (c) metadata for the identification by a user or a model transformation
2. Library management algorithms have to enable a user or a model transformation to browse, add, remove, and instantiate library entries

3 Generic Model-Based Library System

This section introduces the concepts necessary for the realization of a generic model-based library system – the *GenLibrary*. Herein, the term generic describes the fact that the concepts presented are not tied to a single implementation for a specific modeling language. Instead, they are designed to be used and easily extended for any HMI modeling language and workflow. In this paper, we focus on the realization of a generic extensible data structure (cf. Sect. 2) – the realization of the management algorithms is not discussed in detail but briefly covered as part of the case study presented in Sect. 4.

HMI library entries consist of reusable parameterized visualization snippets comprising parts of an HMI description. In MBUID, the HMI description exists as an HMI *model*. Consequently, the library entries should be realized as parameterized *model snippets*, i.e. excerpts of an HMI model representing an aggregation of HMI elements. That way, the same tools and frameworks that are also employed for the realization of editors and model transformations for the selected modeling language can be used. We have created the meta-model that defines the data structure for these model snippets by means of the *Eclipse Modeling Framework*[3] (EMF) as it offers powerful tools, editors, and a simple possibility

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

to store and load models via the XMI[4] standard. EMF-based meta-models may consist of a set of classes (*EClass*) that hold a set of attributes (*EAttribute*) and a set of references (*EReference*) to other classes. References can be either *containment* (meaning that the referenced object is included in the referencing object) or *non-containment* (representing a cross-reference).

3.1 Simple Model Snippets

As every element of an EMF model (except the root element itself) must be a direct or indirect child of the root element via containment references, every model can be displayed as a containment tree with cross references. As described above, a library entry comprises a model snippet (a part of such a containment tree) as illustrated by Fig. 2: It shows a sample model (for a hypothetical HMI meta-model *ABC*) with a root element of type *AContainer*. The snippet in the dashed box shall be stored as library entry. It consists of an element of type *A* and two elements of classes *B* and *C* that are contained in *A* and coupled via a cross-reference. This could, e.g., represent a *text label* (*A*) that contains a *representation* (*B*) as well as an *animation* (*C*) that is referenced by the representation. The additional file *f* could represent an image file used by the representation.

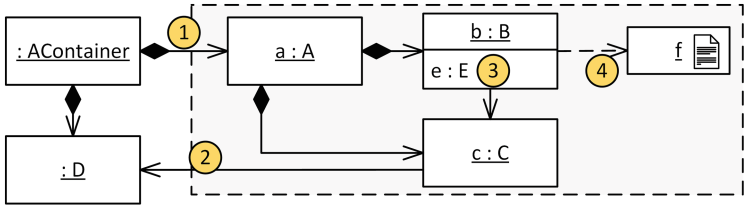

Fig. 2. Hypothetical sample library entry instance (dashed box) as part of a model containment tree

Because of the XMI standard, it is easily possible to create a copy of the element (and its child elements) and persist this model snippet as a library entry. The *GenLibrary* meta-model must however additionally enable the specification of metadata like an ID or a version (cf. Sect. 2). Therefore, it introduces the element *LibraryEntry* that represents one entry inside a library (cf. Fig. 3). In order to persist the actual model elements of a library entry (e.g. the structure presented in Fig. 2), an abstract base class *LibraryItem* has been defined that has to be extended by concrete implementations for a specific HMI meta-model. For the purpose of illustration, a simple example for such an implementation is represented by the hypothetical element *ABCLibraryItem* that allows to store instances of type *A* by directly referencing the respective element (EClass) from

[4] *XML Metadata Interchange*, http://www.omg.org/spec/XMI/.

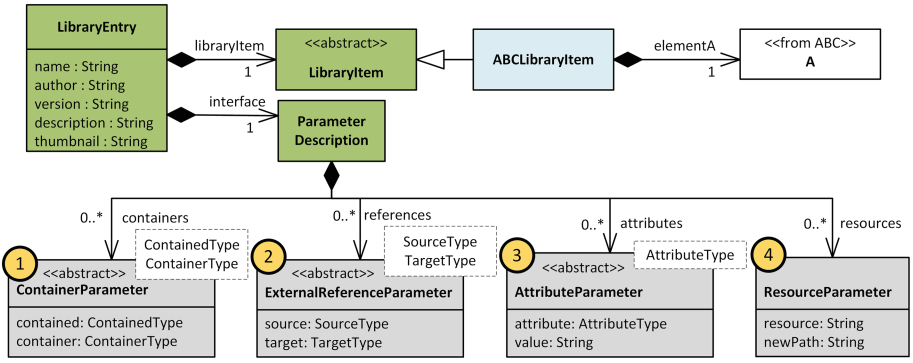

Fig. 3. *GenLibrary* meta-model and hypothetical concrete extension (upper right)

the corresponding HMI meta-model (in the example the EClass *A* from the meta-model *ABC*). Note that all contained elements (e.g. the elements *b* and *c* in case of the structure presented in Fig. 2) are stored as well automatically even if their types are not directly referenced by the *GenLibrary* meta-model.

This mechanism allows specific implementations to declare the type of elements that may be stored as library entries by referencing a specific HMI modeling language with its meta-model. In the case of MBUID, this will usually be elements representing simple widgets, complex widgets, or complete panels (as well as their children). If necessary, even multiple elements that are not contained within each other can be stored. This may be necessary if elements that reside at different parts of the HMI model but logically belong together shall be stored in the library item (e.g. a widget plus information about access rights).

3.2 Parameterized Model Snippets

As described in Sect. 2, a library entry has to provide an explicit specification of the parameters that may resp. have to be specified by the user or by a model transformation. The meta-model elements required for this specification can be abstracted from the tasks that have to be performed during the instantiation of a library entry (i.e. the insertion into a target model). These tasks can also be obtained from Fig. 2 if we consider the dashed box not as entry to be created but as existing entry that has been instantiated in a model.

The first and most obvious task is the insertion into the target model at a specific position in its containment tree. This is equivalent to the creation of a containment reference from an existing element in the target model (the new container object) to a top-level element of the library entry (cf. symbol ① in Fig. 2). If multiple top-level elements exist as described above, a parent element has to be determined for each of these elements. The second task consists in the creation of non-containment references from the inserted library entry to

Table 1. Types of tasks executed during the instantiation of a library item and information necessary for the execution

Type of task	Necessary information/elements to be specified
① Creation of containment references	• Contained element (a top-level element inside the library entry)
	• Container element (part of the target model)
② Creation of external references	• Source element (part of the library entry)
	• Target element (part of the target model)
③ Changing of attribute values	• Attribute to be changed (part of the library entry)
	• New attribute value
④ Copying of resources	• Name of the resource (as stored in the library)
	• New path of the resource

existing elements of the target model (cf. symbol ②)[5]. Third, it can be desirable or necessary to change attribute values of internal elements of the library entry during the insertion process (cf. symbol ③). The last task is responsible for the copying of resources or files that are required by the library entry to a new location (usually inside the project that the target model belongs to) as indicated by the symbol ④ in Fig. 2.

As summed up in Table 1, each of these tasks requires specific information for its execution. Thus, an own parameter type representing each task is required. We have realized this by means of the *ParameterDescription* element (cf. Fig. 3) that is composed of the four abstract parameter types deduced above. The necessary information for each parameter type as of Table 1 is represented by corresponding attributes and references. The references are realized by means of *generic types* (indicated by the dashed boxes on the top-right corner of the parameters) which enabled a generic implementation of the execution of the four tasks as part of the management algorithms. Concrete implementations of these parameters are again dependent on the actual HMI meta-model. They have to be specified by extending one of the abstract parameter types and with that by binding the generic types to concrete types.

4 Case Study: Library System for Movisa

To illustrate a concrete adaptation of the introduced generic library concepts, an implementation for the HMI modeling language *Movisa* [5] is shown in this section. Movisa is a domain-specific language created for the description of HMIs in industrial automation. A simplified subset of its meta-model is shown in the

[5] Non-containment references from the target model to elements of the inserted library item are also possible but not part of the insertion process of the library item.

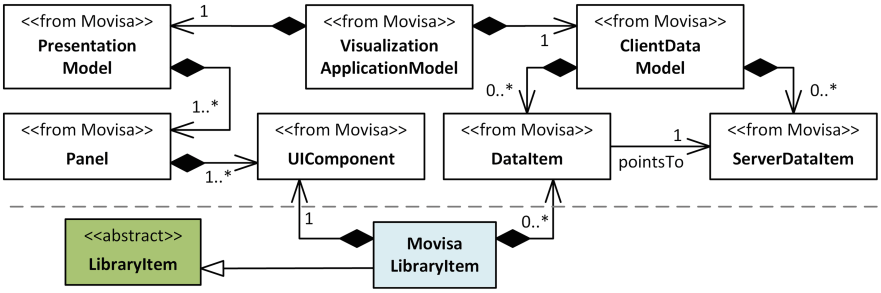

Fig. 4. Top: Simplified subset of the *Movisa* meta-model (cf. [5]); Bottom: Specification of the *MovisaLibraryItem*

top of Fig. 4. The top-level element (*VisualizationApplicationModel*) contains two sub-models: The *PresentationModel* divides the visual aspects of an HMI into *Panels* which consist of multiple HMI elements (*UIComponents*), e.g. containers, buttons, and text labels. The *ClientDataModel* aggregates variables in a collection of *DataItems* (logical viewpoint) that can be used to, e.g., animate the UI components. Therefore, every variable can be updated via a process data server. This is realized by means of the *ServerDataItems* (technical viewpoint).

Information from both sub-models is necessary for a widget to be functional. Therefore, a library implementation for Movisa has to be able to persist library entries containing (1) a UIComponent (that may contain aggregated components) and (2) associated DataItems. Therefore, we have created an extension of the *GenLibrary* meta-model. This Movisa-specific extension introduces a concrete implementation of the abstract *LibraryItem* – the *MovisaLibraryItem* (cf. Fig. 4, bottom). In order to enable the persisting of the two parts described above, the MovisaLibraryItem provides corresponding references to the Movisa meta-model. As the concrete process data server should not be specified in a library entry (because the entry should be independent of a specific communication technology) only the *DataItems* are saved without the associated *ServerDataItems*. Consequently, a parameter has to be created that allows to specify the server information upon instantiation of a library entry. As this is performed via the creation of a *non-containment reference* to a concrete *ServerDataItem* (reference *pointsTo* in Fig. 4), an *ExternalReferenceParameter* (cf. Fig. 3) has to be used. This is implemented by the *ServerDataItemParameter* shown in Fig. 5 which needs to be defined for each DataItem of a library entry. The parameter type therefore binds the *SourceType* to elements of the type *DataItem*. Similarly, the *TargetType* is bound to elements of the type *ServerDataItem* so that only elements of this type can be specified as target of the *pointsTo* reference to be created upon instantiation of the library entry.

Another parameter required for the insertion of a library entry into an existing HMI model is the *UIComponentContainerParameter* (cf. Fig. 5) that allows the container specification for the library entry, i.e. the selection of the panel that the UIComponent inside the entry should be displayed in. In contrast to

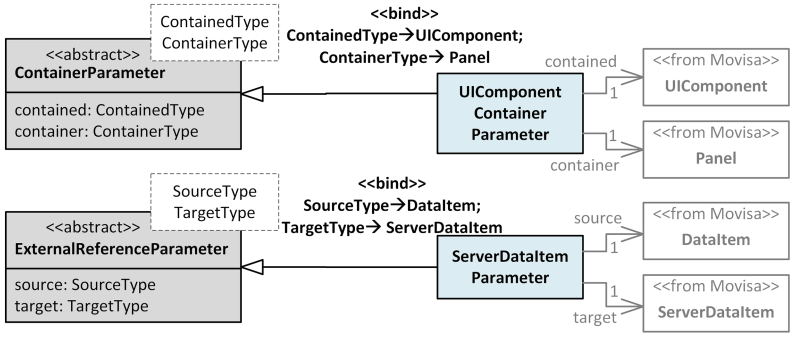

Fig. 5. Subset of the specification of the Movisa-specific parameters (The grayed out references are only shown to emphasize the result of the type binding.)

the UIComponents that can be displayed in one of the multiple panels defined in a Movisa model, the *DataItems* are always located at the same spot in its containment tree (directly below the single instance of *ClientDataModel*). Thus, we have implemented the insertion of the DataItems as part of the specific extension of the library system's management algorithms in order to reduce the number of container parameters necessary for each library entry.

Besides the extension of the meta-model and the management algorithms, we have also integrated the set of generic wizards that is offered by our library system into the existing Movisa editors. These support the user in the manual application of the library system. Figure 6 illustrates the import wizard that supports the user in the insertion of a library entry into an existing Movisa model[6]. Once all parameters have been specified, the instantiation of the selected library entry can be executed. The parameters specified in the wizard are used to insert the library item into the target model, create the non-containment references, and resize and position the UIComponent on the selected Panel. Figure 7 shows the result of this process: The elements inserted during a single instantiation are shown in a tree view of the Movisa model. It can be seen that the UIComponent (represented by a *SimpleContainer*) has been inserted into the panel *NAVTOP* and the DataItems have also been added to the ClientDataModel.

An example for the application of the library system in a more complex MBUID workflow is presented in [6]. It demonstrates the automatic generation of HMI models from a more abstract model via a model-to-model transformation. In such a case, a missing library system requires the specification of the entire target model only with the help of simple built-in elements. All knowledge about aggregated elements has to be implicitly represented in the transformation rules that get unnecessarily complex. The library system enabled a significant simplification of the transformation rules and ensures a consistent look and feel by

[6] Another similar wizard allows to export existing HMI elements as a new library entry but is not shown due to reasons of compactness.

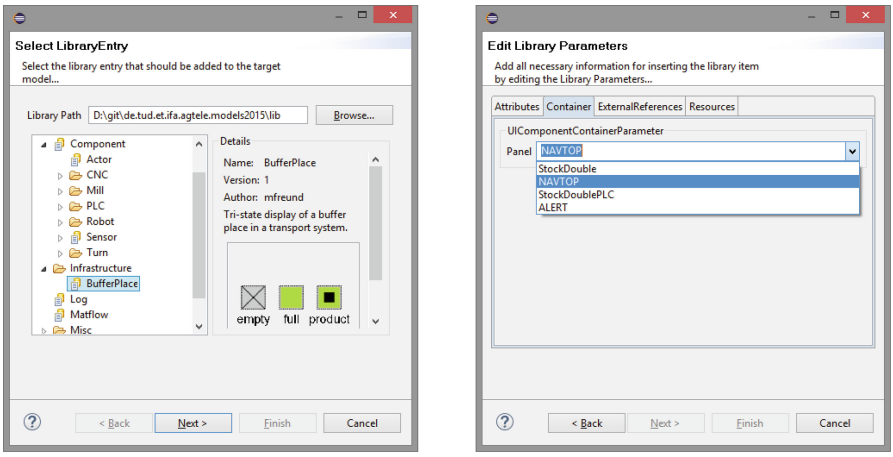

Fig. 6. Specification of the *MovisaParameterDescription*

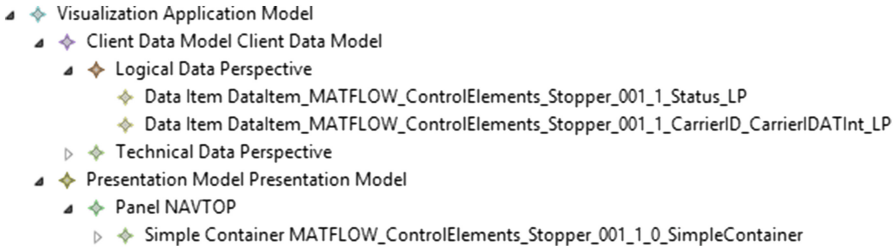

Fig. 7. Result of the insertion of a library entry into a Movisa model displayed in the model containment tree

using the already available domain-specific library entries and the corresponding management algorithms for their insertion.

5 Related Work

As discussed in Sect. 2, a library system has to implement two main functionalities: (1) the provision of a data structure to persist parameterized library entries and (2) library management algorithms. Most modeling languages currently used for MBUID like *UsiXML* [1] or *MARIA* [2] neither offer any possibilities to store parameterized model snippets separate from their enclosing model nor do they provide any library management algorithms. The modeling language *UIML* introduces "a template mechanism that allows defining reusable components" [7] that can be parametrized. However, this constitutes a special implementation for a single HMI meta-model and not a generic solution as targeted by our approach. As UIML does not allow the modeling of process communication because it does

not introduce a data model [8], this UIML-specific library concept is not suited very well for industrial automation. Furthermore, the definition of metadata and storage concepts are not discussed.

While generative UI patterns also enable the instantiation of reusable HMI parts, the concrete realization of a pattern differs depending on its problem context and thus for each concrete instance (cf. [9]). Hence, the structure of the HMI parts described by a pattern has to be flexible – e.g. concerning concrete contents of (sub-)elements – in contrast to the immutable structure of a library item. Consequently, generative UI patterns cannot be used for the description of library entries. However, both approaches could be combined, e.g. by using library entries to realize the concrete contents of a pattern upon instantiation.

Apart from the domain of MBUID, there are some approaches to use XML dialects for the realization of data structures that represent library entries including metadata and parameters. The most promising approach is *AutomationML* [10] that is used as an exchange format in industrial automation. Based on the data format *CAEX* [11], it offers a library concept that allows to define some metadata (e.g. an ID, a version, and an author) and an inheritance concept for library entries. However, *AutomationML* covers a very broad domain (including the description of 3D data and PLC logic) but does not incorporate an HMI meta-model. Thus, a distinct implementation for the domain of HMI libraries based on the introduced concepts seems beneficial.

In the context of software engineering, *(model) templates* allow to define parameterizable elements or models (cf., e.g., [12–14]) that could be used for the definition of library entries. However, most of those approaches consider templates only at the meta-model level resp. the level of UML class diagrams [14]. While the application at the *model* level is proposed in [14], specific extensions that are dependent on the used HMI meta-model are not discussed. Thus, the concept cannot be used to enforce the description of an immutable structure or a dedicated parameter interface specific for the used meta-model. Similar to templates applied at the model-level, *declarative transformation languages* like, e.g., *QVT* or approaches based on *graph transformations* (cf. [15] for an overview of both technologies) allow to define mappings that consist of a left-hand side, a right-hand side, and a relation between both sides. Such a mapping could be treated as a library entry where the right-hand side defines the element structure to be created and the mapping including the left-hand side implicitly defines a parameter interface (those elements that need to be present in the source model and that are applied to the right-hand side). However, just like the model templates described above, these approaches do not allow to restrict the structure inside a library entry and its parameter interface as they are dedicated to another application context.

6 Conclusions and Future Work

In this paper, we have presented an approach for a model-based library system for HMI development that allows the definition, management, and instantiation

of aggregated, reusable HMI objects. Our proposed generic solution enables an extension for a variety of modeling languages. In order to use the library system with a new modeling language, the following steps have to be performed:

1. Analysis of the used HMI meta-model concerning types of elements to be stored in a library entry and parameters to be influenced by the user
2. Extension of the *GenLibrary* meta-model by realizing a concrete subclass of *LibraryItem* and by defining concrete parameter types
3. Extension of the generic management algorithms and wizards to add HMI language-specific behavior resp. integrate them in existing editors and tools

While the generic approach provides a good starting point for extensions, the steps described above still demand considerable work. We have illustrated this by means of an adaptation for the modeling language *Movisa* that has been evaluated in the *AutoProBe* project (cf. [6]). As the library system can be applied for manual application as well as to model-based workflows, it should promote the acceptance of model-based approaches in industrial automation. In the future, we plan to evaluate our approach by creating extensions for additional modeling languages like UsiXML or MARIA.

An important challenge is the support for both the evolution and the aggregation resp. inheritance of existing library entries. While the presented wizards can help with this challenge (import library entries to a model, change or combine them, and create a new entry via an export), our solution does not allow to directly reference other library entries.

As an instantiated library entry is represented by standard elements of the used HMI meta-model, our approach enables existing model transformations to be applied (e.g., to produce an executable HMI). However, the downside of this is that it becomes hard to distinguish between *normal* and *library-based* elements. This makes it complicated to, e.g., completely remove an inserted library entry. In the future, this could be solved by a tracing mechanism.

Acknowledgments. The authors would like to thank *Christian Petzka* for his contributions.

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 by the "Deutsche Forschungsgemeinschaft" (DFG).

References

1. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: a language supporting multi-path development of user interfaces. In: Feige, U., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005)

2. 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)
3. Meixner, G., Paternò, F., Vanderdonckt, J.: Past, present, and future of model-based user interface development. i-com **10**(3), 2–11 (2011)
4. ISO 7000: Graphical symbols for use on equipment (2014)
5. Hennig, S.: Design of Sustainable Solutions for Process Visualization in Industrial Automation with Model-Driven Software Development, 1st edn. Jörg Vogt Verlag, Dresden (2012)
6. Martin, C., Freund, M., Braune, A., Ebert, R.E., Pleow, 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). IEEE (2015)
7. Helms, J., Schaefer, R., Luyten, K., Vermeulen, J., Abrams, M., Coyette, A., Vanderdonckt, J.: Human-centered engineering of interactive systems with the user interface markup language. In: Seffah, A., Vanderdonckt, J., Desmarais, M.C. (eds.) Human-Centered Software Engineering. Human-Computer Interaction Series, pp. 139–171. Springer, London (2009)
8. Trewin, S., Zimmermann, G., Vanderheiden, G.: Abstract user interface representations: how well do they support universal access? In: Proceedings of the 2003 Conference on Universal Usability, CUU 2003, NY, USA, pp. 77–84. ACM, New York (2003)
9. Seissler, M., Breiner, K., Meixner, G.: Towards pattern-driven engineering of run-time adaptive user interfaces for smart production environments. In: Jacko, J.A. (ed.) Human-Computer Interaction, Part I, HCII 2011. LNCS, vol. 6761, pp. 299–308. Springer, Heidelberg (2011)
10. Draht, R. (ed.): Datenaustausch in der Anlagenplanung mit AutomationML. Springer, Heidelberg (2010)
11. IEC 62424: Representation of process control engineering - Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools (2008)
12. Kulkarni, V., Reddy, S.: Separation of concerns in model-driven development. IEEE Softw. **20**(5), 64–69 (2003)
13. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G.: On some properties of parameterized model application. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 130–144. Springer, Heidelberg (2005)
14. de Lara, J., Guerra, E.: Generic meta-modelling with concepts, templates and mixin layers. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 16–30. Springer, Heidelberg (2010)
15. Taentzer, G., Ehrig, K., Guerra, E., De Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varro, D., Varro-Gyapay, S.: Model transformation by graph transformation: a comparative study. In: Workshop Model Transformation in Practice at MODELS 2005 (2005)