Model-Driven Engineering of Dialogues for Multi-platform Graphical User Interfaces

Efrem Mbaki[1], Jean Vanderdonckt[1], Marco Winckler[2]
[1]Louvain Interaction Lab., Louvain School of Management (LSM), Univerisité catholique de Louvain, Place des Doyens, 1
B-1348 Louvain-la-Neuve (Belgium) –{efrem.mbaki@student, jean.vanderdonckt@}uclouvain.be
[2]IRIT Toulouse (France) – winckler@irit.fr

ABSTRACT
This paper describes a model-driven engineering of interactive dialogues in graphical user interfaces that is structured according to the three lowest levels of abstraction of the Cameleon Reference Framework: abstract, concrete, and final user interface. A dialogue model captures an abstraction of the dialogue as opposed to a traditional presentation model that captures the abstraction of the visual components of a user interface. The dialogue modeled at the abstract user interface level can be reified to the concrete user interface level by model-to-model transformation, which in turn leads to code by model-to-code generation. Five target markup and programming languages are supported: HTML V4.0, HTML for Applications (HTA), Microsoft Visual Basic for Applications V6.0 (VBA), and DotNet V3.5 framework. Two computing platforms support these languages: Microsoft Windows and Mac OS X. Five levels of dialogue granularity are considered: object-level (dialogue of a particular widget), low-level container (dialogue of any group box), intermediary-level container (dialogue at any non-terminal level of decomposition such as a dialog box or a web page), intra-application level (application-level dialogue), and inter-application level (dialogue across different interactive applications). A Dialog Editor has been implemented that holds abstractions pertaining for expressing the dialogue at an abstract level and then producing a final user interface for any of these five languages for both platforms.

ACM Classification: C.2.4 [**Computer-Communication Networks**]: Distributed systems – *Distributed applications*. D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – Rapid Prototyping; reusable software. H5.2 [**Information interfaces and presentation**]: User Interfaces – *graphical user interfaces, user interface management system (UIMS)*.

General terms: Algorithms, Design, Languages, Theory.

Keywords: Dialogue model, event-condition-action rule, model-driven engineering, model-driven user interface development, model-to-model transformation, model-to-code generation, user interface description language.

INTRODUCTION
We hereby refer to *dialogue* as being the dynamic part of a Graphical User Interface (GUI) such as the physical and temporal arrangement of widgets in their respective containers and their evolution over time depending on the user's task. The dialogue regulates the ordering of these widgets so as to reflect the constraints imposed by the user's task. The dialogue has been also referred to as *behavior*, *navigation*, or *feels* (as opposed to *look* for presentation) [1,2,12]. Here are some typical examples of dialogues: when the end user selected her native language in a list box, a dialog box is translated accordingly; when a particular value has been entered in an edit field, other edit fields are deactivated because they are no longer needed; when a validation button is pressed, the currently opened window is closed and another one is opened for pursuing the dialog. Conceptual modeling [1], model-based design [4] or model-driven engineering [20] of the dialog has already been introduced since years [12] in order be derived from a task model [11,17,25,31,33], perhaps combined with a domain model [30] or a service model [4], to derive its software architecture from its model [23], to analyze its properties [5,32], to foster component reuse [10], to check some dialogue or usability properties [32], to support adaptation [19], to automatically keep trace of interaction and analyze them afterwards [25]. Dialog models have been used in several domains of applications, such as web engineering [3,5], information systems [18], multi-device environments [27], multimedia applications [23,24], multimodal applications [28], and workflow systems [29,30].

Dialogue modeling has however often been considered harmful for several reasons which may impediment further research and development in this area:

1. *Choosing the modeling language paradigm is a dilemma*: an imperative or procedural language is often more suitable and convenient to represent a GUI dialogue than a declarative language. The last could introduce a verbose representation of something that could be expressed in a straightforward way in the latter. The current trend goes in favor of scripting languages.
2. *Abstracting the right concepts is complex*: finding the aspects of a dialog that should lead to abstraction is not straightforward and turning them into an abstraction that is expressive enough without being verbose is hard. A dialogue model may benefit from a reasonable level of expressiveness, but will prevent the designer from specifying complex dialogues while another dialogue model may exhibit more expressiveness, but is consid-

ered complex to use. Which modeling approach is also an open question: taking the greatest common denominator across languages (with the risk of limited expressiveness) or more (with the risk of non-support).

3. *Heterogeneity of computing platforms is difficult to handle*: Integrated Development Environments (IDEs) are often targeted to a particular programming language or markup language that is dedicated to a particular operating system or platform. Some IDEs exist (e.g., Nokia QT (http://qt.nokia.com/products, QtK) that address multi-platform GUIs, but they remain at the code level or their usage is still complex.
4. *Model-driven engineering of dialogue is more challenging than model-based design*. Model-based GUI design only assumes that one or many models are used to design parts or whole of a GUI, while Model-Driven Engineering (MDE) [21] imposes at least one User Interface Description Language (UIDL) [7] that should be rigorously defined by a meta-model (preferably expressed in terms of MOF language, but not necessarily). Model-based GUI design may invoke virtually any technique, while model-driven engineering imposes that everything is rigorously defined in terms of model transformations, which are in turn based on a meta-model.

This paper is aimed at addressing the aforementioned challenges by applying MDE principles to designing a dialog for GUIs belonging to different computing platforms. The remainder of this paper is structured as follows: Section 2 reports on the main trends so far in dialogue modeling, Section 3 defines the conceptual model of dialogue used in this paper, Section 4 presents an overview of the methodological approach with three views: model & language, stepwise approach, and software support. A running example is given to exemplify how this approach is executed. Section 5 motivates our software implementation with multi-level dialog model editing, model-to-model transformation, and model-to-code generation. Section 5 concludes the paper and addresses some avenues.

STATE OF THE ART

Overview of dialogue modeling techniques

A very wide spectrum of conceptual modeling and computer science techniques has been used over years to model a dialogue [1-5, 8-14,16-35], some of them with some persistence over time, such as, but not limited to: Backus-Naur Form (BNF) grammars [12,16], state-transition diagrams in very different forms (e.g., dialog charts [1], dialog flows [3], abstract data views [10], dialog nets [9], windows transitions [33]), state charts [14] and its refinement for web applications [5], and-or graphs coming from Artificial Intelligence (e.g., function chaining graphs [18]), event-response languages, and Petri nets [2]. Some algorithms [17] have been also dedicated to support the dialog design through models, such as the Enabled Task Set [22]. Rigorously comparing these models represents a contribution that is yet to appear. Green [9] compared three dialogue models to conclude that some models share the same expressivity, but not the same complexity. Cachero *et al.* examine how to model the navigation of a web application [5]. In [9], the context model drives a dialogue model at different steps of the UI development life cycle. So far, few attempts have been made to structure the conceptual modeling of dialogues in the same way as it has been done for presentation, the notable exception being applying StateWebCharts [34] with Cascading style sheets [35] in order to factor out common parts of dialogues and to keep specific parts locally.

Some recent dialogue modeling techniques

The DIAMODL runtime [30] models the dataflow dialog as JFace Data Binding and includes extensions for binding EMF data to SWT widgets in order to link domain and dialogue models. Statechart logic is implemented by means of the Apache SCXML engine [36], while GUI execution utilizes an XML format and renderer for SWT.

The Multimodal Interface Presentation and Interaction Model (MIPIM) [28] could even model complex dialogues of a multimodal user interface together with an advanced control model, which can either be used for direct modeling by an interface designer or in conjunction with higher level models. Van den Bergh & Coninx [31] established a semantic mapping between a task model with temporal relationships expressed according to ConcurTaskTrees notation and UML state machines as a compact way to model the dialog, resulting into a UML profile.

Figure 1 graphically depicts some dialogue models in families of models. Each family exhibits a certain degree of model expressiveness (i.e., the capability of the model to express advanced enough dialogues), but at the price of a certain model complexity (i.e., the easiness with which the dialogue could be modeled in terms specified by the meta-model). At the leftmost part of Figure 1 are located (E)BNF grammars since they are probably the least expressive dialogue models ever. Then we can find respectively State Transitions Networks and their derivatives, then Event-Response Systems. Petri nets [2] are probably the most expressive models that can be used to model dialogues, but they are also the most complex to manipulate.

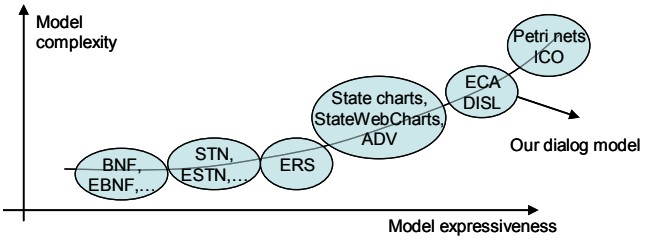

Figure 1 Expressiveness of Model Complexity.

Dialogue modeling techniques in UIDLs

Less expressive and less complex are Event-Condition-Action (ECA) systems that are considered in several UIDLs such as DISL [27,28], UIML [15], MariaXML [22] and UsiXML [35], probably because they are convenient to describe according to a declarative paradigm, that is often predominant in defining models in UIDLs. But their expressivity is limited by the model concepts coverage.

In *UIML* [15], a dialogue is defined as a set of condition-action rules that define what happens when a user interacts with any GUI element, such as a button, a group box, a window. In *MariaXML* [22], a dialog model describes parallel interaction between a GUI and its end user through connections. A *connection* indicates what the next active presentation will be when a given interaction takes place: elementary connection, complex connection (in which a logical formula composes elementary conditions), or a conditional connection (when specific conditions are associated with it). In *UsiXML* [35], a behavior is defined as a set of ECA rules, where: an *event* can be any UI event that is relevant to the level of abstraction (abstract or concrete), a *condition* can state any logical condition on a model, a model element, or a mapping between models, an *action* can be any operation on widgets (abstract or concrete).

The Cameleon Reference Framework

Several UIDLs [7] are structured according to the four steps of the Cameleon Reference Framework (CRF) [6], that are now recommended to consider by W3C [7]:

1) *Task & Concepts* (T&C): describe the various user's tasks to be carried out and the domain-oriented concepts required by these tasks to be performed.

2) *Abstract UI* (AUI): defines abstract containers (AC) and individual components (AIC), two forms of Abstract Interaction Objects (AIO) by grouping subtasks according to various criteria (e.g., task model structural patterns, cognitive load analysis, semantic relationships identification). As in Guilet Dialog Model [26] a navigation scheme between the container and selects abstract individual component for each concept so that they are independent of any interaction modality. The AUI is said to be *independent of any interaction modality*.

3) *Concrete UI* (CUI): concretizes an abstract UI for a given context of use into Concrete Interaction Objects (CIOs) so as to define widgets layout and interface navigation. It abstracts a final UI into a UI definition that is independent of any computing platform. A CUI assumes that a chosen interaction modality, but the CUI remains *independent of any platform*.

4) *Final UI* (FUI): is the operational UI i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after compilation of code in an IDE).

CONCEPTUAL MODELING OF DIALOGUE

In order to apply MDE techniques, we need to define a dialog model that is expressive enough to accommodate advanced dialogues at different levels of granularity and different levels of abstraction, while allowing some structured design and development of corresponding dialogue. The BCHI Dialogue Editor described in this paper will rely on this conceptual model. For this purpose, our conceptual modeling consists of expanding ECA rules towards *dialogue scripting* (or *behavior scripting*) in a way that is independent of any platform. This dialogue scripting is structured according to a meta-model that is reproduced in Figure 2 that enables defining a dialogue at five levels of granularity:

1. *Object-level dialogue modeling*: this level models the dialogue at the level of any particular object, such as a CIO or a AIO. In most cases, UI toolkits and IDEs come up with their own widget set with built-in, predefined dialogue that can be only modified by overwriting the methods that define this dialogue. Only low-level toolkits allow the developer to redefine an entirely new dialogue for a particular widget, which is complex.
2. *Low-level container dialogue modeling*: this level models the dialogue at the level of any container of other objects that is a leaf node in the decomposition. Typically, this could be a terminal AC at the AUI level or a group box at the CUI level in case of a graphical interaction modality.
3. *Intermediary-level container dialogue modeling*: this level models the dialogue at the level of any non-terminal container of objects, that is any container that is not a leaf node in the container decomposition. If the UI is graphical, this could be a dialog box or the various tabs of a tabbed dialog box.
4. *Intra-application dialogue modeling*: this level models the dialogue at the level of top containers within a same interactive application such as a web application or a web site. It therefore regulates the navigation between the various containers of a same application. For instance, the Open-Close pattern means that when a web page is closed, the next page in the transition is opened.
5. *Inter-applications dialogue modeling*: since the action term of an ECA rule could be either a method call or an application execution, it is possible to specify a same dialogue across several applications by calling an external program. Once the external program has been launched, the dialogue that is internal to this program (within-application dialog) can be executed.

Levels of dialogue granularity

Now that these five levels are defined, we introduced the concepts used towards the conceptual modeling of dialogues that could be structured according to the five aforementioned levels of granularity. These concepts are intro-

duced, defined, and motivated in the next sub-sections.

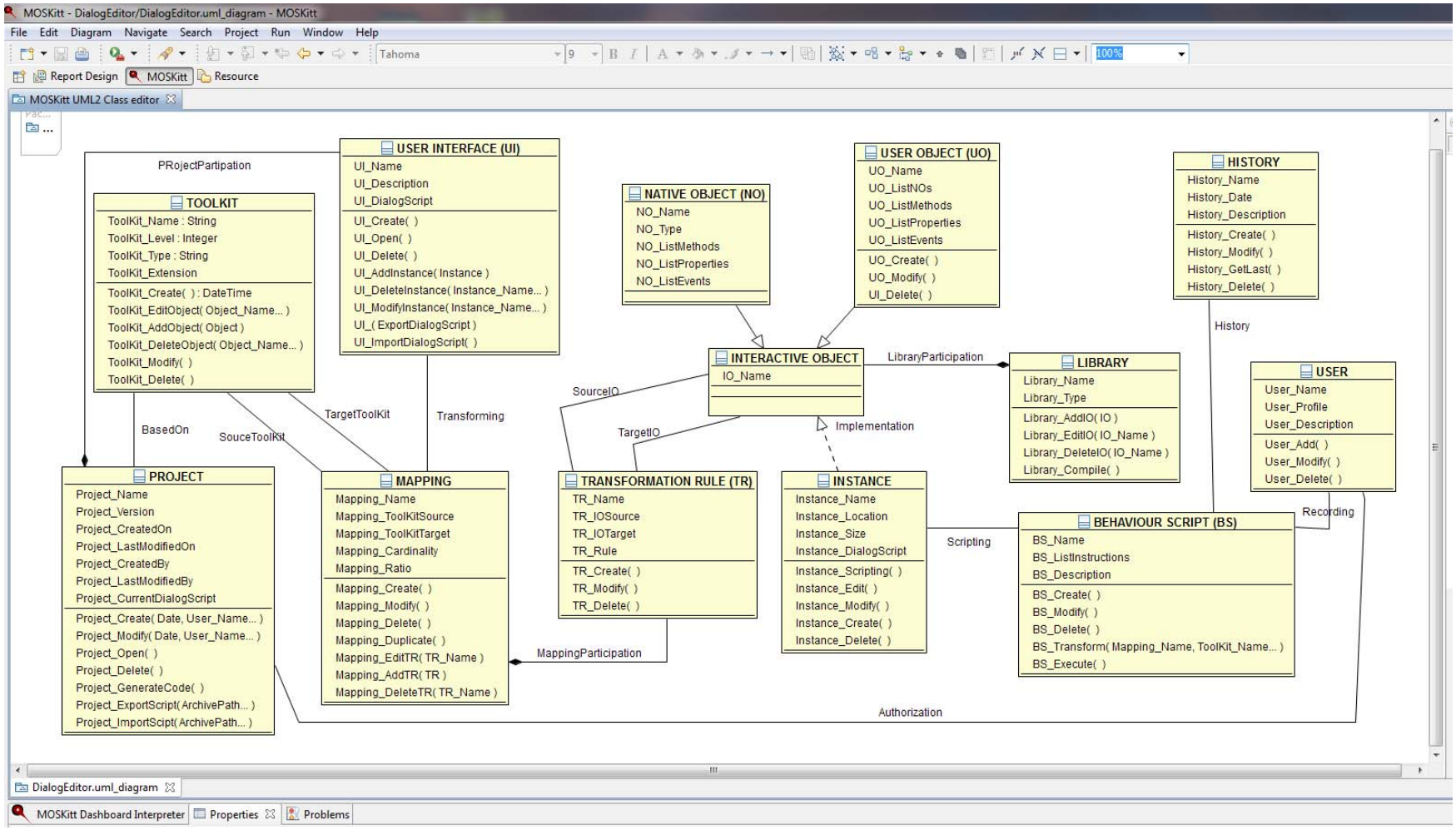

Figure 2. A Conceptual model of dialogue as a basis for model-driven engineering (implemented in Moskitt).

Interactive Object. An *interactive object* is the core component of the conceptual model as it consists of any object perceivable by the end user who could act on it. Interactive objects are further sub-divided into three levels of abstraction depending on the CRF [6]: *abstract*, *concrete*, and *final* (Figure 3 shows how this hierarchy is implemented in the BCHI Dialogue Editor respectively at the three levels).

Abstract Interactive Objects. They describe interactive objects at the Abstract User Interface (AUI) level of the CRF. In the BCHI Dialog Editor, they are implemented as abstract classes compliant with Morfeo's Abstract UI model (http://forge.morfeo-project.org/wiki_en/index.php/Abstract_User_Interface_Model) which has been selected for the following reasons: Morfeo's AUI is one of the most recent effort to define AUI that has been successfully implemented in the Morfeo project and has therefore been recommended as a reference model for European NESSI platform (www.nessi.eu) through the FP7 Nexof-RA project (www. nexofra.eu) which

promotes a ref

erence software architecture for interactive systems, including the GUI part. Morfeo's AUI model holds two object types: an *interactor* manipulates data as input, output, or both, through *simple* interaction mechanism (e.g., a selection) or through *complex* ones (e.g., a vector, a hierarchy); a *container* could contain interactors and/or other containers. Figure 3 details the definition of the abstract class implemented for the *Free* object that serves for general-purpose input/output.

Concrete Interactive Objects. They describe interactive objects at the Concrete User Interface (CUI) level of the CRF. In the BCHI Dialog Editor, they are implemented as abstract classes for one modality at a time. Figure 3 shows that graphical and vocal modalities are included, but only the graphical part is the subject of this paper. Such concrete interactive objects may range from simple widget such as a push button, a slider, a knob to more complex ones such as group box, dialog box, tabbed dialog box.

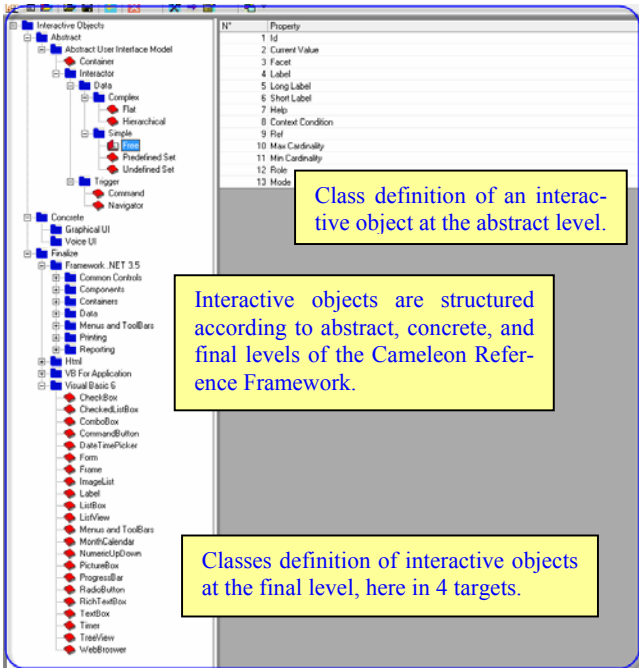

Figure 3. The hierarchy of interactive objects classes as implemented in the BCHI Dialog editor.

If we abstract an interactive object from its various physical representations that belong to the various computing platforms and window managers, any interactive object is be characterized by its attributes and dialogue. An object may react to the end user's actions by handling events generated by this object. Therefore, a class could introduce an abstraction of object characteristics, including its *attributes* (fields or properties), its *methods* (through which a concrete interactive object could be manipulated) and its *events* (that could be generated by, or received by, a concrete interactive object). A class is hereby considered as a model of interactive objects of the same type. For example, a *TextBox* of a GUI consists of a rectangular widget for entering text, characterized by attributes including *width*, *height*, *backgroundColor*, *maxLength* or the *currentText*. Textbox operators are also associated such as *appendText*, *giveFocus*, *selectAll* or *clearEntry*. A textbox generates events such as *textBoxSelected* when the textbox has been selected by any mean (e.g., by clicking in it, by moving the tabulation until reaching the object) or *textBoxEnter* when the GUI pointer enters in the object (e.g., by moving the mouse into it or by touching it).

Final Interactive Objects. They describe interactive objects at the Final User Interface (FUI) level of the CRF. In the BCHI Dialog Editor, they are implemented as real classes corresponding to various toolkits supported (Figure 3 shows the four toolkits that are currently supported with the hierarchy expanded for Visual Basic V6.0). For each interactive object, only the common native dialogue is factored out and rendered as a sub-class of the toolkit. This is why final interactive objects are represented as native objects in Figure 2, while abstract and concrete interactive objects are represented as user-defined classes in Figure 2. We hereby assume that the native dialogue of any final interactive object is preserved. For defining non-native dialogues of a final interactive object, dedicated methods exist, such as the Interaction Object Graph (IOG) [8]. Since defining custom dialogue at the control level requires complex and dedicated programming, it is not supported unless such a dialogue can be characterized as an interactive object.

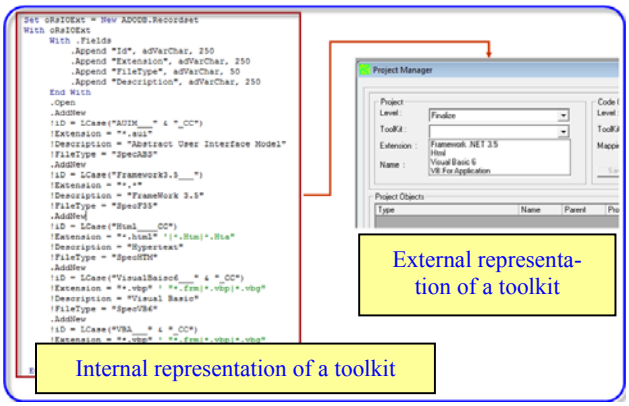

Figure 4. Internal and external representation of toolkits.

Toolkit. In order to support GUIs for multiple computing platforms, each supported toolkit of a particular platform is characterized by its name, its level (e.g., a version), its extensions, and a series of templates describing how this toolkit implements particular dialogues. Three values are accepted depending on which level of abstraction it is considered: abstracted (AUI), concrete (CUI) or final (FUI). Figure 4 shows the correspondence of the external representation of a toolkit that is visible to the end user and the internal representation inside the BCHI Dialogue Editor.

Library. A *library* gathers a series of particular interactive object at any level so as to refer to them as a whole, which is helpful for keeping the same definitions for one target computing platform, typically a toolkit. For the moment, HTML V4.0 is one of the supported toolkits by its corresponding library. Any newer version of HTML, e.g., V5.0, requires implementing a new library for this toolkit.

Instance. An *instance* is any individual object created as an instance of any interactive object class. While a class defines the type of an interactive object, any actual usage of this class is called "instance". Each class instance possesses different values for its attributes. At any *t* time, the instance state is defined by the set of its attributes values. By respecting the encapsulation i.e., the process of hiding all of the attributes of an object from any outside direct modification, object methods can be used to change an instance state. In order to have a login+password, two instances should be created that share the same definition, but with different instance states.

User Interface. A *User Interface* (UI) as it is considered in this conceptual model may consist of any UI at any level of abstract (i.e., abstract, concrete, or final). Therefore, such a UI consists of a set of instances each belonging to the corresponding level of abstraction.

Project. A *project* is considered as a set of UIs for a same case study for a particular toolkit. In a same project, one can typically find one AUI, one CUI, and one FUI. Of course, for the same AUI, different CUIs could be created that, in turn, lead to their corresponding FUIs. Actually, a project could hold as many CUIs and FUIs as model-driven engineering has been applied to the same AUI. This is achieved through the mechanism of mapping.

Mapping. In order to support model-driven engineering, a *mapping* is hereby referred to as any set of transformation rule from one source toolkit to a target toolkit. Note that source and target toolkits could be identical. A transformation rule is written as a PERL regular expression applied from a source class of interactive objects to a target class of interactive objects. In order to support Model-to-Model (M2M) transformation, a transformation rule may be applied from one or many classes of abstract interactive objects to one or many classes of concrete interactive objects. For Model-to-Code (M2C) generation, a transformation rule is applied from one or many classes of concrete interactive objects to one or many classes of final interactive objects (so-called *native objects*). Let us consider again the login and the password example. At the abstract level, two instances of entry fields are created to be mapped onto objects belonging to a particular toolkit. In HTML, both fields are transformed into Input objects, respectively of type Text and Password. In VB6, they are transformed into two text boxes. For the password, *IsPassword* is set to True.

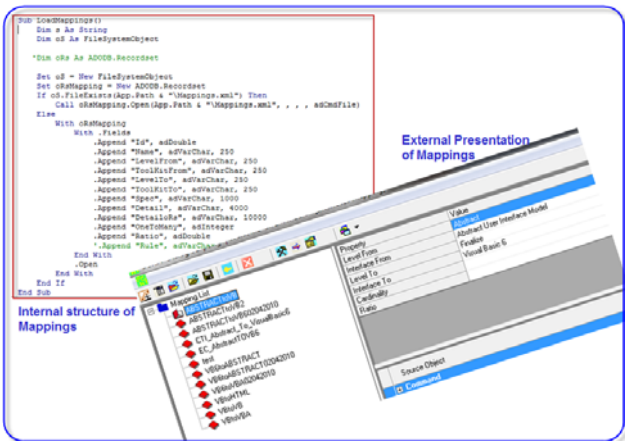

Figure 5. Internal and external representation of mappings.

Figure 5 shows both, the external representation of a mapping that is visible to the end user and its internal representation inside the BCHI Dialogue Editor. Note that the Cameleon Reference Framework [6] enables multiple development paths, and not just forward engineering. In *forward engineering*, transformations are supposed to transform elements of a model into elements belonging to another model whose level of abstraction is inferior (this process is referred to as *reification*). In *reverse engineering*, transformations are supposed to transform elements of a model into elements belonging to another model whose level of abstraction is superior (this process is referred to as *abstraction*). In *lateral engineering*, transformations are applied on models belonging to the same level of abstraction, possibly the same one. Mappings as supported by the BCHI Dialogue Editor support the three types of engineering: (1) *forward engineering*, where mappings transform successively the AUI model into a CUI model that, in turn, is transformed into a FUI for the fours following targets: HTML V4.0, HTML for Applications (HTA), Microsoft Visual Basic for Applications V6.0 (VBA), and DotNet V3.5 framework. HTML V4.0 and HTA are running on both MS Windows and Mac OS X platforms; (2) *reverse engineering*, where mappings transform something concrete into something abstract. Figure 6 depicts a mapping for reverse engineering Visual Basic V6.0 code directly into an AUI model by establishing a correspondence between native objects (Figure 2) and their corresponding user objects (Figure 2), two sub-classes of interactive objects.

Before continuing, we must emphasize that our conceptual and technical choices are guided by a desire to easily integrate our results into the usiXML environment. Indeed, conceptual model of dialogues has been implemented as UML V2.0 class diagram in Moskitt (www.moskitt.org) (Figure 2) that gave rise to a XML Schema.

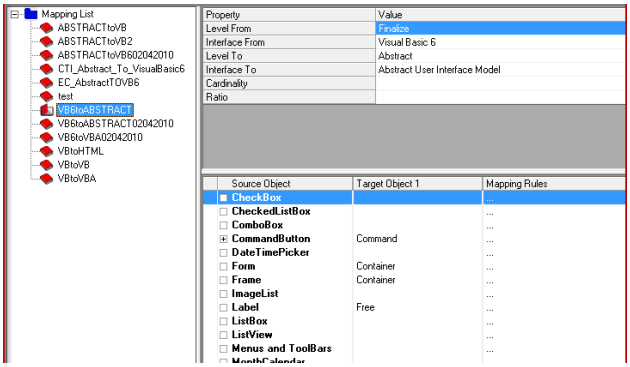

Figure 6. Example of a mapping for reverse engineering.

. Note also that in this example, the reverse engineering does not need necessarily to work between two subsequent levels. The mapping depicted in Figure 6 goes from FUI directly to AUI without passing by the intermediary CUI level. This type of mapping is called *cross-cutting* as it represents a shortcut between two non-consecutive levels of abstraction. For example, Figure 7 depicts a mapping for forward engineering from an AUI model directly to Visual Basic V6.0 code.

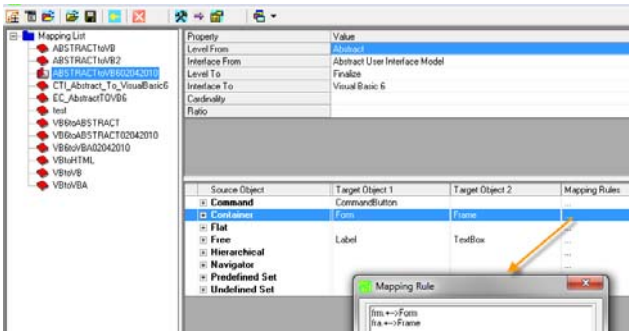

Figure 7. Example of a mapping for reverse engineering.

In the Cameleon Reference Framework, multi-target is also described in terms of different contexts of use. Therefore, any mapping that goes from one context of use to another one is referred to as *lateral engineering*. The BCHI Dialogue Editor also supports this through mappings at the same level of abstraction, but across two different contexts of use, such as between VB6 and HTML V4.0 (Figure 8).

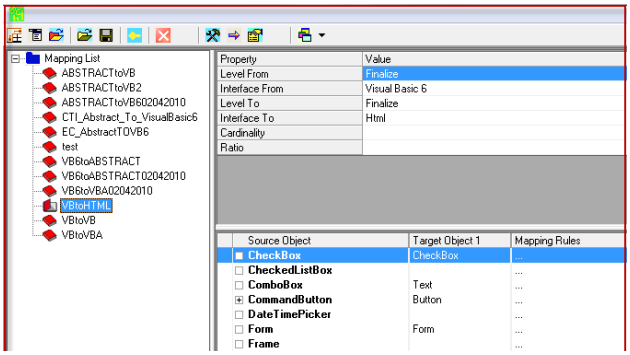

Figure 8. Example of a mapping for lateral engineering.

Dialogue Script. A *dialogue script* (or *behaviour script*) is a sequential text expressing the logic and conditional elements. It describes the actions to be achieved according to a given interaction scenario. An action can be the change of an attribute value, the call of a semantic function belonging to the functional core, or the opening or the closing of another user interface. Three levels of script are possible:

1. *Elementary dialogue scripts*. These scripts are related to instances found in a given project. Often, these scripts are systematically generated accordingly to a template-based approach. They can come from:
   - A change of an attribute value: for example, a read only field implies automatic database requests in its dialogue script ;
   - A layout positioning: for example, two interactive objects may be laid out in their parent according to an adaptation mechanism.
2. *User interface Scripts*. These scripts relate to the implicit or explicit data exchanges between two or several interactive components having a common interactive ancestor. For example, an interactive object is activated or deactivated depending on the state of another object. The verification of a login+password can be initiated only after both fields are properly filled in.
3. *Project scripts*. These scripts express the data exchanges between two or many interactive objects that are independent as they do not share any parent.

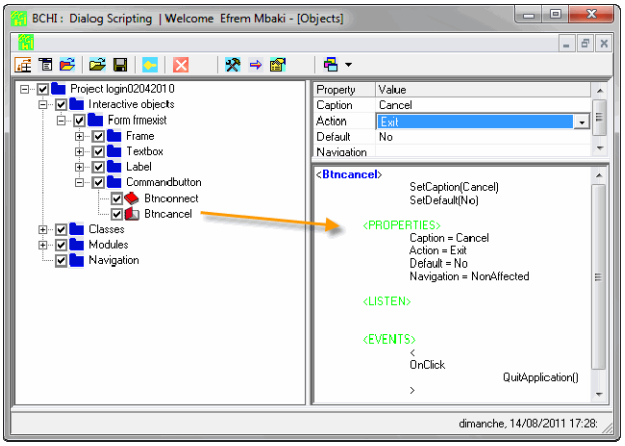

Figure 9. Definition of a dialogue script.

Any dialogue script is structured into three parts (Figure 9): a *condition* of realization, the *event* to consider and a *list of actions* to be undertaken when the event is fired and the condition is satisfied. A single script language has been defined in common for all the three types of dialogue scripts. These scripts use in harmony three models of dialogues; transition networks, grammars, and events [13]. Scripts of dialogues at the abstract and concrete levels are written with a generic language that we described using a BNF grammar. At final level, the code generator translates from generic scripting to specific language relative to a target model. It should be noted that some of these scripts are au-

tomatically deducted through some attribute values. A simple example is to associate the exit of an interactive task with the click of a button. Such a script is generated automatically. As in useML editor [21], other scripts are derived semi-automatically. Indeed, by combining the event of an interactive object to a function call, the developer will have to make the links between the function parameters (input and output) with the attributes of interactive objects. Then, the editor automatically builds the script.

History. A *history* consists of a set of time-stamped operations applied to dialogue scripts over time in order to preserve the design history. In this way, some traceability of dialogue scripts (i.e., who created, retrieved, updated, deleted which dialogue script over tie in the same project) and some reusability (i.e., copy/paste an already existing dialogue script) are ensured (Figure 10). Any dialogue script definition can be validated for a particular toolkit.

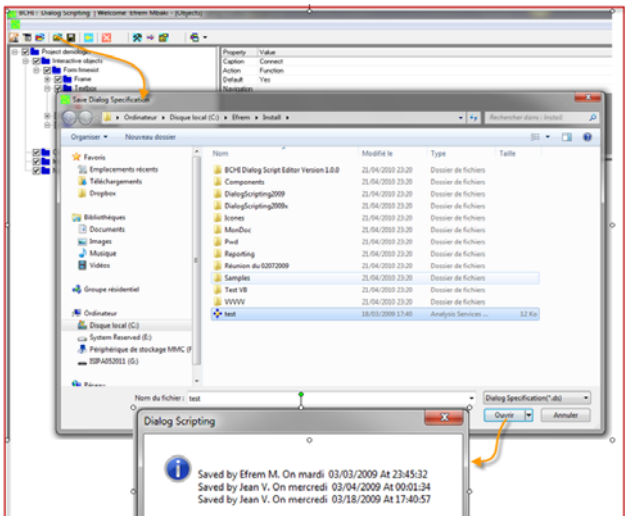

Figure 10. Recovering a previously saved history.

MODEL-DRIVEN ENGINEERING OF DIALOGUES

The four main phases of model-driven engineering

In order to achieve the goal of Model-Driven Engineering of dialogues for multi-platform GUIS based on the conceptual model of Figure 2, the process supported by the BCHI Dialogue Editor (Figure 12) is decomposed into four main phases (Figure 11): (i) *project editing* includes all facilities required to create, retrieve, update, and delete any UI project during the development life cycle; (ii) *project transforming* is aimed at supporting the creation of new mappings between levels and applying them via a mapping editor (Figures 6,7,8 provide significant examples of mappings that support AUI to CUI reification or others), respectively the transformation engine; (iii) *scripting* is aimed at specifying any desired dialogue script at any time, before or after transformation; (iv) *code generating* calls the mappings corresponding to the target platform for which the code of the FUI should be produced.

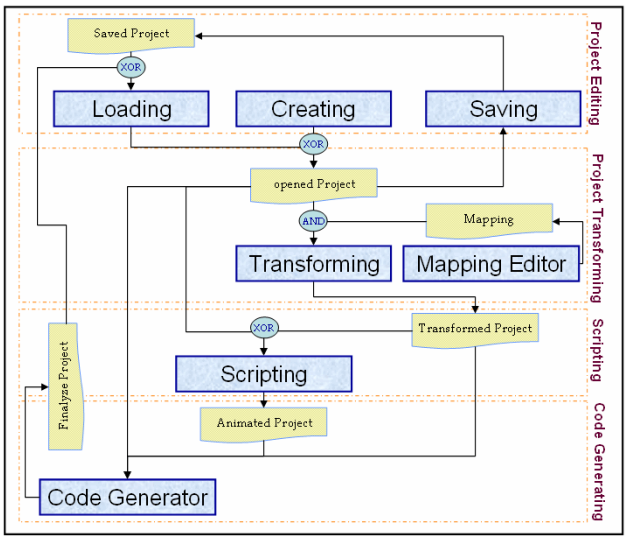

Figure 11. Four phases of dialogue model-driven engineering.

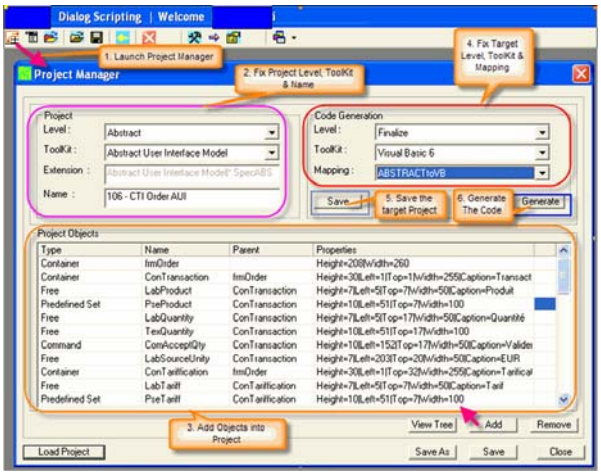

Figure12. The Dialogue Editor for model-driven engineering.

Applying model-driven engineering on a simple case

In this sub-section, an overview is provided of how the four main phases of model-driven engineering are applied on a running example (i.e., the login+password – Figure 13), whose simplicity has been selected for fostering understandability.

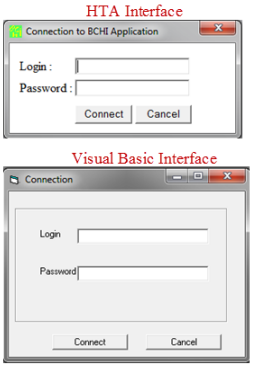

Figure 13. Final User Interfaces of Login+Password.

Project editing. Figure 12 explains the main first steps for creating a new UI Project in the BCHI Dialogue Editor, which basically consists of choosing the starting level of abstraction (typically, the AUI), the ending level (typically, one FUI), the toolkit, possibly with some extension, and the library of mappings to be used. Note that one can start also at any other level such as FUI or CUI since multiple types of mappings are supported.

For the login+password example, we limit ourselves to use five properties: two properties (i.e., *left* and *top*) determine the location of each interactive object, two others properties (i.e., *height* and *width*) specify the dimensions of each interactive object and a fifth property (i.e., *label*) gives the object label text. The values of these attributes are taken into account during future transformations. Therefore, the resulting UI Project holds the login+password with a quintuplet <*Label*, *Left*, *Top*, *Height*, *Width*> for each interactive object (Table 1).

IO Name	Parent	Description	Type	Properties
FrmExist		Main Form	Container	<Connect, 3615,60,450,6360>
fraIdent	frmExist	Secondary Form	Container	<Identification, 2295,120,240,6135>
lbLogin	fraIdent	Login invitation	Free	<Login,300,480,480, 1000>
txtLogin	fraIdent	Login contain	Free	<,300,1200,480,2535>
lbPwd	fraIdent	Password invitation	Free	<Password,300,480,1200,1000>
txtPwd	fraIdent	Password contain	Free	<,300,1200,1200,2535>
btnOk	frmExist	Validation Trigger	Command	<Connect, 300,3000,2880,1455>
btnCancel	frmExist	Cancel Trigger	Command	<Cancel, 300,4800,2880,1455>

Table 1. Interactive objects of the login+password example.

Project transforming. Let us assume that we want to apply Model-to-Model transformation (M2M) from AUI to CUI. For this purpose, Table 2 lists some mappings that have been implemented for this purpose, here for a vocal UI and a GUI, both appearing at the CUI level: *Container* is translated to questionnaire/Form if its name begin by *frm* or SubQuestionnaire/SubForm if its name begin by *fra*. Free object change to Request/Label if its name begin by *lb* or to Answer/Text Box if its name begin by *txt*. Command object is expressed as verbal validation or a button depending on the interaction modality. By applying the mappings for a GUI, we obtain a CUI with a graphical modality.

Code generating. In order to transform this CUI into a FUI (say here that we want both the VB6 and HTA GUIs), Table 3 lists some mappings that have been implemented.

Abstract UI	Vocal UI	Graphical UI
Container	frm* → Questionnaire fra* → Sub Questionnaire	frm*→ Form fra*→ Sub Form
Free	Lb* → Request Txt* → Answer	Lb* → Label Txt* → Text
Command	Validation	Button

Table 2. Mapping from Abstract to Concrete.

Graphic	Visual Basic	HTA
Form/Sub form	frm* → Form fra* → Frame	frm*→ Form/Page fra*→ FieldSet
Text/Label	Lb* → Label Txt* → Textbox	Lb* → Input (Text) Txt* → Input (Text or Password)
Command	CommandButton	Button
Ratio	1	0.05

Table 3. Mappings from Concrete to Final User Interface.

DESCRIPTION OF THE BCHI DIALOGUE EDITOR

Software architecture of the BCHI Dialogue Editor

The software architecture of BCHI Dialog Editor is composed of four components depicted in the UML Class Diagram of Figure 14 corresponding to the four phases of model-driven engineering of dialogues (Figure 11): the *Design* meta-class supplies facilities to edit any UI project (e.g., create, read, update, delete) during the project editing (Figure 11, first swim lane); the meta-class *transformation* manages mappings as defined in Figure 2 (e.g., it enables transforming a UI project of a given toolkit towards another one, possibly the same) in order to support project transforming (Figure 11, second swim lane); the meta-class *Behaviour* manages and interprets scripts at any of the three levels for the scripting (Figure 11, third swim lane); the meta-class *generation* parses any dialogue script, validates it, and transforms into code for a particular target platform in order to support the code generating (Figure 11, fourth swim lane).

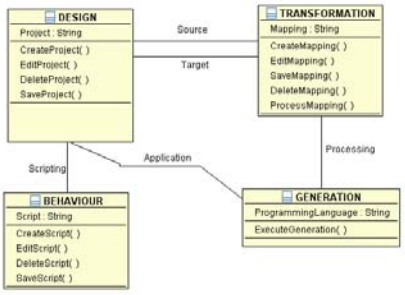

Figure 14. UML Class Diagram of the Dialogue Editor.

Implementation of the BCHI Dialogue Editor

The BCHI Dialogue Editor has been entirely programmed with Visual Studio 6 (Basic 6-VB6) and Visual Basic for Applications (VBA). To standardize the GUI look & feel produced by the BCHI Dialog Editor interfaces, an OCX component has been developed for every interactive object at any level (e.g., Input/output, TextBox, Combox, CheckBox). These OCX components are gathered in a library that is used in the videos demonstrating how the BCHI Dialogue Editor could be used to generate multi-platform dialogues are available on YouTube (Table 4).

Description	URL
Global View	http://www.youtube.com/watch?v=x3CtCj47iZQ

Architecture	http://www.youtube.com/watch?v=Nx3d-w19Oug
Project Editing	http://www.youtube.com/watch?v=GRKWwq5cQzU
Mappings	http://www.youtube.com/watch?v=gVQ8bz9wEXY
Scripting	http://www.youtube.com/watch?v=EZGtL7fXtlE
Code Generation	http://www.youtube.com/watch?v=n7YlgpDihtY

Table 4. Video demonstrations of the BCHI Dialogue Editor.

The underlying conceptual model of dialogues has been implemented as UML V2.0 class diagram in Moskitt (www.moskitt.org) (Figure 2) that gave rise to a XML Schema according to a systematic procedure from Moskitt. Based on this XML Schema, the conceptual model of Figure 2 is stored and maintained in the BCHI Dialogue Editor through RecordSet internal data structures from which an XML file could be exported and to which a XML file could be imported. A RecordSet has been implemented for both native objects (Figure 15) and user objects. The UIDL that is maintained by the BCHI Dialogue Editor is therefore based on this XML Schema. Therefore, any project created in the editor is compliant with the XML Schema (Figure 16).

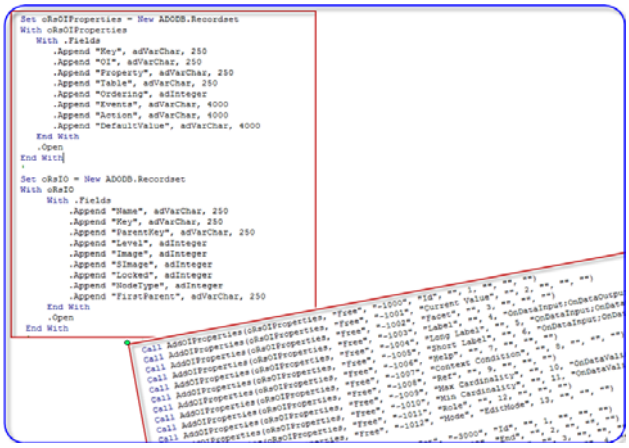

Figure 15. A RecordSet for native objects.

Figure 16. An excerpt of a XML file corresponding to a UI Project according to the UIDL of the BCHI Dialog Editor.

CONCLUSION

This paper introduced an approach for conducting Model-Driven Engineering of dialogues for multi-platform GUIs that are compliant with the CRF [6]. For this purpose, a BCHI Dialogue Editor has been implemented that ultimately automatically generate code for four different targets (i.e., HTML V4.0, HTA, VBA V6.0, and DotNet V3.5) for two different computing platforms (Windows 7 and MacOS X) as a proof-of-concept. The main originality of this editor relies in its capability to always maintain a correspondence between native objects (belonging to the targets) and user objects (at AUI and CUI levels) and to support four types of mappings (i.e., forward, reverse, lateral, adaptation) possibly between two consecutive levels or not (cross-cutting). The BCHI Dialogue Editor however only holds mappings for GUIs only, although interactive objects have been introduced for addressing Vocal User Interfaces. Future work will be dedicated towards this goal and to integrate the conceptual model of dialogue into UsiXML V2.0 in an adequate way.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the ITEA2-Call3-2008026 USIXML (User Interface extensible Markup Language – www.itea2.usixml.org) European project and its support by Région Wallonne DGO6 as well as the FP7-ICT5-258030 SERENOA (Multidimensional context-aware adaptation of Service Front-ends) project supported by the European Commission.

REFERENCES

[1] Ariav, G. and Calloway, L.-J. Designing conceptual models of dialog: A case for dialog charts, *SIGCHI Bulletin*, *20*, 2 (1988) 23–27.

[2] Bastide, R. and Palanque, P. A Visual and Formal Glue Between Application and Interaction. *Journal of Visual Language and Computing, 10*, 5 (October 1999) 481–507.

[3] Book, M., Gruhn, V., and Richter, J. Fine-grained specification and control of data flows in web-based user interfaces. In *Proc. of ICWE'2007* (Como, 16-20 July 2007). LNCS, Vol. 4607, Springer-Verlag, Berlin, 2007, 167–181.

[4] Breiner, K., Maschino, O., Görlich, D., Meixner, G. Towards automatically interfacing application services integrated in a automated model based user interface generation process. In *Proc. of MDDAUI'2009*.

[5] Cachero, C., Melia, S., Poels, G., and Calero, C. Towards improving the navigability of Web Applications: a model-driven approach. *European J. of ISs*, *16* (2007) 420–447.

[6] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers, 15*,3 (2003) 289–308.

[7] Cantera, J.M., González Calleros, J.M., Meixner, G., Paternò, F., Pullmann, J., Raggett, D., Schwabe, D., Vanderdonckt, J. Model-Based UI XG Final Report. W3C Incubator Group Report, 4 May 2010. Available at: http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui/

[8] Carr, D. Specification of interface interaction objects. In *Proc. of CHI'94*. ACM Press, New York, 1994.

[9] Clerckx, T., Van den Bergh, J., and Coninx, K. Modeling Multi-Level Context Influence on the User Interface. In *Proc. of PERCOMW'2006*. IEEE Press, 2006, pp. 57–61.

[10] Cowan, D. and Pereira de Lucena, C. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Trans. on Soft. Eng. 21*,3 (1995) 229–243.

[11] Dittmar, A. and Forbrig, P. The Influence of Improved Task Models on Dialogues. In *Proc. of CADUI'2004*, pp. 1–14.

[12] Elwert, T. Continuous and Explicit Dialogue Modelling. In *Proc. of EA-CHI'96*.

[13] Green, M. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, *5*, 3 (July 1986) 244–275.

[14] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8* (1987) 231-274.

[15] Helms, J., Schaefer, R., Luyten, K., Vermeulen, J., Abrams, M., Coyette, A., Vanderdonckt, J. *Human-Centered Engineering with the User Interface Markup Language*. In "Human-Centered Software Engineering", Chapter 7, HCI Series, Springer, London, 2009, pp. 141–173.

[16] Jacob, R.J.K. A specification language for direct manipulation user interfaces. *ACM Transactions on Graphics, 5*, 4 (1986) 283–317.

[17] Luyten, K., Clerckx, T., Coninx, K., and Vanderdonckt, J. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In *Proc. of DSV-IS'2003*. LNCS, Vol. 2844, Springer-Verlag, Berlin, 2003, pp. 203–217.

[18] Mbaki, E., Vanderdonckt, J., Guerrero, J., and Winckler, M. Multi-level Dialog Modeling in Highly Interactive Web Interfaces. In *Proc. of IWWOST'2008*, CEUR Workshop Proc., Vol. 445, 2008, pp. 38–43.

[19] Menkhaus, G. and Fischmeister, S. Dialog Model Clustering for User Interface Adaptation. In *Proc. of ICWE'2003*. LNCS, Vol. 2722, Springer-Verlag, 2003, pp. 194–203.

[20] Meixner, G., Görlich, D., Breiner, K., Hußmann, H., Pleuß, A., Sauer, S., Van den Bergh, J. Proc. of 4th Int. workshop on model driven development of advanced user interfaces. MDDAUI'2009. In *Proc. of IUI 2009*, pp. 503–504.

[21] Meixner, G., Seissler, M., Nahler, M., Udit – A Graphical Editor for Task Models. In *Proc. of MDDAUI'2009*.

[22] Paternò, F., Santoro, C., and Spano, L.C. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact. 16*, 4 (November 2009)

[23] Pleuß, A. Modeling the User Interface of Multimedia Applications. In *Proc. of MoDELS 2005*, pp. 676–690.

[24] Pleuß, A. MML: A Language for Modeling Interactive Multimedia Applications. In *Proc. of ISM'2005*, pp. 465–473.

[25] Reichart, D., Dittmar, A., Forbrig, P., and Wurdel, M. Tool Support for Representing Task Models, Dialog Models and User-Interface Specifications. In *Proc. of DSV-IS'2008*. LNCS, Vol. 5136, Springer, Berlin, 2008, pp. 92–95.

[26] Rückert, J. and Paech, B. The Guilet Dialog Model and Dialog Core for Graphical User Interfaces. In *Proc. of EIS'2008*. LNCS, Vol. 5247, Springer, 2008, pp. 197–204.

[27] Schaefer, R., Bleul, S., and Müller, W. Dialog Modeling for Multiple Devices and Multiple Interaction Modalities. In *Proc. of TAMODIA'2006*. Lecture Notes in Computer Science, Vol. 4385, Springer-Verlag, Berlin, 2007, pp. 39–53.

[28] Schaefer, R., Bleul, S., and Müller, W. A Novel Dialog Model for the Design of Multimodal User Interfaces. In *Proc. of EHCI-DSV-IS'2004*. LNCS, Vol. 3425. Springer-Verlag, Berlin, 2005, pp. 221–223.

[29] Traetteberg, H. Dialog modelling with interactors and UML Statecharts. In *Proc. of DSV-IS'2003*. LNCS, Vol. 2844, Springer-Verlag, Berlin, 2003, pp. 346–361.

[30] Traetteberg, H. Integrating Dialog Modeling and Domain Modeling – the Case of DIAMODL and the Eclipse Modeling Framework. *JUCS 14*, 19 (2008), 3265–3278.

[31] Van den Bergh, J. and Coninx, K. From Task to Dialog model in the UML. In *Proc. of Tamodia'2007*, pp. 98–111.

[32] van Welie, M., van der Veer, G.M.C., and Eliëns, A. Usability Properties in Dialog Models. In *Proc. of DSV-IS'99*.

[33] Vanderdonckt, J., Limbourg, Q., Florins, M. Deriving the Navigational Structure of a User Interface. In *Proc. of Interact'2003*. IOS Press, Amsterdam, 2003, pp. 455–462.

[34] Winckler, M. and Palanque, P. StateWebCharts: A formal description technique dedicated to navigation modelling of web applications. In *Proc. of DSV-IS'2003*. LNCS, Vol. 2844, Springer-Verlag, Berlin, 2003, pp. 61–76.

[35] Winckler, M., Trindade, F., and Vanderdonckt, J. Cascading Dialog Modeling with UsiXML. In *Proc. of DSV-IS'2008*. LNCS, Vol. 5136, Springer, Berlin, 2008, pp. 121–135.

[36] W3C State Chart XML (SCXML), State Machine Notation for Control Abstraction, Working Draft, 16 May 2008. Accessible at http://www.w3.org/TR/SCXML.