Flexibility in MDE for scaling up from simple applications to real case studies: illustration on a Nuclear Power Plant

Eric Céret, Gaëlle Calvary, Sophie Dupuy-Chessa

Flexibility in MDE for scaling up from simple applications to real case studies: illustration on a Nuclear Power Plant

Eric Céret[1] **Gaëlle Calvary**[1] **Sophie Dupuy-Chessa**[2]
Grenoble Institute of Technology[1], Pierre Mendès France University[2]
Grenoble Informatics Laboratory
41 rue des Mathématiques
38041 Grenoble Cedex 9, France
FirstName.Name@imag.fr

ABSTRACT
Model Driven Engineering provides powerful solutions for the development of User Interfaces. However, concepts and techniques are difficult to master and to apply: the threshold of use is said to be high, making designers and developers reluctant to use it. This paper investigates process model flexibility as a solution. We present three kinds of flexibility for improving design and development process models: (1) variability for equivalent choices, (2) granularability for several levels of details, (3) completeness for possibly optional and pre-defined reusable components. Flexibility decreases the threshold of use by reusability of knowledge, know-how and pieces of code. We illustrate these forms of flexibility on an industrial case study from the nuclear power plant domain. We explain how they are implemented in FlexiLab, a running prototype based on OSGi. The innovation is twofold: on one hand, the operationalization of flexibility; on the other hand, the jump from simple applications to real case studies thanks to flexibility.

Keywords
Flexibility, Model Driven Engineering, User Interface Development.

ACM Classification Keywords
D.2.10 Design : Methodologies, D.2.9 Management : Software process models, D.2.2 Design Tools and Techniques, H.5.2 User Interfaces

PROBLEM AND MOTIVATION

Many User Interfaces (UIs) design methods rely on models, like PUC [27], UsiXML [21], TERESA [26] or TADEUS [13]. They promote a Model Driven Engineering (MDE) approach, based on the generation of applications from models. High level models are successively reified into more concrete models until reaching the final code. This paradigm offers benefits like more evolving and reusable systems [18], greater quality, early detection of defects, incorporation of knowledge in executable models [25], and, carried to the extreme, dynamic adaptation of the UIs to the context of use [8]. However, this paradigm also suffers from several drawbacks. According to Lu and Wan [22] or to Traettenberg *et al.* [34], the adoption of MDE in the industry is not largely achieved and will only be possible if the development becomes more efficient than direct coding. Resnick *et al.* [30] state that MDE suffers from a high threshold of use (complexity of techniques and tools), a low ceiling (poor quality of the generated UIs) and tight walls (insufficient exploration of alternatives).

We are especially interested in decreasing the threshold of use. One approach is to support adaptable guidance (enactment of methods that suit well the knowledge, know-how and context of the project team) with appropriate tools. After presenting UI design and development methods in a first section, we survey how flexibility is defined in the literature. In the third section, we describe a case study from the nuclear power plant domain, whose results are detailed in the fourth section.

EXISTING DESIGN AND DEVELOPMENT METHODS

UI design methods based on MDE are numerous and have been studied by several authors, like Pinheiro da Silva [29], Barclay *et al.* [3], Mori et al. [26], and Lu and Wan [22]. In the following, we summarize their main drawbacks and analyze these limitations with regard to their threshold of use.

PUC [27] does not support tasks modeling and requires the direct design of the UI, annihilating some important benefits of MDE, such as the dynamic distribution of the UI over several devices. Aura [33] proposes to switch from one platform to another by running equivalent applications which cannot be considered as a dynamic adaptation of the UI. Teallach [3] and TADEUS [13] do not make a clear difference between the abstract and concrete UI levels, loosing flexibility in the gradual refinement of the presentation description.

In 2001, Calvary *et al.* [8] proposed an unifying framework combining the context of use (user, platform, environment), the domain model, the presentation model defined at several levels of abstraction (Tasks, Abstract UI (AUI), Concrete UI (CUI) and Final UI (FUI)) as well as the evolution model, giving rise to the generation of plastic UIs. Many projects implement this set of models and their successive transformations, like Cameleon-RT [2], TERESA [26] and UsiXML [21].

Figure 1 presents the transformations sequence proposed by UsiXML, which is similar to Cameleon but includes more (meta)models. This sequence supports two paths: one from the task model down to the code (i.e. FUI), and the other from the FUI up to the task model. Successive reifications transform the task model into an AUI, the AUI into a CUI and, finally, the CUI into a FUI. Several models are used during the transformations, see the box on the left of Figure 1). Two of these models are required in the UI generation process: the domain model, which represents the concepts that are manipulated in the UIs, and the context model, which represents the user, the platform(s) and the environment. Therefore, a team developing for instance an account management system would have to perform several stages. First, modeling the various concepts involved in the application (purchase demand, order, invoice, articles, provider, Value Added Taxes, ...) , and then defining the user model, platform models (smartphone, PC, Mac, tablet, ...) as well as the task model. The team would then have to create all the transformations for generating the AUIs, CUIs and FUIs. The development of such a simple application requires an important learning for an inexperienced team, even when not considering some other models such as the environment, transformation or quality models.

To face this complexity, UsiXML proposes some variants. For instance, it is possible to reverse-engineer an existing system, as, according to Limbourg *et al.* [21], the abstraction abilities of the approach make it possible to start from any model. However, even this reverse-engineering approach does not save much learning effort, as it proposes to abstract the entry model (the code, for instance) to obtain the task model that will be then step by step reified into the FUI. The flexibility brought by the abstraction sequence does neither avoid the reification flow nor the use of other tools than the UsiXML ones.

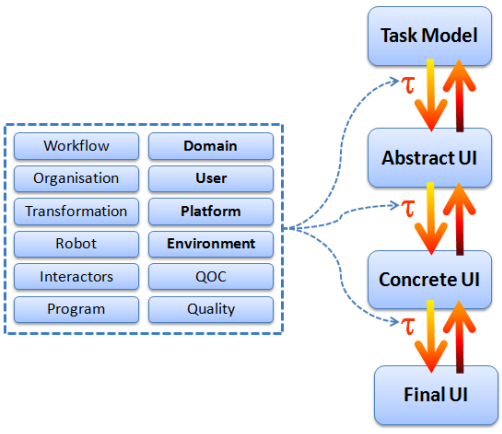

Figure 1. UsiXML development process

In all the existing works based on Cameleon, whatever the entry point is, designers and developers always have to learn the same competencies for manipulating models and transformations. Once trained, they have to produce lots of models and transformations, most often starting from scratch even when an application already exists. So flexibility remains limited.

Another approach for flexibility is services. For instance, ServFace [28] aims to automatically generate UIs by annotating existing web services. The composition of these annotated services is interpreted to generate the UI. Thereby, flexibility emerges from the choices made in the services and annotations. However, the designer is here limited by the availability (or the creation) of the needed services. If some original project requires a brand new functionality, it becomes difficult to find the corresponding service and to annotate it.

According to this state of the art, flexibility is not well integrated into UIs design and development processes. However, flexibility is needed for adapting processes to the designers' and developers' actual needs. The next section presents the works done in Information Systems and Software Engineering communities for embedding flexibility into design and development processes.

DESIGN METHODS FLEXIBILITY

For a long time, Information System and Software Engineering researchers [4, 14, 17] have noticed that design methods are often poorly or badly used because of their poor adaptability to the project specific needs, their conceptual and implementational complexity and rigidity, making flexibility a challenge. According to Agerfalk and Fitzgerald [1], the underlying hypothesis is that flexibility provides adaptability which in turn increases the ease of use, i.e. decreases the threshold of use. Different approaches offer various degrees of flexibility. Harmsen *et al.* [19] classify design methods in ascending flexibility: (1) "classical" methods considered as rigid, (2) choice between rigid methods, (3) selection of a path in a method outline, (4) selection and tuning of method outline, and (5) modular method construction. Design methods are said to be flexible at the third grade of this classification. For instance, JECKO [24] defines a set of fixed main stages refined into "fragments". Some fragments are systematically needed regardless of the context, while some others are only implemented when they fit the needs.

MAP [31] is a design method ranked at the fourth grade. This method proposes to elaborate a graph of successive goals and to identify the various strategies that can be used to achieve each goal. Searching the different possible paths de facto creates some flexibility.

However, as the authors themselves say [32], “it seems to be difficult to deal with the fuzzy concept of a goal”.

Situational methods [7, 12, 20] are graduated at the highest level. These methods are specifically built for a project. They are elaborated by composing methodological fragments. The project starts with a stage of method engineering, which aims to identify and then compose the interesting fragments. However they suffer from some limits: they require a deep knowledge of the methods from which the fragments are extracted and a specific expertise about the composition of these fragments which have not been conceived to fit together.

Moreover, these methods are rigid at enactment time. If, during the project lifecycle, some changes occur and imply a new methodological need, it becomes necessary to reconsider the method and eventually to reevaluate the usefulness of each element (fragment, strategy) and the need for other ones.

A NEW VISION OF FLEXIBILITY

In [9], we defined four kinds of flexibility: variability, granularability, completeness and distensibility. They are fully generic, coming from existing methods in Software Engineering and UI design.

Variability refers to the ability of a process model to offer designers equivalent variants. For instance, the goal “describe users' activities” can be achieved by the creation of an activity diagram, or by filming a user relating his daily activities, then transcribing the video and then creating a task model. Variability can also concern other elements of the process model, such as the choice between equivalent artifacts (e.g. documents with different structures but the same information).

Granularability is the ability of a process model to support elements with different granularities, e.g. with different languages and/or quantities of details. For instance, if the process model includes an activity for defining the structure of a database, it can suggest a goal "create the database". An expert database designer will not need more information whilst a novice designer will need a step-by-step process.

Completeness is the possibility of fulfilling or not the proposed process, some activities and/or artifacts are then optional or can be replaced by a predefined result or product. For instance, the activity "define the platforms model" can be avoided; in this case, the platform model can be replaced by “default” models that the designer picks up in a repository proposed by the process model.

Distensibility is the ability of a process model to be extended or reduced at enactment time, i.e. to accept that proposed elements (e.g., activities, roles, artifacts) are avoided or that unexpected elements are added.

Based on this taxonomy, we created M2Flex [10], a process metamodel supporting the four kinds of flexibility. M2Flex makes is possible to create process models that suit the needs of designers and developers at enactment-time, i.e. when they are selecting and realizing their design activities and producing the artifacts. In order to ensure the consistency of these process models all over their lifecycle, M2Flex is completed with a set of constraints, such as to verify that the process models contain an activity that produces the artifact needed by another activity. M2Flex is also empowered by D2Flex, an editor for creating flexible process models.

In this paper, we intend to show that flexibility can decrease the threshold of use of a rigid UI design and development process. To that end, we have studied several possibilities making the UsiXML method [35] more flexible and usable by companies. As the question of the threshold of use mainly concerns novice designers, which are not expected to modify the process model, distensibility is not addressed.

THE USIXML PROCESS MODEL

The UsiXML method [35] proposes an approach and a set of tools for the generation and execution of plastic UIs. It relies on the successive transformation of a task model into an Abstract UI, Concrete UI and then into a Final UI, while integrating, amongst others, the models representing the manipulated concepts and the context.

UsiXML is already supported by a wide set of various and efficient tools, but this set offers only a very partial flexibility. For instance, several tools can be used to create models and to prototype UIs, like SketchiXML [11], VisiXML or GraphiXML [23]. These tools have different usefulness, because they offer various degrees of reliability and make it possible to create more or less detailed and precise prototypes, according to the design or development stage. The prototype can be transformed into a Final UI for various runtime environments, either by transforming the underlying models directly in these tools or by using some plug-in or specific tools. This palette of tools offers a first level of flexibility. However, they all require that the designers create the UsiXML models and master the rational of these models.

Bouillon *et al.* [6] propose some flexibility with the ResersiXML tool. This tool makes it possible to generate an AUI and a CUI from an existing UI, saving part of the effort required to learn the models, and bringing ways to reuse existing systems. However this tool has a limited scope, being dedicated to Web only. Thus, the flexibility it supports is not generalized to all existing systems.

Despite the rich palette of tools sustaining UsiXML, the flexibility is partial, existing knowledge is poorly exploited and the reuse of existing elements is limited. We propose to significantly increase this flexibility by improving the process model. Reusing the designers and developers’ competencies and know-how becomes then possible, thereby decreasing the threshold of use of MDE in UIs development. Our contribution is illustrated on a

real industrial project, thereby showing how flexibility makes it possible to scale up from simple to real systems.

CONTEXT OF THE CASE STUDY

Industrial context

Atos develops software for control command in nuclear power plants. It is interested in extending its solutions with mobile possibilities. Indeed, if some of the actors in a nuclear power plant are staying in the control room (e.g. control room operators), the others are on the field (e.g. electricians, mechanics, safety engineers). As they are poorly equipped with electronic devices, they have to carry paper-based documentation and to synchronize with colleagues by phone. Part of this could become more efficient by using mobile devices. The nuclear industry is therefore seeking for solutions to provide complementary UIs in mobile situations. Of course, these UIs, named "Support UIs", are not intended to replace control room UIs, that have to be compliant with norms, classified and certified. Support UIs are limited to non-critical activities. For example, the system could provide field operators with technical documentation about the nuclear plants components they are working on, the global overview of the plant status and important information such as the current alarms.

Need for plastic UIs and model-driven solution

Mobility implies a highly dynamic context of use, which calls for plastic UIs:

(i) With regard to the user: different field operators do not need the same information. For instance, an electrician needs the electrical schemas of an equipment whilst a mechanics would rather use a vapor or an air tubes schema. The existing UIs present all the various schemas together.

(ii) With regard to the platform: a field operator is intended to use PCs, information terminals, tablets and possibly smartphones.

(iii) With regard to the environment: needed information depends on the plant status and on the situation of the equipment the operator is working on, because he needs to evaluate the compatibility of his action with the rest of the installation and the ongoing actions. For instance, the operator has to know the current state of the installation (e.g. pressures, temperatures, water levels) and to synchronize with other operators.

Another requirement is in favor of a model driven approach: the duration of the systems. Nuclear power plants are designed to last several decades. The knowledge and representations of the installation (maps, schemas,...) have to be updated during all this time. For instance, when a valve has to be changed, if this kind of valve is no more on the market, the documentation has to be updated to track the versioning of the component. In current practices, paper documents are updated. To increase their performance, users need to be provided with pertinent information only. Computerized models could provide an adapted view of all these documents: current situation, relevant part of the situation (e.g. electrical schema without mechanical equipment), situation at a given time, modifications and so on.

Need for flexibility

The Advanced Data Acquisition And Control System (Adacs) developed by Atos provides the nuclear industry with proven supervision and command/control systems, full scope engineering and training simulators and intelligent predictive maintenance systems. Adacs is designed for nuclear industry requirements, and for addressing new build and refurbishment projects. It relies on over 30 years experience in instrumentation and control for the nuclear power industry, with 300 systems installed in over 70 nuclear plants. Adacs offers several hundreds of UIs, especially for control room operators.

Obviously, converting one by one all these UIs into models, creating by hand all needed transformations (as it would be done in the classical Cameleon or UsiXML approaches) would be out of proportion with the

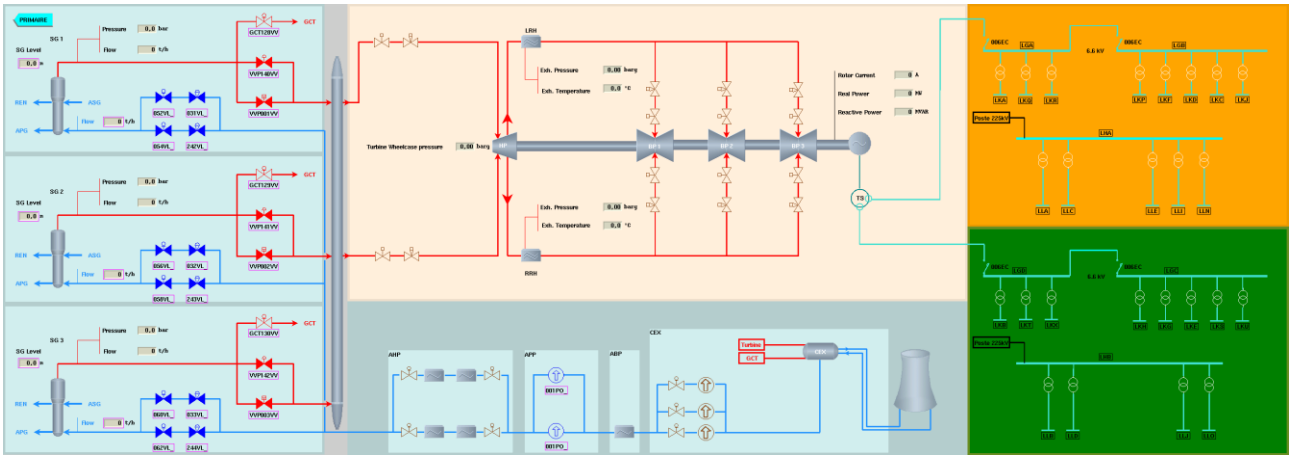

Figure 2. An existing UI from the MIMIC module

benefits, requiring huge budget and period of work. Moreover, Atos is not used to apply such approaches. They fear investing lot of time and energy for a result that would finally not suit their needs. They also have a strong expectation of reusing the existing system.

As a consequence we studied how plastic UIs could be implemented at the lowest cost and difficulty, making it possible to learn the design process step by step.

Goals and methodology of the case study

The study has been divided into three stages:

(1) First, we have presented the UI plasticity and the underlying concepts (remolding, redistribution, models, transformations,...) and the UsiXML approach to the team in charge of the UIs in Adacs.

(2) Then, together with the Atos team, we have elicited one Adacs module, which we wanted to be representative of the existing code.

(3) Finally, we have studied how to ease the creation of models and transformations for "plastifying" the UIs of the selected module. This work was conducted in close collaboration with Atos.

In the second step, we have chosen the MIMIC module, which provides a control room operator with a graphical representation of the process that takes place in the power plant. It relies on a model of (a) all components (electrical, mechanical, thermal-hydraulic and neutronic materials), and (b) all circuits, like water, air or vapor circuits. The MIMIC module contains 1,000 to 1,500 UIs. An example is given in figure 2, which represents (a) a vapor-producing circuit on the left, with three vapor generators and several valves, (b) the turbines and the cooling circuit in the center, and (c) the electrical circuits on the right. The menus to access the MIMIC UIs are structured according to the hierarchy of components (e.g. the vapor generators cooling systems, like the Regulated Water Supply and the Emergency Water Supply) and circuits (e.g. primary, secondary water circuits).

The existing UIs are developed with Rogue Waves Views. This design studio offers functionalities to define layers, elementary and structured interactors and the mapping between graphical objects and business objects. Adacs specific interactors are defined in graphical objects libraries. Each UI is recorded in a text file, with the ilv extension (named “ilv file” in the following). At runtime, the program that has to display a UI calls the Rogue Wave API and asks to read an ilv file. The API produces C++ objects corresponding to all elements included in the UI (structured interactors, elementary interactors - independent or embedded in other interactors - layouts,...). Each C++ object has up to 100 attributes, representing all needed information to display it (layout number, position in the layer, business object identifier it is linked to, default values, family, minimum and maximum value and so on).

CASE STUDY

We studied how process model flexibility can alleviate the design and development of the UIs. In the next sections, we present how three kinds of flexibility can be used in this industrial case study. We illustrate four examples of variability, one example of completeness and one example of granularability in the specific MIMIC case. Distensibility, the ability of extending or reducing the process model at enactement time, is out of scope of this study which focuses on design time.

Variability

The first form of flexibility is variability i.e. the possibility of choosing one path in a set of equivalent variants. In the following examples, we illustrate the variability by proposing variants to the classical MDE approach: domain model generation, CUI generation from existing UIs, task model generation from application parameters and existing UIs and creation of ATL transformations by composing existing elementary rules.

CUIs generation from existing UIs

In the classical UsiXML approach, designers could start from any level of abstraction, for instance tasks, AUI or CUI. The other levels are then reified or abstracted from the starting level. We propose a first example of variability: CUIs are generated from existing UIs, saving considerable part of the effort to be made for learning the CUI model and (re)modeling the UIs.

The goal of our case study is to ease redesigning the MIMIC UIs for mobile devices. To achieve that, we explored the possibility of converting these UIs into plastic UIs, in other words transforming the existing UIs into a CUI model. If needed, the other models (AUI, tasks) could then be classically generated from these CUIs.

The ilv files are at the same level of abstraction than the CUI. Our first idea was to write a metamodel of these files and an ATL transformation that accepts one ilv file as entry and produces a CUI model. However, Atos experts estimated that the syntax of the ilv files is too complex and too poorly known to easily write such a transformation. By contrast, they are used to manipulate the C++ objects generated from the ilv files. They estimated that it would be much easier to write a C++ program that parses all the UI objects and their attributes. Layouts are then transformed into containers whilst structured and elementary objects are respectively transformed into structured and elementary interactors. The C++ parser can then produce an XML file representing the complete structure of a UI. This XML file can then be in turn transformed into a CUI.

The use of the parser generating the CUIs from existing UIs saves the hand-work for creating the CUI models. However it does not save all the hand-work: the C++ parser and the XML to CUI transformation have to be

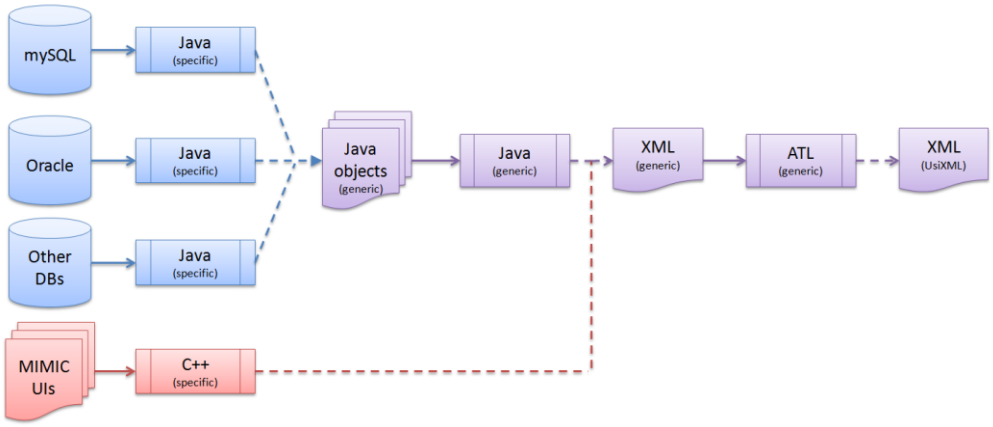

Figure 3. Database to domain model transformation in UsiComp

created, as well as the graphical objects libraries for the targeted languages. Indeed, existing UIs rely on libraries built to be used by the Rogue Waves API. These libraries are not usable by other languages.

Moreover, the generated CUIs will contain the same defects than the source C++ UIs. Some of these defects will be fixed by the application of ergonomic rules during the adaptation process, e.g. generating radio buttons instead of a drop-down list when there are three options. The other defects will have to be corrected by hand in a second time. This process matches our goal: making it possible to convert the existing UIs into the models paradigm without having to learn a lot about models and transformations. The first version of the generated UIs will obviously be imperfect. However, we estimate that improving it is easier for novice designers than to develop it from scratch.

Creation of the concept model

Another example of variability is the possibility of generating the concept model from existing applications. One difficulty in creating a concept model is that very few industrial teams know the meta-model of concepts – named the domain meta-model in UsiXML. By contrast, many of them master the creation of a database and some of them already have an existing database in the system that has to evolve. This is why we created DB2Domain, a generic tool for generating the concept model from a database. The underlying principle is to extract from the database the metadata that represent the structure of the information and to create a standardized XML file. This file is then converted, using ATL transformations, into a concept model that conforms to the UsiXML meta-model. DB2Domain has been implemented in Java in 2011 and used in our academic demonstrators.

This process can easily be extended for the MIMIC module. As presented before, the C++ graphical objects and layouts can be parsed to generate the CUI. Each object refers to a business object. Some attributes of a graphical object represent the attributes of the referred business objects. For instance, the interactive object A refers to the business object ASG120987VAN, which is a valve. The attribute named 'value' in A represents the openness percentage in ASG120987VAN.

In a graphical object, the number and the names of the attributes that represent a business attribute depend on the family of the graphical object. An electrical valve has an openness rate, a pomp has a pressure at the inlet and at the outlet, and so on. Business objects may have attributes that have no correspondence with graphical objects, but we are not interested in these attributes because they are not presented to the user. Conversely, graphical objects have attributes that do not refer to business attributes, such a color, size or position.

It is then possible to extract the structure of the manipulated concepts from the graphical objects. In other words, it is possible to generate the domain model by parsing the UIs objects.

The general process for transforming databases in a UsiXML domain model is shown on figure 3, with the extension for the MIMIC module (in red on the figure). For databases, the first step consists in executing a specific Java code that instantiates the objects representing the structure of the database. This code is dedicated to one DBMS (e.g. Oracle), but works for all databases managed by the same system: every team using an Oracle database can use the same Oracle specific code. The Java objects are then used for generating an XML file that has a generic structure. For the MIMICS module, the transformation is realized by the C++ parser, which generates the same generic XML file. In both the DBMS and the MIMICS cases, the XML file is finally transformed into a domain model, thanks to an ATL transformation that already exists. In order to align the procedures, the parser will interpret a class of graphical objects (e.g. valves) as a table and the attributes as the fields of this table.

The creation of the domain model is thereby made much easier: the team developing MIMIC only has to write a C++ routine for parsing objects that they are used to manipulate. They also have to learn the structure of the

XML file they have to produce. This is an easy work which most developers are familiar with.

However, the potential defects in the source C++ objects will be transferred into the generated domain model. Usually we recommend to improve this model in a second step. However, as the goal in this case study is to connect the UIs with an existing data structure, the possibilities of modifying the domain model - which has to match this data structure - are limited. In such a case, we recommend to create if needed a functional core adapter, as promoted by the Arch architecture [5]. Another limit is that the mapping with the task model is missing. We will see in the next section that some part of it can also be generated in the specific case of this study.

Task model generation

Part of the task model can be abstracted from the CUIs generated from existing files. However, the resulting task model would be incomplete: the navigation between the UIs is for instance missing. We present hereafter a solution for completing tasks models.

In Adacs, the available UIs are described in files. These files are used to instantiate objects in the functional core. As already mentioned, the UIs are grouped into elementary systems and circuits. This organization represents all the possible actions for an operator. In other words, it is somehow the image of a task tree, where leaves (the UIs) are concrete tasks and nodes (circuits and elementary system) are abstract tasks. It is then possible to offer a variant to the classical approach: the menus for generating the task model nodes, the leaves being obtained from the CUIs already generated from UI objects.

However, this task model would not be adapted to the end-user's role. As already mentioned, the UIs presented to the control room operators show all kinds of information, whilst an electrician would certainly not be interested in all the UIs presenting the mechanical circuits. Thus, parts of the UIs would have to be manually deleted from the task model of his profile. Moreover, the generated task tree would reflect the existing organization of the MIMIC menus. There is no evidence that this organization is the good one.

To provide a complete model driven solution, we also need to generate the mapping between tasks and the elements of the domain model. This can be done simultaneously with the CUI generation from the C++ objects: as mentioned before, some objects contain information for identifying the domain element they are representing. This makes it possible to infer the link between these interactors and the domain objects. The other objects refer to a named business object but do not include information about the data themselves.

Transformation rules creation

Usually, the developer has to create a transformation containing all the elementary rules driving the conversion of one (or several) model(s) into a resulting model (or several resulting models). He has to elaborate a set of rules specifically dedicated to each model(s)-to-model(s) transformation. Reusability is limited, and it is necessary to master the transformation language.

To make this more flexible, we propose to create transformations by composing elementary rules that the designer can pick up in a repository, as it has been done in UsiComp [15]. On the graphical UI of this tool, the designer sees the source model and builds graphically the target model component by component, while indicating which elementary rule manages this transformation. When he is satisfied, he validates the transformation and a tool generates automatically a file containing and composing all the chosen elementary rules. This tool makes it possible for designers to easily reuse elementary rules even if they do not know the transformation language. This solution has been used in our demonstrator [15]. According to our experience, it facilitates the creation and the reuse of the transformation rules and thereby the generation of UIs. This is especially true for generic rules that deal with classical interactors, such as the transformation of a task into an AUI unit, or the generation of a text field. It appears to be widely usable, because ATL rules are by construction an aggregation of elementary rules. However, we noticed one major limit: as the selected rules are included in the resulting transformation, it is no more possible to identify them individually and to change one of them while calculating the UI adaptation.

As illustrated by the four previous examples, adding variability in the approach can save designers efforts. Transforming existing UIs and parameters into models saves the learning of the task, CUI and domain models. Transformations are made easier to create.

Even if some parts of the solutions presented above are application-specific, they are easy to adapt to other applications. For instance, another project could require other parsers to produce the various XML files, but designers could reuse the transformations for converting these XML files into domain, task or CUI models. They can be integrated into the design and development

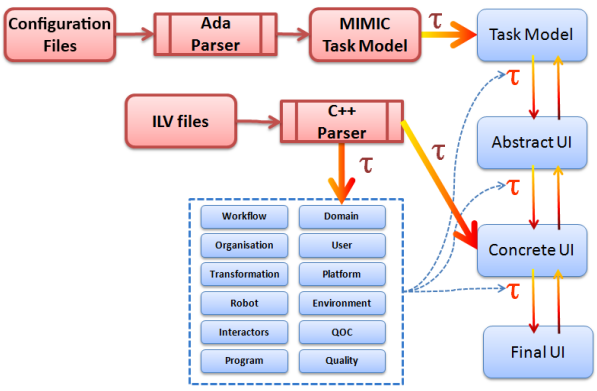

Figure 4. Three possible degrees of variability

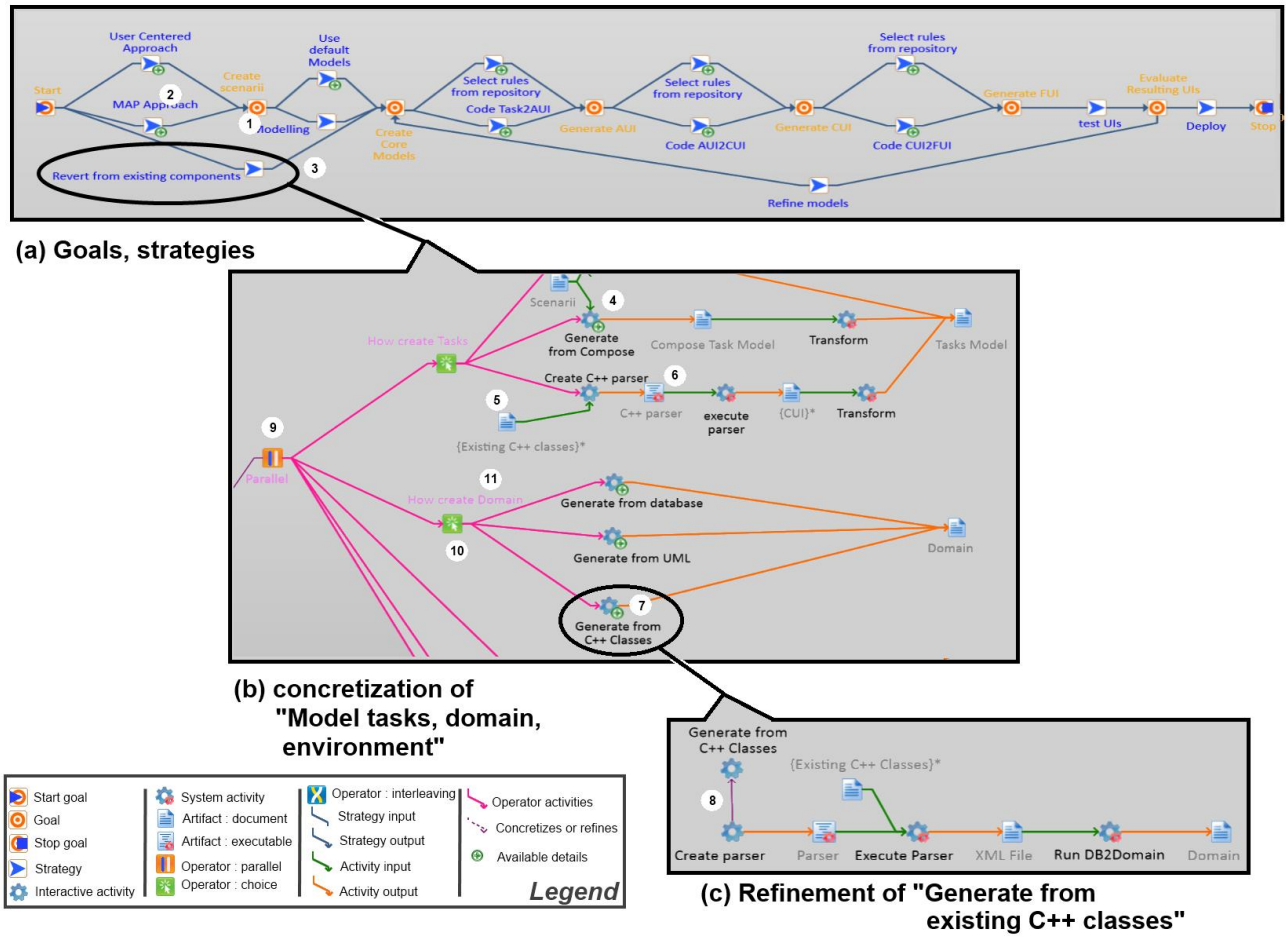

Figure 5. UsiXML flexible process model

process and thus open new generic entry.

The four forms of flexibility detailed above illustrate the variability dimension of flexibility. Figure 4 indicates how three[1] of the alternative processes (written in bold text) are integrated into the UsiXML design and development process: existing objects are transformed into UsiXML models. However, variability is not the only form of flexibility that can be included into the UsiXML classical process. It is also possible to save efforts in the modeling of the context of use, as illustrated in the next section.

Completeness

Completeness questions if the creation process requires that all stages have to be fully completed. For instance, do we have to produce all ad-hoc models? In our case, designers do not need to implement their early prototypes with all the characteristics of the context models. In our case study, the number of possible platforms is low, the operating systems are known, and UI modeling will take into account the size of the display surface. The platform characteristics to be managed are very common: there is no need for a specific platform model. In such a case, our approach proposes to not fulfill the complete design and development process and to use "default" platform models. These simplified models are pre-defined (and thereby generic) and include only the essential characteristics needed for the generation of the FUI: the operating system and the screen size.

Completeness of the process could be studied for others models, like the user model (the user being then simplified as monolingual, able-bodied, and competent) or the environment model (the environment being seen as "average", i.e. neither too much or too less dark or luminous or noisy). In our case study, it not possible to use default user or environment models, because we need to manage characteristics that cannot be defaulted: as mentioned in the presentation of the case study, the UIs have to adapt dynamically to the user's profession and therefore the user model has to include information about the user's profile.

Granularability

UsiXML does not support various levels of details nor the languages addressing various designers' expertise. We have added several possibilities. A simple example is

[1] The creation of transformation rules is not represented because it would overburden the figure.

the various levels of detail that we propose when configuring and executing our tool for generating the domain model from a database. Expert designers just execute the tool whilst step by step explanations are provided for novice developers. This corresponds to the requirement of granularability: supporting various levels of complexity for covering various levels of expertise.

Generalization

In the previous sections, we have shown how the UsiXML process model could be completed with alternative activities that would make it easier for Atos engineers to create the models needed for "plastifying" the UIs. In the one hand, some of these new activities are very specific, dedicated to the existing elements, like the generation of the CUIs from the C++ classes. In the other hand, some are fully generic and reusable in other projects. This is true for the selection of elementary rules to be aggregated into transformations, the default models, and the various amounts of details. Thanks to D2Flex, our tool for defining process models, all these possibilities have been integrated into the UsiXML classical process model and can now be used for guiding developers.

Figure 5 presents screenshots of the flexibilized UsiXML process model in D2Flex. Figure 5a presents the highest level of the process model, with goals (number 1 on the figure) and various strategies (2) for reaching the goals. (3) represents the newly added strategy for generating the model from the existing components. On figure 5b, activities (4) concretize strategies and input (5) or output (6) artifacts, that can be documents (5) or executables (6). Refinement is also implemented: refined activities are represented with a "+" sign drawn of them (7). When displaying a refined activity, a new window is displayed, e.g. parts (b) and (c) of figure 5. It is then possible to design how the activity is detailed. On part (c) of figure 5, the activity "Generate [Domain model] from existing C++ classes" is detailed (8) by a sequence of (sub-)activities. This figure also shows that activities can be realized in parallel (9) and that the designer is can make choices between various possibilities (10).

CONCLUSION AND PERSPECTIVES

This paper promotes flexibility for the development of plastic UIs while decreasing the threshold of use of MDE. We presented three forms of flexibility, variability, granularability and completeness. We shown how they are enacted on an industrial case study. Obviously, the UIs produced by such a flexible development process cannot be "perfect". However, thanks to the process flexibility, designers and developers can reuse parts of their know-how and competencies, and are able to transfer some existing components into the paradigm of models: it makes it possible for them to create a first, albeit imperfect, version of their UIs, that they can iteratively improve, acquiring step by step the needed competencies. With our industrial partner, we plan to conduct evaluations, in order to more comprehensively estimate the reaction of designers when facing flexibility. We also intend to extend and complete the flexible design and development process, in order to integrate, in the one hand, activities for improving UIs while acquiring competencies and, in the other hand, activities for designers and developers who already have skills in MDE.

We also intend to improve the aggregation of elementary ATL rules into transformations. Instead of directly generating the transformation, we will generate the list of rules to be aggregated at runtime. Thus, the adaptation module will be able to replace one of the rules by a more relevant one and finally to generate the transformation and to execute it.

We also intend to increase the ceiling of the MDE approach, for instance by integrating systems sustaining designers' creativity, like Magellan [20], which is based on genetic algorithms allowing to generate mutations into the UIs and thus to propose variations in the process.

Lastly, we aim to provide designers and developers with flexibility at enactment-time, by completing our software suite. A new module, R2Flex, will make it possible to modify the design and development process while executing it. Distensibility will then also be supported.

ACKNOWLEDGMENT

The authors warmly thank the Connexion project (2012-2016) for the real-size and challenging case study.

REFERENCES

1. Agerfalk, P.J. and Fitzgerald, B. Flexible And Distributed Software Processes: Old Petunias In New Bowls? *Commun. ACM 49*, 10 (2006), 26–34.
2. Balme, L., Demeure, A., Barralon, N., Coutaz, J., and Calvary, G. CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. *EUSAI*, (2004), 291–302.
3. Barclay, P., Griffiths, T., McKirdy, J., et al. Teallach - a flexible user-interface development environment for object database applications. *The Journal of Visual Languages and Computing*, 2003, pp. 47–77.
4. Barry, C. and Lang, M. A Survey of Multimedia and Web Development Techniques and Methodology Usage. *IEEE MultiMedia 8*, 2 (2001), 52–60.
5. Bass, L., Faneuf, R., Little, R., et al. A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop. *SIGCHI Bull. 24*, 1 (1992), 32–37.
6. Bouillon, L., Limbourg, Q., Vanderdonckt, J., and Michotte, B. Reverse Engineering of Web Pages based on Derivations and Transformations. *Proc. of 3 rd Latin American Web Congress LA-Web'2005 (Buenos Aires, October 31-November 2, 2005), IEEE*

Computer Society Press, Los Alamitos, 2005, (2005), 3–13.
7. Brinkkemper, S., Saeki, M., and Harmsen, F. Meta-modelling based assembly techniques for situational method engineering. *Information Systems 24*, 3 (1999), 209 – 228.
8. Calvary, G., Coutaz, J., and Thevenin, D. A Unifying Reference Framework for the Development of Plastic User Interfaces. In M. Little and L. Nigay, eds., *Engineering for Human-Computer Interaction*. Springer Berlin / Heidelberg, 2001, 173–192.
9. Céret, E., Dupuy-Chessa, S., Calvary, G., Front, A., and Rieu, D. A taxonomy of design methods process models. *Information and Software Technology, Elsevier 55*, 5 (2013), 795–821.
10. Céret, E., Dupuy-Chessa, S., and Calvary, G. M2Flex: a process metamodel for flexibility at runtime. (2013), 117–128.
11. Coyette, A. and Vanderdonckt, J. A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces. In M. Costabile and F. Paternò, eds., *Human-Computer Interaction - INTERACT 2005*. Springer Berlin / Heidelberg, 2005, 550–564.
12. Cross, N. *Engineering Design Methods*. John Wiley and Sons Ltd., Chichester, West Sussex, England, 1989.
13. Elwert, T. and Schlungbaum, E. Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. *DSV-IS*, (1995), 193–208.
14. Fitzgerald, B. An empirical investigation into the adoption of systems development methodologies. *Information & Management 34*, 6 (1998), 317 – 328.
15. Frey, A.G., Ceret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. UsiComp: an extensible model-driven composer. *EICS*, (2012), 263–268.
16. Gabillon, Y., Petit, M., Calvary, G., and Fiorino, H. Automated planning for user interface composition. *Proceedings of the 2nd International Workshop on Semantic Models for Adaptive Interactive Systems: SEMAIS'11 at IUI 2011 conference*, Springer HCI (2011).
17. Garzotto, F. and Perrone, V. Industrial Acceptability of Web Design Methods: an Empirical Study. *Journal of Web Engineering 6*, 1 (2007), 73–96.
18. Hamid, B., Radermacher, A., Lanusse, A., Jouvray, C., Gérard, S., and Terrier, F. Designing Fault-Tolerant Component Based Applications with a Model Driven Approach. *SEUS*, (2008), 9–20.
19. Harmsen, F., Brinkkemper, S., and Oei, J.L.H. Situational method engineering for informational system project approaches. *Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle*, Elsevier Science Inc. (1994), 169–194.
20. Kornyshova, E., Deneckère, R., and Salinesi, C. Using Multicriteria Decision-Making to Take into Account the Situation in System Engineering. *CAiSE Forum*, (2008), 17–20.
21. Limbourg, Q. and Vanderdonckt, J. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence. *ICWE Workshops*, (2004), 325–338.
22. Lu, X. and Wan, J. Model Driven Development of Complex User Interface. *MDDAUI*, (2007).
23. Michotte, B. and Vanderdonckt, J. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. *ICAS*, (2008), 15–22.
24. Mirbel, I. and De Rivière, V. Introducing flexibility in the heart of analysis and design. (2002).
25. Mohagheghi, P., Fernandez, M., Martell, J., Fritzsche, M., and Gilani, W. MDE Adoption in Industry: Challenges and Success Criteria. In M.V. Chaudron, ed., *Models in Software Engineering*. Springer Berlin Heidelberg, 2009, 54–59.
26. Mori, G., Paterno, F., and Santoro, C. Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. *IEEE Trans. Softw. Eng. 30*, 8 (2004), 507–520.
27. Nichols, J., Myers, B.A., Higgins, M., et al. Generating remote control interfaces for complex appliances. *UIST*, (2002), 161–170.
28. Paternò, F., Santoro, C., and Spano, L.D. Designing Usable Applications based on Web Services. *Proceedings of the International Workshop on Interplay between Usability Evaluation and Software Development (I-USED)*, (2008).
29. Pinheiro da Silva, P. User Interface Declarative Models and Development Environments: A Survey. In P. Palanque and F. Paternò, eds., *Interactive Systems Design, Specification, and Verification*. Springer Berlin / Heidelberg, 2001, 207–226.
30. Resnick, M., Myers, B., Nakakoji, K., et al. Design Principles for Tools to Support Creative Thinking. *A Workshop Sponsored by the National Science Foundation*, (2005), 25–35.
31. Rolland, C., Prakash, N., and Benjamen, A. A Multi-Model View of Process Modelling. *Requirements Engineering 4*, 4 (1999), 169–187.
32. Rolland, C. From Conceptual Modeling to Requirements Engineering. In D. Embley, A. Olivé and S. Ram, eds., *Conceptual Modeling - ER 2006*. Springer Berlin / Heidelberg, 2006, 5–11.
33. Sousa, J.P. and Garlan, D. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *WICSA*, (2002), 29–43.
34. Trætteberg, H., Molina, P.J., and Nunes, N.J. Making model-based UI design practical: usable and open methods and tools. *Proceedings of the 9th international conference on Intelligent user interfaces*, ACM (2004), 376–377.
35. Vanderdonckt, J. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. *Proc. of 17th Conf. on Advanced Information Systems Engineering CAiSE'05*, Springer-Verlag (2005), 13–17.