User Driven Evolution of User Interface Models — the FLEPR Approach

Stefan Hennig[1], Jan Van den Bergh[2], Kris Luyten[2], and Annerose Braune[1]

[1] Institute of Automation
Technische Universität Dresden
01062 Dresden, Germany
{stefan.hennig, annerose.braune}@tu-dresden.de

[2] Hasselt University — tUL — IBBT
Expertise Centre for Digital Media
Wetenschapspark 2, B-3590 Diepenbeek, Belgium
{jan.vandenbergh, kris.luyten}@uhasselt.be

Abstract. In model-based user interface development, models at different levels of abstraction are used. While ideas may initially only be expressed in more abstract models, modifications and improvements according to user's feedback will likely be made at the concrete level, which may lead to model inconsistencies that need to be fixed in every iteration. Transformations form the bridge between these models. Because one-to-one mappings between models cannot always be defined, these transformations are completely manual or they require manual post-treatment.
We propose interactive but automatic transformations to address the *mapping problem* while still allowing designer's creativity. To manage consistency and semantic correctness within and between models and therefore to foster iterative development processes, we are combining these with techniques to track decisions and modifications and techniques of *intra-* and *inter-model* validation. Our approach has been implemented for abstract and concrete user interface models using Eclipse-based frameworks for model-driven engineering. Our approach and tool support is illustrated by a case study.

Keywords: User interface models, model transformations, interactive model transformations, model consistency, model synchronization

1 Introduction

Industrial and particularly safety critical systems have to meet extensive requirements since the cost of failure is high and might result in loss of life. Thus, also the *User Interfaces* (UI) to operate such a industrial system need to be clear and without ambiguities with respect to defined behavior of the system. Specifically in the industrial automation domain, *Visualization Systems* enable factory-trained operators to monitor the operative states of automation systems as well as to operationally intervene in the process.

Since these operators should react appropriately to exceptions in the technical process even in extreme situations, they are required to give an early feedback to the system design. Therefore, a *User-centered Design* (UCD) [17] process is recommended. UCD is an iterative process: Ideas and concepts of the UI will be concretized into prototypes that are tangible for end-users in an early point of time. According to the end-users' feedback after evaluation, the prototype will be improved or modified and thereafter be evaluated again. Fig. 1 gives an overview of this iterative process.

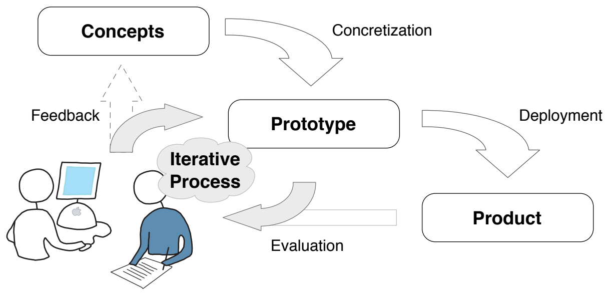


Fig. 1: Overview of the *User-centered Design* procedure.

Moreover, *Model-based User Interface Development* (MBUID) procedures have established to ensure the quality, usability, and sustainability of interactive applications. Since visualization systems are interactive applications [24], it is recommended to apply the methods proposed by MBUID: Basically, it uses models at different levels of abstraction, whereas moving from more abstract models to more concrete models means adding new information related to the UI design of the particular level of abstraction. This concretization is achieved by transformations. These are completely manual or require manual post-treatment because one-to-one mappings between models cannot always be defined.

User-centered design processes are about continuous improvement of user interface prototypes, already in early development stages. In MBUID, these modifications can take place at a concrete level where the UI is getting more concrete. However, altering UI models at the concrete level might not influence the abstract models, which leads to model inconsistencies. Therefore, we can identify two main challenges which arise if we want to combine MBUID with UCD: (1) *Ambiguities during transformation* and (2) *Model Consistency*.

In this paper, we propose a *Flexible Workflow for early User Interface Prototyping* (FLEPR) — an approach to resolve ambiguities during transformation interactively, thus keeping the user in control over design decisions. FLEPR combines interactive transformations with *intra-* and *inter-model* validation and with techniques to track decisions and modifications. Therefore, FLEPR combines UCD with MBUID by keeping the models consistent and preserving their semantic correctness at any time. The FLEPR is supported by a proof-of-concept tool. This tool was used in a small project to show the feasibility of this approach.

2 Problem Classification

UCD as an iterative process advocates the use of tangible prototypes in order to involve end-users in the design process as early as possible. These prototypes will be altered according to end-users' feedback after an evaluation — and then be evaluated and be improved again. This is what the term *evolution* implies — modifications to artifacts at all stages of software development [13]. Once the end-users' needs are satisfied, the prototype can be provided with complete functionality and deployed as a product. Fig. 1 illustrates the UCD process. The input of end-users is of great importance to create an interactive system that is well received by and tailored for the target end-user group. However, for industrial automation (and by extension safety critical systems), we need to be wary of inconsistencies between user preferences and required system behavior. After all, a correct operation of the system is more important in this case.

The *Cameleon Reference Framework* [2] defines development steps for *Model-based User Interface Development* (MBUID) by introducing UI models at different levels of abstraction. The *Final UI* level captures UIs ready for execution on a particular platform (see *Product* in Fig. 1). A *Concrete UI* level abstracts the *Final UI* from a particular platform. Movisa (see Section 3.2) is our representative for this level. It defines a high-fidelity concrete syntax which is tailored to the needs of automation engineers. Therefore, it corresponds to the prototype level of UCD depicted in Fig. 1 — an evaluation by end-users can be provided. *Abstract UI* definitions form the next level of abstraction proposed by the Cameleon Reference Framework. *CAP3* (see Section 3.1) forms an appropriate realization. It provides a concrete syntax which supports interaction designers in their work (see *Concepts* in Fig. 1).

Transformations are the key to progress from one model to another in MBUID. Three types of transformations have been defined in [30]: (1) *Abstraction* creates a more abstract model from a given model, (2) *reification* concretizes a model, and (3) *translation* produces models at the same level of abstraction. Another taxonomy, introduced by Mens et al. [16], distinguishes in that context *vertical* (abstraction, reification) and *horizontal* (translation) transformations. Furthermore, they define an additional dimension: A transformation is *endogenous* if source and target models are compliant to the same metamodel; it is *exogenous* if they are compliant to different metamodels. In [3], the authors refer to transformations as mappings and explicitly include human intervention during the mapping as a possibility. This has not been fully explored in MBUID literature to the authors' knowledge. For resolving ambiguities while ensuring consistent system behavior, human intervention during transformations is inevitable.

When combining MBUID and UCD for the problem at hand, the starting point is a CAP3 model (see Fig. 2) situated on the abstract UI layer. Using a vertical, exogenous transformation, a CAP3 model will be *refined* into a Movisa model (① in Fig. 2) whereas unambiguous one-to-one mappings cannot be ensured. According to the end-user feedback, the Movisa model will be modified or improved using horizontal, endogenous transformations (② in Fig. 2). In other words, the Movisa model will be *refactored*. Refactoring the Movisa models might lead to inconsistencies with the CAP3 model. Resolving these issues requires *Model Synchronization* (③ in Fig. 2, see also dashed arrow in Fig. 1). According to Ivkovic and Kontogiannis [13], model synchronization is “[...] the process of establishing an equivalence between two models when one of them is altered.”

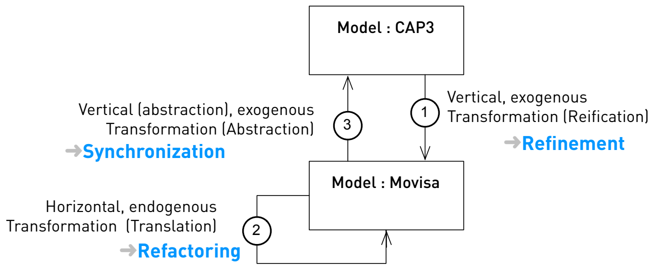

Model : CAP3
Vertical (abstraction), exogenous
Transformation (Abstraction)
➜Synchronization
3
1
Vertical, exogenous
Transformation (Reification)
➜Refinement
Model : Movisa
Horizontal, endogenous
Transformation (Translation)
2
➜Refactoring

Fig. 2: Overview of the different types of transformation.

The aim of this paper is to combine the *User Centered Design* process and *Model-based User Interface Development* techniques complying with the *Cameleon Reference Framework*. FLEPR enables a flexible workflow that solves ambiguous mappings during model refinement by means of interactive transformations (① in Fig. 2). It enables incorporating end-user feedback into concrete models and keeps track of these manual model refactorings (② in Fig. 2). As this can cause inconsistencies with abstract models, FLEPR provides model synchronization means (③ in Fig. 2. After explaining the basics of both CAP3 and Movisa, the following sections discuss our FLEPR approach on a conceptual and on a technical basis. A case study proves its feasibility.

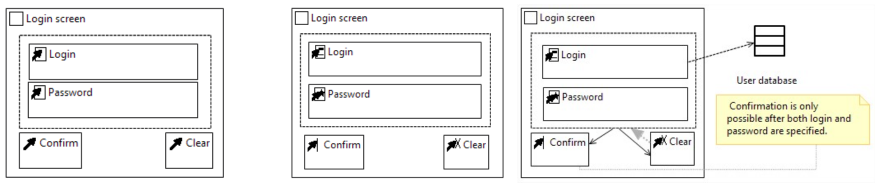
3 Background

This section gives a brief introduction of the used modeling concepts *CAP3* (Section 3.1) and *Movisa* (Section 3.2).

3.1 CAP3

CAP3 is a modeling language for abstract user interface models that provides both a concrete syntax (a graphical notation) and an abstract syntax (meta-model). Its concrete syntax is based upon the *Canonical Abstract Prototypes* notation [4] (CAP). CAP provides a number of abstract UI components to describe the structure of a (graphical) user interface in a way that is independent of any concrete toolkit or even modality. There are three main UI components: *tool*, *container* and *active material*. A *tool* allows a user to trigger a change in the UI or in the functional core (e.g. a button), a *container* can hold data (e.g. a label, image) or any other UI component (e.g. a window), and an *active material* is a combination of both previous components and thus can both hold data and trigger a change in the UI, data or functional core (e.g. a textfield, the Microsoft Ribbon). There are many other UI components besides these three that have much richer semantics and visuals, but are entirely optional as they are special cases of them.

The difference is illustrated in Fig. 3. Fig. 3a only uses the three basic UI components to specify a login dialog: A user can enter a login and a password as well as confirm or clear this information. Fig. 3b shows the same login dialog, but this time using richer UI components where appropriate. It shows that the login can be selected from a set of available options (using a *selectable collection*) and that the password has to be entered from scratch each time (using an *input*). The confirmation ends the activities in this dialog (using an *end component*) and visually confirms that *Clear* removes the data in the interface. The dashed rectangle (*conceptual group*) around the login and password show that both UI components belong together.



(a) (b) (c)

Fig. 3: *CAP3* login window using (a) the basic CAP UI components; (b) richer CAP UI components suggesting possible options, and (c) UI components specifying behavior and relations to the domain model.

Both versions of the login dialog, however, do not specify the relations between components. For example, they do not specify that confirmation can only be done when both login and password are specified or that the *clear* tool clears both the login and password. This behavior can be specified by using the additions made in CAP3; it adds the capability to specify behavior and relations to the domain model or functional core, as can be seen in Fig. 3c. It specifies that *Confirm* and *Clear* are only available after the login and password are specified using the *enable* relation. The fact that *Clear* clears both the login and password is shown using the *update* relation (thick gray dotted arrow). Fig. 3c also shows that the logins are fetched from the *user database*. Note that the *conceptual group* is used to reduce the number of arrows; all relations that are connected to a *conceptual group* can be replaced by relations to all its contained components.

3.2 Movisa

Movisa is a *Domain Specific Language* (DSL) to create models for technology independent development of interactive systems in industrial automation that require complex graphical representations (e.g. visualization of a plant process). This domain has certain requirements for the UI components, the behavior of those components, and the protocols to gather the data that influence this behavior. A Movisa compliant model contains three sub-models (see Fig. 4): The *Presentation Model*, the *Algorithm Model*, and the *Client Data Model*.

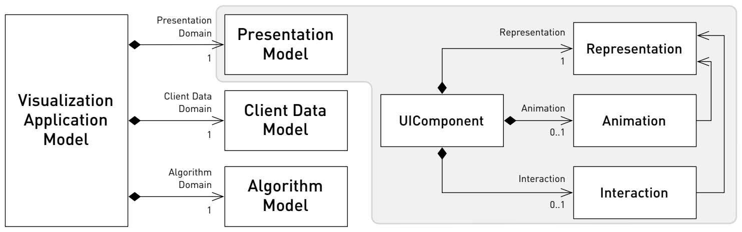


Fig. 4: Excerpt of Movisa's metamodel, the relevant part for this paper is the *Presentation Model*, which defines UI components.

As shown in Fig. 4, the *Presentation Model* defines *UI Components* whose properties where classified in the following three categories: (1) The *Representation* category defines the initial appearance of a UI component on the screen. It encapsulates representations for properties such as *Position*, *Size*, or *Border Color*. (2) An *Animation* category captures properties defining the dynamic behavior of UI components in order to reflect the current state of the process. Thus, properties of the *Animation* influence properties of the *Representation* during runtime. (3) The *Interaction* category allows to configure the user's interaction means as well as the resulting actions. Based on a survey of conventional visualization systems for industrial automation, we deducted that these properties — and the relations between them (see Fig. 4) — meet the requirements of *industrial automation*.

Conventional visualization systems provide scripting environments in order to enable developers to integrate application specific logic. This is what the *Algorithm Model* provides in a technology independent manner using a customized *Executable UML* realization. *Executable UML* is a computationally complete abstract software specification or modeling language.

Movisa's *Client Data Model* contains modeling elements dedicated to configure communication parameters. Basically, these parameters depend on concrete data server specifications. The *Client Data Model* therefore provides an adapter-like architecture: An adapter for each data server specification allows fine-grained parameterization so that reliable communication relationships with any automation specific data server can be ensured. A common information model provides these data to the elements of the *Algorithm Model* and to the elements of the *Presentation Model*.

To illustrate Movisa's principles, we use the CAP3 model introduced in Section 3.1 and realize the *Selectable Collection* "Login" exemplarily on the concrete UI level. (This is the level of the *Cameleon Reference Framework* where Movisa resides; it has been discussed in Section 2.) Therefore, the *Presentation Model* defines a UI component *Drop Down*. This component has an appropriate *Animation Property* that is responsible to fill up the *Drop Down* component with available options from the "User database". An additional requirement may be not to show up the user name "Administrator". This can be realized with an appropriate set of actions — defined within the *Algorithm Model* — dedicated to remove this particular user name from the data set. Finally, the *Client Data Model* defines the required data items and the particular data server from which the data will be fetched.

4 FLEPR: UCD meets MBUID

In order to define an effective approach for the problem described in Section 2, we consider the following three perspectives: (1) A *User Perspective* examines the different users and roles involved in the UI development process; (2) a *Conceptual Perspective* derives an appropriate concept from the tasks of the user perspective; and (3) a *Technology Perspective* introduces the concrete technical realization of the derived concepts.

4.1 User Perspective

Users involved in the UI development process are characterized by different responsibilities and tasks, different knowledge and different skills. They are therefore assigned one-time to one of the following user roles:

Interaction Designer/Information Architect is familiar with the standards concerning usability (e.g. [8]), dialog design (e.g. [6]), and/or information representation (e.g. [7]) — with respect to safety critical systems. All of these can be described as common knowledge of the interface design; it is captured by conceptual models such as CAP3. This user role is therefore responsible for creating and maintaining the abstract UI model CAP3.

Domain Expert is an Application Engineer. She or he has advanced knowledge about the technical process to be monitored and operated by the visualization system. Furthermore, he or she is acquainted with the devices realizing the technical process and the relationships between them. Domain experts are additionally characterized by knowing the physical backgrounds, possible events and hazardous situations as well as the details on the product to be manufactured (e.g. product compounds). Movisa has been designed to capture specific knowledge of the automation domain. As this user role specifies these implementation details, it is mainly responsible for creating and maintaining the concrete UI model Movisa.

End User knows about the (manufacturing) process and the product to be manufactured. She or he also has knowledge of how to react to particular events. They are intended to work with the *product* of the user-centered design process. Therefore, users of this role are responsible for evaluating the UI prototypes.

Fig. 5 shows these user roles assigned to the particular development step (see Fig. 2) which the users are mainly responsible for.

There is, however, no clear border between the responsibilities of the participating user roles. Interaction designers might collaborate with domain experts at the CAP3 level. Domain experts might collaborate with interaction designers at the Movisa level. In any case, the users have deep knowledge about there belonging domain, but they are neither familiar with model transformations nor with details of the models itself. Therefore, they have to be supported appropriately by dedicated tools.

4.2 Conceptual Perspective

The basis we use for this Section is a CAP3 model initially developed by the responsible users (see Section 4.1). Starting from that, Fig. 2 points out the different kinds of transformation while Fig. 5 introduces particular user roles mainly responsible for each part

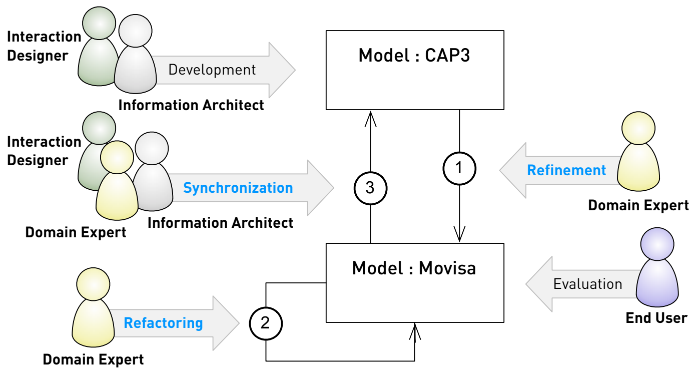


Fig. 5: Different types of users are involved in the UI development process at different development steps.

together with their particular task. Based on this, Fig. 6 introduces the overall concept which is composed by the following steps: ① *model refinement*, ② *model refactoring*, and ③ *model synchronization*.

Model Refinement (①). This is the development step which transforms a source model (CAP3) into a target model (Movisa) using a vertical, exogenous transformation. In this development step, an unambiguous one-to-one mapping between elements of the CAP3 model and elements of the Movisa model cannot be ensured. The following list presents several fundamental ambiguities we identified:

- ❶ CAP3's *Selectable Collection*[1] can be mapped to one of the following Movisa elements: (1) *Radio Button Group*, (2) *Check Box Array*, or (3) *Drop Down*.
- ❷ A CAP3 *Tool* can be mapped to a Movisa *Button* or to any other Movisa UI component with an appropriate *interaction property*.
- ❸ CAP3 defines an element *Notification* which can be transformed to Movisa's *Alarm-Widget* or any other Movisa UI component with an appropriate *animation property*.
- ❹ CAP3 introduces an *Update* relationship, however it makes no assertions which property of the particular Movisa UI component should be updated under which condition.

Clearly, these ambiguities cause problems because one element can be mapped to multiple alternatives. Decisions regarding these mappings are always subject to project specific requirements or assumptions, company guidelines, or even personal taste. In the best case, there is only a single one-to-one mapping possible avoiding all ambiguities. Nevertheless, these decisions influence the transformation process — it needs to be customized based on them. Since *Domain Experts*, as the users that steer this development step, are not familiar with transformations (see Section 4.1), the following four different approaches can increase accessibility of these transformations:

[1] A *Selectable Collection* can also be represented by a *Collection* and a separate (nested) *Selection Tool*.

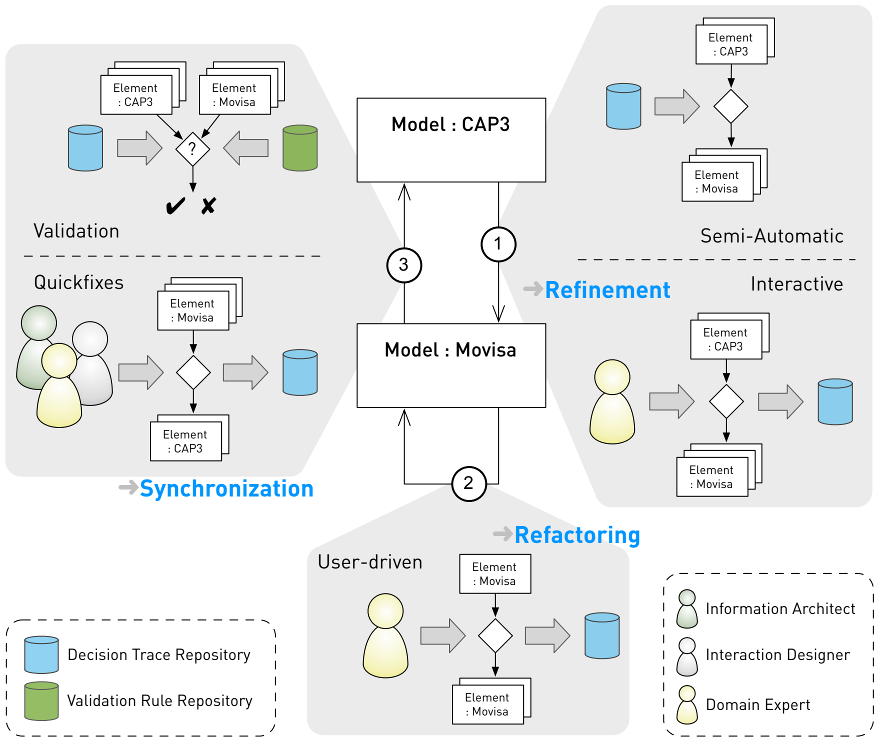


Fig. 6: The overall concept showing details about each development step — ① Model Refinement, ② Model Refactoring, and ③ Model Synchronization — using the transformations defined in Fig. 2. It also shows in what extend the users introduced in Fig. 5 are involved in the procedure.

(1) A default mapping is used within an automatic transformation process. The domain expert then needs to customize the resulting Movisa model subsequently according to project specific requirements.

(2) A separate *Mapping Model* can be used which captures the project specific mappings between elements. This concept has been introduced by the *Human Computer Interaction* (HCI) community, e.g. in [3]. Sottet et al. [26] have defined a metamodel to specify mappings between different kinds of models. Elements of a source model can be connected to elements of a target model on both the metamodel and the model level. UsiXML [29] defines a mapping model, too, which only provides mappings on the model level. According to Siikarla and Systa, transformations relying on this additional information are called *semi-automatic* since they require “the transformation engineer [to make] decisions that guide an [...] ambiguous automatic transformation” [25].

(3) The source model (CAP3) can be annotated in order to remove ambiguities in a semi-automatic transformation. For example, the *Eclipse Modeling Framework* (EMF) [27] provides an *EAnnotation* element which enriches arbitrary modeling

elements with additional information. XML introduces a similar mechanism with the `<annotation>` element. In both cases, such mechanism can be used to specify mappings between elements of two models.

(4) In contrast to the previous two approaches which require user interventions before the actual transformation, *interactive transformations* require the user to intervene only during the transformation process.

We argue that the last (4) approach is most suitable, despite the others being used more often in existing solutions. Defining mappings on metamodel as well as on model level (2) or annotating a model (3) requires deep knowledge of the model's abstract syntax. It cannot be assumed that the responsible user roles have this knowledge (see Section 4.1). Interactive transformations (4), however, do not expect users to have deep knowledge neither of abstract syntax nor of the transformation itself. While the user will guide the transformations, the transformations also have to guide the user; the more precise the descriptions of ambiguities are the more powerful is the procedure.

Model Refactoring (②). This development step uses a horizontal, endogenous transformation to modify or improve a model: Once the Movisa model has been derived by transformation, a manual post-treatment is necessary because information e.g. regarding an element's position or size are not part of the CAP3 model. Moreover, the role *end-user* comes into play: He or she evaluates the actual design and according to this she or he gives her or his feedback to the domain expert. It entails the following tasks:

(1) Improvements such as element's size or border color can simply be made using Movisa's model editor.
(2) Modifications such as changing a *Radio Button Group* to a *Drop Down* element while preserving the functional behavior is a more sophisticated task. It can be done using appropriate transformations. Kolovos et al. define them as *update transformations in the small*: They "are applied in a user-driven manner on model elements that have been explicitly selected by the user" [15].

Model Synchronization (③). This is the process to manage consistency between two or more models if one of these models has been altered: After the Movisa model has been modified due to end-user feedback, the CAP3 model has to reflect these modifications. As stated in Section 2, there are two classes of modifications, those which do not entail updates in the CAP3 model and those which do. For example, changing only the border color of a particular UI element in the Movisa model has no effect to the CAP3 model. Adding a further item to a *Drop Down* box requires an update of the CAP3 model. Otherwise, the consistency is violated.

According to Nuseibeh et al. [18], inconsistencies can be tolerated as long as they will be fixed in a future point in time. Therefore, we do not artificially constrain the domain expert in improving a model and treat consistency management in a subsequent validation procedure which is explicitly driven by the user (see ③, Fig. 5). Nuseibeh et al. [18] describe model consistency management basically with the tasks (1) detecting inconsistencies, (2) characterizing inconsistencies, and (3) handling inconsistencies. Following that scheme, our approach for model synchronization is a two-staged process:

Intra- and inter-model validation: The models are consistent if both syntactical and semantical correctness is guaranteed. While syntactical correctness appears if the model is compliant to its metamodel, the process of ensuring semantical correctness is subject to a check against a set of appropriate rules. Basically, these rules define to what extent the Movisa model is allowed to alter to be consistent to the CAP3 model. Additionally, semantical correctness has to be proven not only within one model but also between them.

Solving inconsistencies: If inconsistencies were identified by the preceding validation, members of the particular user role (see ③, Fig. 5) decide whether to roll back the corresponding modifications or to update the counterpart model. The latter option will be supported by user-driven update transformations. The system therefore suggests valid alternatives.

Traceability. Inter-model validation, as previously introduced, needs knowledge about which elements of both models are interrelated. It is desirable to be able to reproduce all previous decisions since models may contain hundreds of elements and therefore require hundreds of decisions. Thus, a transformation (①, Fig. 5) performed for a second time — after all other transformations (①, ②, and ③, Fig. 5) are executed — should automatically produce the model which was the result of a previously performed transformation (②, Fig. 5).

Czarnecki and Helsen [5] point out that "[t]ransformations may record links between their source and target elements" [5] which fosters synchronization between models. It also enables reproducibility of transformation results. They prefer to store these links separately which has been consolidated by Van Gorp who stated in [11]: "[T]raceability links should be treated as first class software artifacts."

4.3 Technology Perspective

Fig. 6 shows that the *Decision Trace Repository* is the core of the entire approach. It feeds the refinement transformation (①) in order to make all steps reproducible. It also supports the process of model synchronization (③) by providing information about how the elements of the CAP3 model are linked to the elements of the Movisa model. For that purpose, a model dedicated to capture these relationships should be provided separately, as [5] and [11] suggest. Czarnecki and Helsen [5] propose to add an additional GUID[2] to each model element (in our case elements of both the CAP3 and the Movisa model). The third model responsible for capturing the relationships between these models only establishes links between these GUIDs.

Fig. 7 shows the metamodel of our *User Decision Repository*, further on named *FleprMap*. As Section 4.2 claims, the *FleprMap* only establishes a relationship between source and target elements. Therefore, a particular *Mapping* points to the respective elements with the *Source* and *Target* references. These references can be of any type as long as they are derived from an *EClass*[3].

[2] *Globally Unique Identifier* (GUID).

[3] *EClass*: Metamodel entity of the *Eclipse Modeling Framework*.

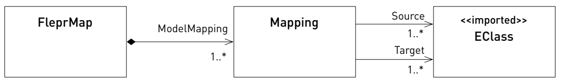


Fig. 7: Simplified metamodel of the FleprMap, our *Decision Trace Repository*.

Transformations guide the different users through the entire user interface development process. Mens et al. [16] distinguish mainly *declarative* and *operational* transformation mechanisms. While declarative approaches have their strengths in compactness and maintainability because they hide procedural details, operational approaches may unfold their advantages if incremental model updates are required [16]. Nevertheless, a suitable transformation approach has to deal with the following model management tasks (see Section 4.2): (1) transformation, (2) comparison, (3) validation, and (4) merging. There are different types of languages and transformation approaches each addressing only some of these tasks. For example, OCL[4] [21] is only for model validation; *openArchitectureWare* [20] captures besides validation only transformation. With its set of dedicated languages, the model management tool *Epsilon* [10,14] addresses all of the tasks. Furthermore, the Epsilon languages combine declarative and operational approaches. A further advantage is that it is tightly integrated into the *Eclipse* [9] platform, since both the CAP3 and the Movisa editors are distributed as *Eclipse* plugins. Epsilon's transformations also allow for user interaction. We used (1) the *Epsilon Transformation Language* (ETL) to realize the interactive transformations (derived in Section 4.2) to be applied during model refinement (see ①, Fig. 2), whereas the concrete mapping options are implicitly contained in the transformations; (2) the *Epsilon Wizard Language* (EWL) to foster user-driven model refactoring (see ②, Fig. 2) which has been introduced in Section 4.2; and (3) the *Epsilon Validation Language* (EVL) to apply a set of predefined validation rules in order to achieve model synchronization (see ③, Fig. 2) mentioned in Section 4.2.

Finally, the result of our investigation is a prototypical implementation of the aforementioned concepts (see 4.2) in the form of suitable Eclipse plugins realized with the technology introduced in this Section.

5 Case Study

Our laboratory plant is equipped with three tanks and pipes, pumps, or valves, respectively, between them. Fuel level sensors ensure that the water in the tank does not spill over; a programmable controller enables monitoring and operating. The requirement is to provide a user interface for observing the automatic process as well as enabling manual operator intervention.

Fig. 8 depicts the *Abstract UI* modeled in CAP3 using the CAP3 model editor. The UI contains four *Container* elements: (1) a top level container "Overview" contains all other UI elements, (2) an "Interaction" container captures interaction elements, (3) the container "Fuel level map" defines elements for monitoring the process, and (4) a

[4] Object Constraint Language (OCL).

Repeated Conceptual Group “Fuel Level Trends” providing appropriate trend charts. CAP3’s *Domain Objects* represent process variables.

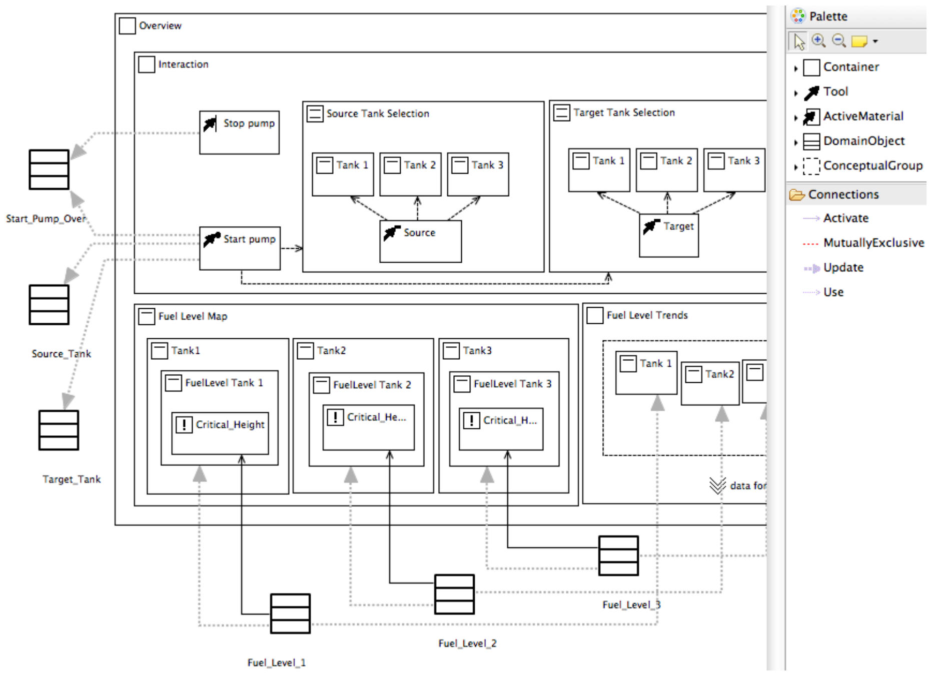
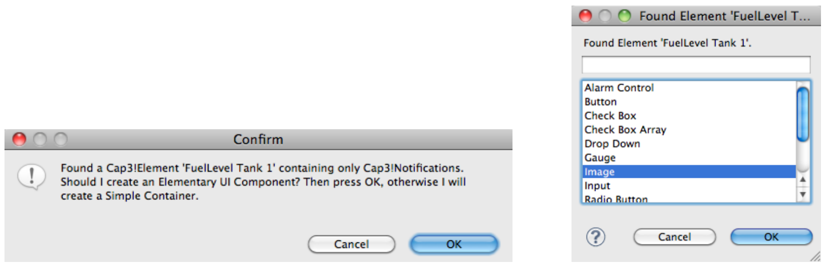


Fig. 8: Excerpt of the CAP3 model created in its dedicated model editor.

During a refinement transformation (see ①, Fig. 6) a first ambiguity to be resolved appears when detecting the *Element* “FuelLevel Tank 1” which is connected to a *Domain Object* via an *Update* relationship. The domain expert therefore has to decide which Movisa element is suitable to monitor a process variable and in which way it can be animated according to its current value. Fig. 9 illustrates this interactive task where the user chooses to create an *Image* element. The transformations store these decisions in the FleprMap, the mapping model. (If the user performs this transformation again, the transformations recognize that both models exist and that the links stored within the FleprMap are valid — in that case no user interaction is required.)

Once all ambiguities are resolved, a complete and valid Movisa model is available. Fig. 10a shows the entire model using Epsilon’s model tree view. The “Interaction” container, e.g., has been transformed to a *Simple Container* containing two *Buttons* in order to “Start” and “Stop” an inter-tank transfer process and two *Drop Down* elements dedicated to choose the source and the target tank. Fig. 10b shows the *Drop Down* elements in the Movisa model editor.

An evaluation points out that *Drop Down* elements are not suitable. Therefore, the domain expert invokes an appropriate EWL script directly within the model editor (see


Confirm

Found a Cap3!Element 'FuelLevel Tank 1' containing only Cap3!Notifications. Should I create an Elementary UI Component? Then press OK, otherwise I will create a Simple Container.

Cancel OK

Found Element 'FuelLevel T…

Found Element 'FuelLevel Tank 1'.

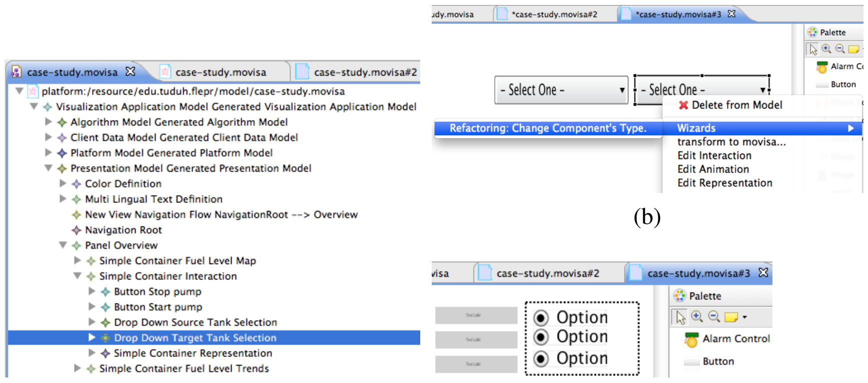
Alarm Control
Button
Check Box
Check Box Array
Drop Down
Gauge
Image
Input
Radio Button

Cancel OK

(a) (b)

Fig. 9: *Model Refinement:* (a) notifies the user about ambiguities to be resolved; (b) suggests possible options.

Fig. 10b) — this invokes a refactoring transformation. First, the user chooses the element to be created using the same procedure as provided for the model refinement process (see Fig. 9b). A transformation (1) creates this new model element, (2) adds it to the right position within the Movisa model, (3) ports all properties of the old element to the new one, (4) deletes the old element, and (5) updates the FleprMap.

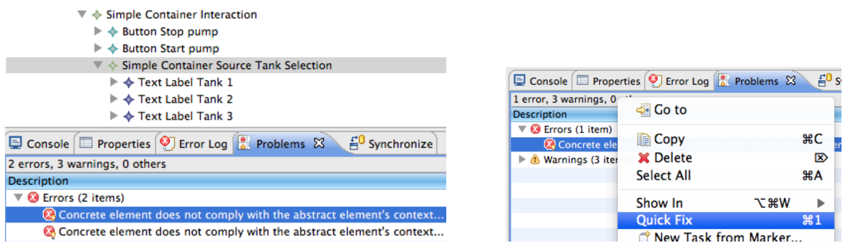

case-study.movisa | case-study.movisa | case-study.movisa#2

platform:/resource/edu.tuduh.flepr/model/case-study.movisa
Visualization Application Model Generated Visualization Application Model
Algorithm Model Generated Algorithm Model
Client Data Model Generated Client Data Model
Platform Model Generated Platform Model
Presentation Model Generated Presentation Model
Color Definition
Multi Lingual Text Definition
New View Navigation Flow NavigationRoot --> Overview
Navigation Root
Panel Overview
Simple Container Fuel Level Map
Simple Container Interaction
Button Stop pump
Button Start pump
Drop Down Source Tank Selection
Drop Down Target Tank Selection
Simple Container Representation
Simple Container Fuel Level Trends

(a) (c)

Fig. 10: *Model Refactoring*: (a) shows a compact tree view of the Movisa model after the refinement process; (b) shows an excerpt of Movisa's model editor where a user starts to refactor it; (c) depicts the result of the refactoring process.

Fig. 10c shows the resulting interaction elements. The first *Drop Down* element was refactored to a set of *Text Labels*, the second one to a *Radio Button Group*. A subsequent model validation, as part of a *Model Synchronization* process (see ③, Fig. 6), points out that the *Text Label* alternative has led to semantical incorrectness (see Fig. 11a). The reason is that the CAP3 model defines the *Tool* "Start pump" to initialize an inter-tank transfer process as an atomic task — the end-user has to set source and target tank before the particular process variables will be updated[5] using the *Tool*. Thanks to the FleprMap and to the validation rules, expressed in Epsilon EVL, the individual inconsistency can be resolved (see Fig. 11b) using the same procedure as provided for the model refinement process — clarify ambiguities interactively.



Simple Container Interaction
Button Stop pump
Button Start pump
Simple Container Source Tank Selection
Text Label Tank 1
Text Label Tank 2
Text Label Tank 3

Console | Properties | Error Log | Problems | Synchronize

2 errors, 3 warnings, 0 others
Description
Errors (2 items)
Concrete element does not comply with the abstract element's context...
Concrete element does not comply with the abstract element's context...

Console | Properties | Error Log | Problems

1 error, 3 warnings, 0
Description
Errors (1 item)
Concrete ele
Warnings (3 ite

Go to
Copy ⌘C
Delete
Select All ⌘A
Show In ⌥⌘W
Quick Fix ⌘1
New Task from Marker...

(a) (b)

Fig. 11: Problems View of the Eclipse tool: The validator has recognized semantical incorrectness between the CAP3 model and the altered Movisa model (a) and provides means to fix it (b).

The first step after selecting the *Quick Fix* (see Fig. 11b) is deciding whether to roll back the recent modifications or to update the CAP3 model. In this case study, the user chooses the latter option. Fig. 12 highlights the relevant parts of the updated CAP3 model introduced in Fig. 8.

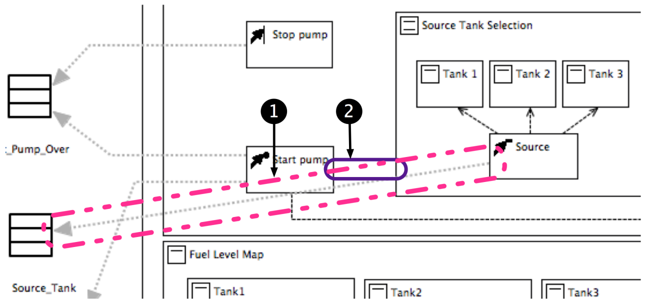


Fig. 12: Excerpt of the CAP3 model after Model Synchronizing.

Since the “Source Tank Selection” is now realized by a set of *Text Labels*, the user decides to let the particular *Text Label* update the process variable immediately after clicking it. (It does not need to be an atomic task.) This can be achieved using an appropriate interaction property defined in Movisa’s metamodel (see Section 3.2). A set of

[5] In industrial automation intervention in the process often requires to modify several process variables as an atomic task.

transformations update both the Movisa model by adding interaction properties to the *Text Label* components and the CAP3 model by moving the origin of the *Update* relationship from the *Tool* "Start pump" to the *Tool* "Source" (shape ❷ in Fig. 12). Since the *Tool* "Start pump" does not depend on the "Source Tank Selection" anymore, the transformations remove the *Use* relationship between the "Start Pump" *Tool* and the *Selectable Collection* (see shape ❶ in Fig. 12 compared to the same area of Fig. 8). Finally, these transformations update the FleprMap according to the latest modifications in the CAP3 model. All three models — the CAP3 model, the Movisa model, and the FleprMap — are consistent and semantically correct; together they describe a *User Interface* which is appreciated by end-users.

6 Related Work

Model-based UI development is most completely addressed by UsiXML [29], a sophisticated *XML-based User Interface Description Language*. It fosters the MBUID process at each level of abstraction by defining appropriate models and providing tools to reify abstract models. Moreover, the tool *ReversiXML*, proposed by Vanderdonckt et al. [31], can automatically reverse engineer web pages into UsiXML's CUI and AUI models. Another tool for *UI Reverse Engineering* is *VAQUITA* proposed by Bouillon et al. [1]. With it, one can create an XIML [22] *Presentation Model* from any web site. For that purpose, VAQUITA lets developers participate in the engineering process. The purpose of both tools is the migration of an existing UI to another context of use. Beyond that, Stroulia et al. [28] as well as Ramón et al. [23] use reverse engineering methods to extract the knowledge included in legacy UIs into models for further treatment. The presented approaches have in common, that they always create the more abstract UI model from scratch. However, aspects of *Model Synchronization* are desirable, too: Information that is encapsulated in abstract models but not part of concrete models needs to be preserved.

Model synchronization, also referred to as *Model Inconsistency Management*, is addressed by *xlinkit* [19] which uses *XLink* [32] to establish relations between elements of XML based models. *CAViT* [12] is a consistency maintenance framework that binds particular transformation rules to OCL invariants. OCL checks if consistency constraints are satisfied; the particular transformation rule can resolve a possibly identified inconsistency. While the former approach requires XML based models, the latter one relies on OCL. Both prerequisites are not addressed by Eclipse based approaches.

7 Conclusion & Future Work

We investigated to what extend the User-centered Design process can be combined with the Model-based User Interface Development process. A prototype Eclipse implementation using different types of (interactive) transformation and intra- and inter-model validation in combination with a model dedicated to trace decisions, proves our approach to be feasible. It provides an iterative development procedure where modifications can take place at each level of abstraction defined by MBUID. A subsequent model synchronization ensures consistency during this process. Since the methods introduced in this paper enable a forward engineering as well as a reverse engineering, our approach provides a *Round Trip Engineering* in MBUID. While Clerckx et al. [3] state that "[t]he aggregation of all of the abstract and concrete models is called the interface model", we have also to assign our *FleprMap* model to be that part of the *interface model* which ensures consistency.

Future work should be dedicated in a first instance to the usability during the development process. For example, the transformation environment should highlight the particular element if ambiguities have been identified, so that users can reason about the element's context in order to make the right decision. Future work should also be dedicated to maintainability of the transformation environment. The validation rules, for example, could be made explicit with the following consequences: (1) The transformation environment will be more generic and (2) the validation rules can simply be enhanced without knowledge of the transformation itself.

Acknowledgments

The authors would like to thank *Henning Hager* and *Pieter Van Gorp* for their contribution to this paper in the form of fruitful discussions. Part of this work is supported by the FWO project G.0296.08 and by the ESF project 080904707 (together with the Free State of Saxony).

References

1. Laurent Bouillon, Jean Vanderdonckt, and Kwok Chieu Chow. Flexible re-engineering of web sites. In *Proceedings of the 9th international conference on Intelligent user interfaces*, 2004.
2. Gaëlle Calvary, Joelle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 2003.
3. Tim Clerckx, Kris Luyten, and Karin Coninx. The mapping problem back and forth: customizing dynamic models while preserving consistency. In *Proceedings of the 3rd annual conference on Task models and diagrams*, 2004.
4. Larry L. Constantine. Canonical abstract prototypes for abstract visual and interaction design. In *Interactive Systems. Design, Specification, and Verification*, Lecture Notes in Computer Science, pages 1–15. Springer, 2003.
5. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
6. EN ISO 9241-10: Ergonomic requirements for office work with visual display terminals (VDTs) — Part 10: Dialog principles, 1996.
7. EN ISO 9241-8: Ergonomic requirements for office work with visual display terminals (VDTs) — Part 8: Requirements for displayed colours, 1998.
8. EN ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) — Part 11: Guidance on usability, 1999.
9. Eclipse Website. http://www.eclipse.org.
10. Epsilon Project Website. http://www.eclipse.org/gmt/epsilon.

11. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Traceability and Fine-Grained constraints in interactive inconsistency management. In *Tor Neple, Jon Oldevik and Jan Aagedal (editors), Second ECMDA Traceability Workshop (ECMDA-TR 2006)*, 2006.
12. Pieter Van Gorp and Dirk Janssens. CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In *Transformation Techniques in Software Engineering*, 2005.
13. Igor Ivkovic and Kostas Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004.
14. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Epsilon development tools. In *Eclipse Summit 2006*, 2006.
15. Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack, and Louis M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 2007.
16. Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 2006.
17. Donald A. Norman and Stephen W. Draper. *User Centered System Design: New Perspectives on Human-computer Interaction*. 1986.
18. Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 2001.
19. Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. ViewPoints: meaningful relationships are difficult! In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
20. openArchitectureWare Project Website. http://www.eclipse.org/workinggroups/oaw.
21. Object Constraint Language. http://www.omg.org/spec/OCL/.
22. Angel Puerta and Jacob Eisenstein. XIML: a common representation for interaction data. In *Proceedings of the 7th international conference on Intelligent user interfaces*, 2002.
23. Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010.
24. Thomas B. Sheridan. Supervisory control. In *Handbook of Human Factors*. 1987.
25. Mika P. Siikarla and Tarja J. Systa. Decision reuse in an interactive model transformation. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, 2008.
26. Jean-Sébastien Sottet, Gaëlle Calvary, and Jean-Marie Favre. Mapping Model: A First Step to Ensure Usability for Sustaining User Interface Plasticity. In *Model Driven Development of Advanced User Interfaces (MDDAUI 2006)*, 2006.
27. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2. edition, December 2008.
28. E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson. User interface reverse engineering in support of interface migration to the web. *Automated Software Engg.*, 2003.
29. UsiXML V1.8 Reference Manual, February 2007.
30. Jean Vanderdonckt. A MDA-Compliant environment for developing user interfaces of information systems. In *Advanced Information Systems Engineering*. 2005.
31. Jean Vanderdonckt, Quentin Limbourg, Benjamin Michotte, Laurent Bouillon, Daniela Trevisan, and Murielle Florins. UsiXML: a user interface description language for specifying multimodal user interfaces. In *Proceedings of W3C Workshop on Multimodal Interaction WMI'2004*, 2004.
32. XML Linking Language (XLink) Version 1.1. http://www.w3.org/TR/xlink11/.