A Model Driven Engineering Design Approach for Developing Multi-Platform User Interfaces

EMAN SALEH[1] , AMR KAMEL[2], AND ALY FAHMY [3]

Department of Computer Science

University of Cairo

EGYPT

(1) eman_maghary@yahoo.com (2) a.kamel@fci-cu.edu.eg (3) a.fahmy@fci-cu.edu.eg

Abstract:- The wide variety of interactive devices and modalities an interactive system must support has created a big challenge in designing a multi-platform user interface and poses a number of issues for the design cycle of interactive systems. Model-Based User Interface Design (MBUID) approaches can provide a useful support in addressing this problem. In MBUID the user interface is described using various models; each describes a different facet of the user interface. Our methodology is based on task models that are attributed to derive a dialog model, from which different concrete models with different appearances can be generated.

This paper presents a semi-automatic Model-Based transformational methodology for multi-platform user interface (MPUI) design. The proposed methodology puts dialog modeling in the center of the design process. A core model is integrated in the design process namely our Dialog-States Model (DSM); which represents our initial step to adapting to multiple target platforms by assigning multiple Dialog-State models to the same task model. A multi-step reification process will be taken from abstract models to more concrete models until reaching a final user interface customized according to the target platform

Key-Words: ConcurTaskTrees, Dialog model, Model-Based User Interface Design, StateCharts, UsiXML.

1   Introduction

To meet the challenges of the diverse and unpredictable number of computing platforms, ad hoc development of the user interfaces is no longer considered acceptable in terms of the cost and time required for software engineering development and maintenance. There is an increasing interest and adoption of Model-Based User interface Approach [6, 9] due to the applicability of the approach in MPUI development. Today, due to the fact that no method has really been emerged from the various attempts to establish a comprehensive Model-Based approach for MPUI design, a standardization process has been adopted by researchers[8, 9, 17], mainly to follow a Model Driven Engineering (MDE) approach by implementing the Model Driven Architecture
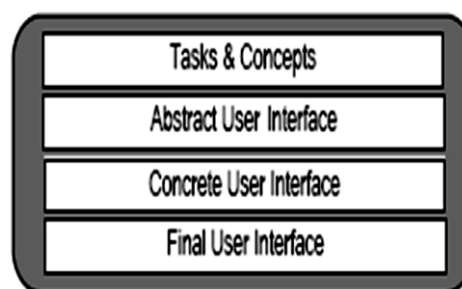

Fig. 1 CAMELEON Framework

(MDA) [13,18] launched by the OMG group [12]. Calvary et al in [8] introduced the CAMELEON Reference Framework; the framework divides the development process into four levels of abstractions (Fig.1 [8, 16].)

Structuring our design process according to this framework and using UsiXML [15] language as the target modeling language supports the creation of MPUIs in a MDE compliant approach [9].

2   Related Work

Many model-based UI design approaches have considered MPUI design and development. In this section we will focus on the most recent related work. Dygimes [10] is a run time environment that automatically generates UIs for mobiles and embedded systems, the environment is a user centered approach, similar to our approach; starts the UI design from task specification using the ConcurTaskTrees (CTT) formalism [2]. TERESA [7]; also based on the CTT; is a transformational approach that enables the design of multi-device UIs with graphical or vocal modalities. TERESA; similar to our approach; is a transformational approach structured according the CAMELEON Reference Framework [8] and followed a forward engineering process; while we use one task model to derive multiple UIs, TERESA requires the designer to specify many task models by filtering the original task model according to the target iplatforms. TransformiXML [16] is a UsiXML tool based on attribute graph grammar; the tool follows a transformational approach following transformation at the same level of abstraction for a different context of use. Another work based on UsiXML is an approach called "Graceful Degradation" [11], the approach aims at creating Multi-Platform UIs by splitting an existing user interface designed for the least constrained platform (e.g. a PC) to a more constrained platform (e.g. a mobile phone),the transformations are semi-automatic but do not follow the CAMELEON Framework. We extended the work done in both TERESA and UsiXML by introducing the Dialog-States model [6], which is more concrete than the task model and more abstract than their abstract user interface model.

Unlike TERESA and UsiXML, our Dialog-States model gives an explicit design of the navigational model, and gives the opportunity to adapt to context of use at early stages of the design process.

3   The Design Methodology

Our methodology aims at producing multiple Final UIs for multiple computing platforms, at design time.

We believe that the navigation structure of the UI is the core aspect of the UI and the most affecting model in Multi Platform context.

One of the major difficulties on designing MPUIs is how to distribute the user interface over the available physical screen space associated with every target device and how to handle the navigation according to this distribution; hence, we are placing dialog modeling (DSM) is in the center of the design process, this also helps to achieve continuity and consistency between the models and to allow designers to predict earlier about the presentation of the user interface. Fig. 2 describes the design process as a four step process supporting forward engineering from the "Tasks & concepts" level to the "Final UI" level as depicted in the CAMELEON Framework:

The following sections will explain the steps of the design process; a case study is used to illustrate the process.
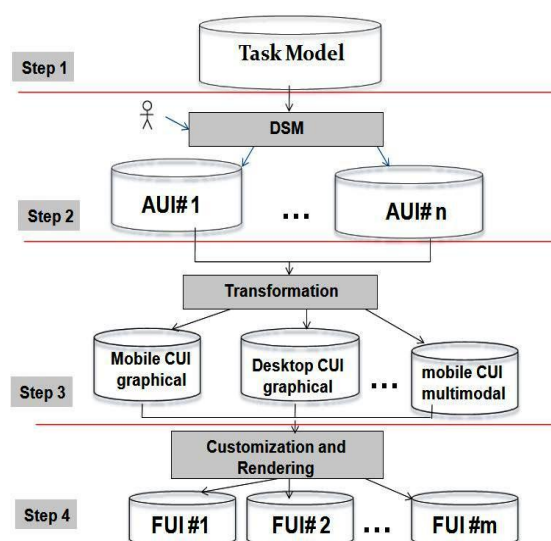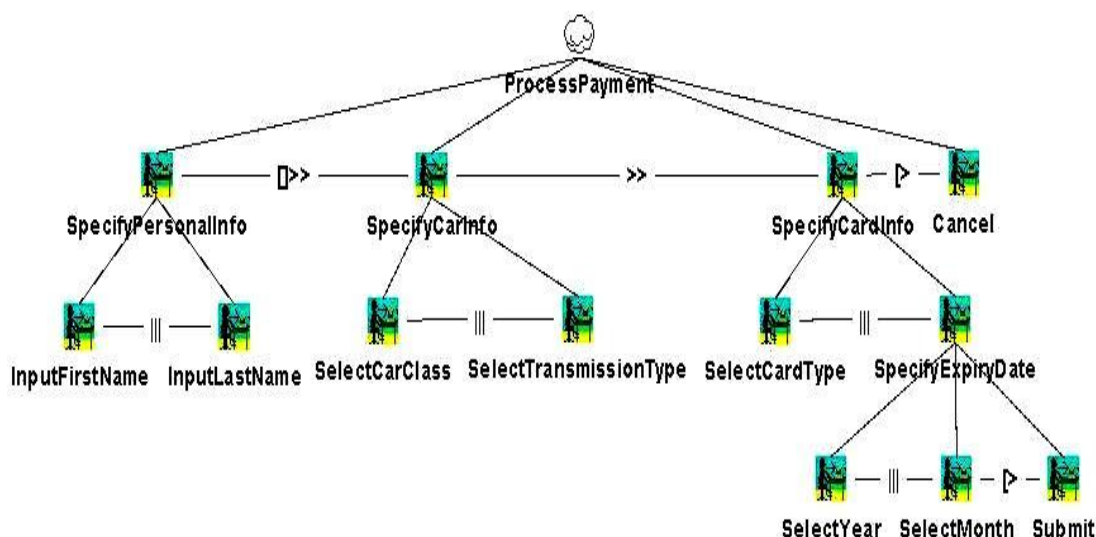

Fig. 2 The design process

Fig. 3 The task model

3.1 Creation of the Task and Domain Model

The task model is expressed using the ConcurTaskTrees (CTT) notation [4]: The designer uses IdealXML [7] tool that enables the creation of the task, domain model and the mapping between them, Fig. 3 show the CTT for the payment task, for a car rental system. Leaf tasks should be specified using two attributes: *userAction* and *taskItem* that enable a refined expression of the nature of the task and are essential in the next transformation to derive the AUI model [15, 5]. The *userAction* indicates a user action required to perform the task. The *userAction* values are: start/go, stop/exit, select, create, delete, modify, move, duplicate, toggle, view, monitor and convey [14]. These are the same values as for the *actionType* attribute for Abstract Individual Components at the Abstract User Interface level [14]. The t*askItem* attribute refers to a type object or subject of an action; which can be: an element, container, operation or a collection of them. For example for the task "EnterName" actionType ="input" and *taskItem* = "Element", the mapping model specifies that this task *maimpulates* an attribute of the domain model.

3.2 Deriving the Abstract User interface (AUI)

Although we are using IdealXML [5] for specifying the task model, we do not rely on the generated enabled task sets that were defined in [10], nor the AUI presented in the tool. The AUI

in IdealXML restricts the navigation since the containment of UI elements corresponds to the user tasks is done based on the level in the task tree; this indicates a level of automation based on the structure of the task model and is a limitation of the approach which is more flexible and can be tailored according to the target device screen size.

A pragmatic approach will be taken in which usability is emphasized over a completely automated transformation. Thus, transformation from task model to AUI model is done by our semi-automated dialog model; the DSM. To derive the AUI from the task model two intermediate sub-steps are performed:

Step 1) Task model to DSM mapping

At this step the DSM; a model based on Harel's StateCharts [3]; is created. This model captures both the containment and the navigation structure of the user interface. The model combines tasks that should be presented to the user at the same time in a state. Hence we define a state in the DSM as the set of all tasks that are logically enabled to start their performance during the same period of time; thus will represent a presentation unit in the user interface. This is similar to the concept of Enabled Task Sets [9, 15].

A dedicated algorithm, automatically computes an initial DSM based on a set of defined semantics of the task types and temporal relationships among tasks [6]. While in [9] and [15] only the enabling operator is considered to represent a new presentation unit, our algorithm considers the enabling operator as a place to create a new state and the concurrent operator when appears at an

intermediate level of the tree. Distribution of necessary disabling and choice tasks among presentation units is taken into consideration. To keep the model consistent with the task specification the algorithm creates bi-directional transitions in case of splitting at a concurrent operator. Hence, this model initially creates the maximum set of states that represent the maximum distribution of the user interface among containers [6], since our initial target is devices with very small screen size.

Step 1.1) Generating the states of the DSM

Considering the semantics of temporal relationships we identify two categories of tasks that are logically enabled at the same period of time, and hence are candidate for composing a state: the *necessary* tasks and *anchorsWith* tasks. The difference between the two categories is that in the first category the relation holds for descendents of the task while it's not the case for the second category. So we define two functions: *Necessary*(t), and *anchorsWith*(t) for the two categories respectively. There are other functions that are used by the algorithm, before explaining the algorithm lets first define these functions:

- *Necessary* (t): Returns all tasks involved with temporal disabling operator ($|>$) with t; at the same level or through parents on higher levels in the tree.
- *anchorsWith*(t):
1. If t is a leaf tasks involved with concurrent ($|||$) operator with another leaf task t' then t' is in *anchorsWith*(t).
2. If t is a task that is linked with the concurrency with information exchange operator ($|[]|$) with t' then first(t') is in *anchorsWith* (t)
3. If t is not leaf, and children(t)={n1,n2,…,nm} are involved in choice [] operator then AnchorsWith(first(n1))={first(n2), first(n3),…, first(nm)}Where the function *first*(t) is defined by:
  - *first* (t): Left subtasks of t that should be executed first, first(t)={first(n1), first(n2), …, first(nm)}, where n1,n2,…nm are child tasks of task t. If t has no subtasks, then first(t) = t. Notice first({t1, ..., tn}) = {first(t1), ..., first(tn)}.
  - *isMarked(t)* : returns true Boolean value if a task is already marked by the mark function.
- The function *createState* (Si, $S_1,S_2,..,S_n$): is a function that creates a composite XOR state with id= Si and contains basic sub-states $S_1,S_2, ..., S_n$; where the initial state of the composite state is the state S_1.

The process of generating the states of the DSM starts at the root of the tree, at every iteration of the inner while loop a depth search for the left-most leaf task is done, which we name an *Anchor* task since it represent a start task for the current subtree. The algorithm generates a composite state that combines this *Anchor* task with its *necessary* and *anchorsWith* tasks (Fig 4).


```
1.  // t1 →ᵒᵖ t2 : indicates that task t1 is linked by temporal operator op to task t2 in
    the task model.
2.  // S: A stack (Last in First out structure)
3.  Create an initial state
4.  Create a final state
5.  S= {Root}
6.  WHILE ( NOT S.isEmpty())
7.    isAnchor = FALSE; // isAnchor means that a leaf task has been encountered
8.    viewdT = S.pop();
9.    WHILE (NOT isAnchor )
10.     viewdT= first(viewdT);
11.     IF( viewdT →» t) OR (viewdT →[]» t) OR ((viewdT →||| t AND NOT Leaf(t))
12.       S.push(t);
13.     IF  (Leaf(viewdT))
14.       isAnchor=TRUE;
15.       i ← i + 1
16.       IF(NOT marked(viewdT)
17.         createState(Si+1,viewdT,necessary(viewedT),AnchorsWith(viewedT);
18.         mark(viewdT)
19.         For ( every t IN anchorsWith(viewedT))
20.           mark(t)
21.           IF(t →» t') OR (t →[]» t') OR (t →||| t' AND NOT Leaf(t'))
22.             S.push(t')
```

Fig. 4 The DSM derivation algorithm

Step 1.2) Detecting the initial state of the DSM

Considering the semantics of StateCharts; a StateCharts model starts in an initial state represented by an arrow with a black circle at its starting end; this initial state will point to the start state of the DSM model, this state contains the *start task* of the task tree which is the *left most leaf task* in the task tree. As the algorithm start creation of the states seeking for the *anchor* tasks from left side of the task tree, the first anchor found is the first task. Thus, the first state created by the algorithm combines the start task with and its *necessary* and *anchorsWith* tasks. Hence, the initial state will point to S0 which the first state created by the algorithm.

Step 1.3): Finding the transitions between the states

To model the dynamic behavior of the user interface, we need to model the navigational part of the dialog model by detecting the actual transitions between the states of the DSM. This is also computed in an algorithmic way. Going back to the states generation process together with the semantics of the task model we can infer that the operators: enabling (>>), disabling ([>) and the concurrent operator (|||) are candidate of transitions between composite states. The first two operators are candidate of transitions whether occurred between leaf of non-leaf tasks while the concurrent operator is only candidate of a transition when occurs between non-leaf tasks, still we can rely on leaf tasks (states) to detect these transitions, because they present places of user interaction with the user interface.

A leaf task is either linked via a temporal operator to another task on its right or not linked to any other task if it is the right most leaf. I call leaf tasks that are not connected to another task via a temporal operator a *LastTasks*.
Transitions either occur due to enabling or disabling temporal operators between leaf tasks or ancestors of *LastTasks* that are involved in concurrent or enabling operators with or without information exchange. For *LastTasks* there are two categiories:

(1) *LastTasks* that do not have an ancestor involved with any temporal relationship with a right hand side task in the task model (e.g. the "Submit" task in Fig. 3); in this case this task is a task that terminates the application and considered as an accept state. Thus, it enables a transition to the final state of the DSM (line 14 of Fig. 5)
(2) *LastTasks* that have an ancestor (we locate the first ancestor) linked with a temporal operator with a task to its right; in this case these tasks enable a transition from their super state to the state that contains the *first* task of the right hand side of that ancestor. Hence we give the following definitions:

Definition 1: *AcceptTasks:*
A task t is in *acceptTasks* if t is in *LastTasks* and t has no ancestor P with $P \overset{op}{\to} t$ in the task model.

Definition 2: *LinkedAncestor(t).*
If a task t is in *LastTasks* and P is the first ancestor of t, if $P \overset{op}{\to} t'$ in the task model and op is in {>>, |||, []>>, |[]|} then P is called the

LinkedAncestor(t).

For the purpose of finding the transitions between the different states of the DSM we need to define the following functions:

- *addTransition*(Si,Sj,L): creates a transition from the state Si to the state Sj labeled with L, note that at this level, task execution represent the triggering events of transitions.
- *superState*(S): returns the parent state of state S.

Based on the above definitions the algorithm in Fig. 5 finds the transitions between the states of the DSM.

```
1.  AddTransition(initialState,S1)
2.  For every state Si in the DSM do
3.     For every substate Sj do
4.        IF  Sj ≫→ Sk
5.           Addtransition(Si, SuperState(first(Sk),Sj)  // if Sk i
6.        IF  Sj [□≫ Sk Then
7.           Addtransition(Si, SuperState(first(Sk),Sj)
8.           Addtransition(SuperState(first(Sk),Si,"BACK")
9.        IF  Sj |||→ Sk Then
10.          IF  NOT Leaf(Sk)
11.             Addtransition(Si, SuperState(first(Sk),Sj)
12.             Addtransition(SuperState(first(Sk),Si,"BACK")
13.       IF  Sk [▷→ Sj  AND Sj In Accepttasks
14.          Addtransition(Si, finalState,Sj)
15.       ELSE // The task is in LastTasks
16.          p= linkedAncestor (Sj)
17.          IF  p ≫→ t
18.             Addtransition(Si, SuperState(first(t),Sj)
19.          IF  p |||→ t
20.             Addtransition(Si, SuperState(first(t),Sj)
21.             Addtransition(SuperState(first(t),Si,"BACK")
22.          ELSE
23.             Addtransition(Si, SuperState(first(p),Sj)
```

Fig. 5 The transitions detecting algorithm

The initial DSM created for our example (the task tree in Fig 3) is shown in Fig. 6.

Step 1.4) Creating multiple Dialog State Models

After computing the initial DSM, the designer can refine this model and/or create one or more Dialog-State models, each for a target platform by merging states; hence the DSM is our initial step in handling adaptation to context of use; (device screen size at this phase); by mapping the same task model to different DSMs. For example the designer can save both DSMs in Fig. 7.
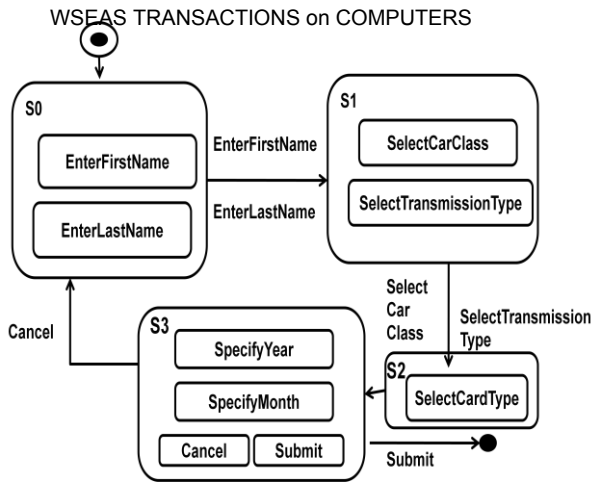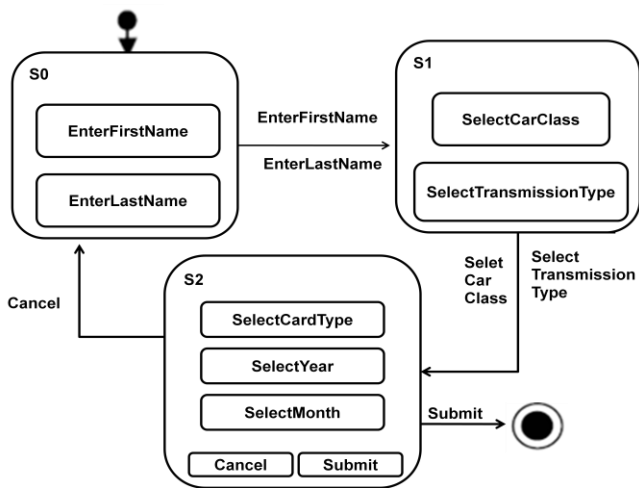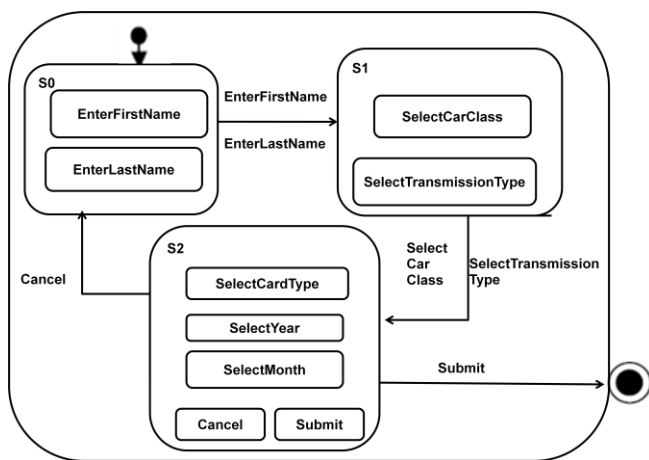
Fig. 6 The initial DSM

Note that in Fig 7(b) the designer combined the three states into one compound state according to target screen size.

(a)

(b)

Fig. 7 Two possible combinations of the initial DSM

It is the designer responsibility to create multiple Dialog-States models from the initial DSM. Some guiding rules are necessary to ensure consistency with the task model specification, on top of all merging must start with states that represent higher level number in the tree, where the root is at level number=0 (i.e. merging is a bottom-up process). As the concurrent and enabling operators were the basis of creating new state, the state combination process must merge these states by including these tasks in the same state. States that have been created due to concurrent operator has a higher priority over those that have been created due to enabling operator and hence will be combined first. The main heuristics that must be followed in top-down logical order, when merging states are:

H1) If two states are different by one basic state merge the states into one single state

H2) If a state contains only one task it can be merged with its successor state or predecessor according to linking task temporal relation applying H1 and taking into consideration that precedence of merging concurrent tasks is higher than merging tasks linked with enabling operator.

H3) Merge states that contain tasks linked with concurrent operator in an outer composite state.

H4) Merge states that contain tasks linked with enabling operator in an outer composite state.

Consider the tree in Fig. 8, three states are created initially by the algorithm. The designer cannot combine S0 with S2 since there is no common parent following the logical order of heuristics the combining should be in a bottom-up order where concurrent tasks should be combined first: Combining states may done in the following order:

First the designer can combine S1 and S2 into one state, the resultant state can be combined with S0 into a composite state.

Now let's consider that task7 and task 8 are linked with enabling operator then they will belong to different states; these states should be combined
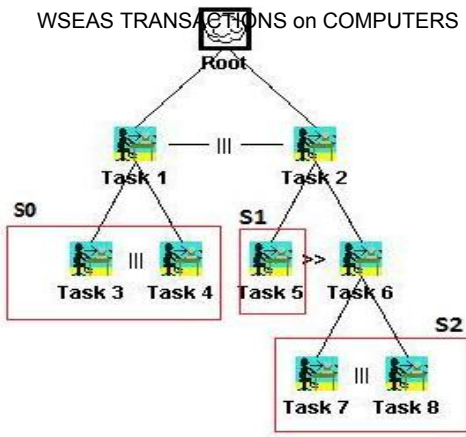
Fig. 8 State composition example

first then the state that contains the combination can be combined with S1. This ensures consistency with the task execution order as specified in the task model.

A post processing stage is needed to:

1) Deleting redundant states and replace by one in the *top super state.* These redundant states correspond to disabling tasks which are usually have a navigation or control facet thus they insure the navigation between the states of the dialog states model, thus we place them in the outer top *SuperState* to avoid repeating transitions.

2) Replacing transitions that goes from an *intermediate sate* to an outer state by a transition from the Top *SuperState* to the outer states

Step 2): DSM to AUI mapping:

The AUI in UsiXML is composed of Abstract objects: Abstract Containers (ACs) and Abstract Individual Components (AICs) [14, 16], at this step we map composite states to ACs and basic states AICs, then assigning the suitable facets to the AICs, also we define both the navigation and control between AUI elements. Table 1 presents the potential mappings between the two models constructs.

According to transformation rules, Each of the DSM in Fig. 7 will be mapped to an AUI model. The DSM in Fig. 7(a) will be transformed to an AUI with three containers, the First Conainer will

contain two AICs, corresponding to the basic states an extra AIC with navigation facet will be added to replace the transition that goes out of the state, to ensure navigation while for the DSM in Fig 7(b) an AC that embeds three ACs will be created.

Table 1 mappings between the DSM and the AUI in UsiXML

DSM Construct	UsiXML AUI model construct
Basic state	AIC
Composite State	AC
Transition	*abstractDialogControl* relationship + AIC with navigation facet
Hierarchy	*abstractContainment* relationship

Each AIC can be equipped with facets describing its main functionality (input, output, Navigation and control) [14]. These facets are derived from the combination of task model, domain model and the mappings between them, using transformation rules, as these listed in table 2.

Table 2: Mapping between task attributes and AIC facet types

UserAction	TaskItem	Facet
Create	Element	Input
Select	Element	Input
Start	Operation	Navigation / control
Convey	Element	Output
Start	Container	Navigation

Transitions between the states of the DSM are modeled by assuming sequential navigation and Global placement on interaction components (i.e *NEXT* button is placed in the outer container); that is done by a transformation rule that creates AICs with navigation facet (*NEXT, PREVIOUS* buttons at the next step) and placing them in the outer Container (line 15-17 Fig. 9). At the AUI dialog control between Abstract Objects is ensured by *dialogControl* relationship, using LOTUS operators (lines 58-.69.Fig. 9)

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <auimodel>
3   <abstractContainer id="ACS0" name="Provide Personal Data">
4       <abstractIndividualComponent id="AIC01"
5        name="Input First Name">
6       <facet id= "FA011" TYPE="input"
7        name="create firstName"
8        actionType="create" actionItem="element" dataType=String"/>
9       </abstractIndividualComponent>
10      <abstractIndividualComponent id="AIC02" name="Input Last NAme">
11          <facet id= "FA012" TYPE="input"
12          name= "create lastName" actionType="create"
13          actionItem="element" dataType=String"/>
14      </abstractIndividualComponent>
15      <abstractIndividualComponent id="AIC03" name="Next">
16          <facet id= "FA013" TYPE="navigation" name="Neaxt"
17          actionType="start" actionItem="operation" dataType=String"/>
18      </abstractIndividualComponent>
19  </abstractContainer>
```

```
58  <auiDialogControl symbol=">>">
59      <source sourceId= "AIC01"/>
60      <target targetId= "AIC02"/>
61  </auiDialogControl>
62  <auiDialogControl symbol=">>">
63      <source sourceId= "AIC03"/>
64      <target targetId= "ACS1"/>
65  </auiDialogControl>
66  <auiDialogControl symbol=">>">
67      <source sourceId= "AIC01"/>
68      <target targetId= "AIC02"/>
69  </auiDialogControl>
```

Fig. 9: Part of the AUI model expressed in UsiXML

3.3 Mapping the AUI to Concrete User Interface (CUI) Model

This level is modality dependent, at this level the designer chooses the target modality, currently we only consider graphical modality. In UsiXML the CUI is populated by *Concrete Interaction Objects* (CIO's) and *Concrete User Interface relationships* between them. For graphical modality UsiXML further classifies graphical CIO's in two categories: *graphical containers* (GCs) and *graphical individual components* (GIC). A GC is a graphical CIO that can contain other CIO's, including other containers. UsiXML's metamodel [14,15] contains a list of 11 types of containers such as: dialog box, menu bar, menu pop-up, tool bar, status bar, window and box. GIC's are a direct abstraction of widgets found in popular toolkits. For example, UsiXML's *checkbox* component corresponds to <INPUT TYPE = CHECKBOX> in HTML 4 or JCheckBox in Java Swing. The list of GICs in UsiXML includes: *text component*, *button*, *radio button*, *checkbox*, *combobox*, etc. [14]. Dialog control relationship can be defined between both types of interaction objects. We derive the CUI by set of transformation rules: mapping AC to Graphical containers (GCs), AICs to graphical Individual components (GICs), some of these rules are shown in table 3.

Many other rules are available for matching the target platform, for example an AIC with input facet and *actionType*=select can be mapped also to radio button group if the target platform supports this widget. Other rules as resizing rules can be applied; for example to change the font size and picture size. The dialog control relationship at this level is a reification of the dialog control relationship at the AUI, transitions at the DSM which where mapped to AIcs with navigation facet will be transformed to *NEXT-PREVIOUS* buttons at this level, that are endowed with *graphicalTransition* relation[14]. That enables giving them an activate/deactivate power. Two rules are applied here:

R1: Endow the OK button with *graphicalRelationship* type= "*graphicalTransition*" and *transitionType* = "*activate*".

R2: Endow the Cancel button with *graphicalRelationship* type= "*graphicalTransition*" and *transitionType* = "de*activate*".

Table 3 Mapping AUI components to CUI components

AUI(AIC)		CUI(GIC)
Facet	Type	
Input	Create element	create two GICs: An input text and an output text(for the label)
Input	Select element	create two GICs: A *list box* and an *output text*(the label), for every value in the tag *<selectionValue>* create an *item* in the *list box.*
Navigation	Start operation	Create GIG of type button.

3.4 From CUI to Final User Interface (FUI)
After the code of the CUI is produced, this code could be either interpreted or compiled by a rendering engine. UsiXML can be rendered by set of rendering engines (e.g. GrafiXML, FlashiXML, QtkXML, InterpiXML)[9].
The FUI for the DSM in Fig. 7(a) and Fig. 7(b), are shown in Fig. 10 and Fig. 11, as previewed by GrafiXML [1] tool.
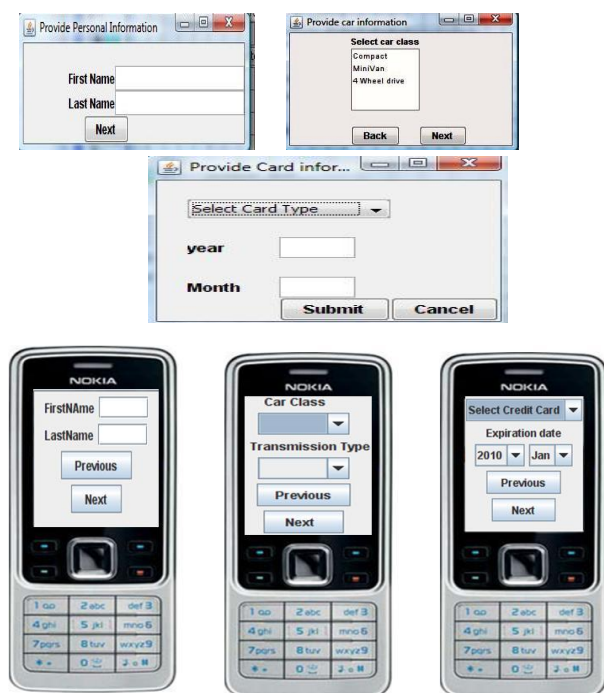
Fig. 10 FUI for DSM in Fig. 7(a) as previewed by GrafiXML and possible presentation on a mobile
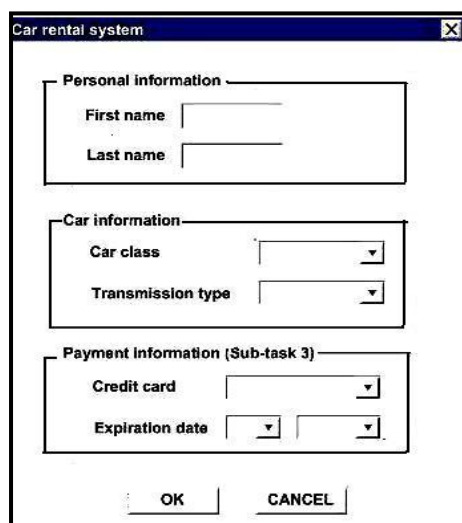

Fig. 11 FUI for DSM in Fig. 6(b)

4 Conclusion

In this paper we presented a MDE transformational approach to design MPUI, the design process is structured according to the CAMELEON Reference Framework and the target modeling language is UsiXML. A core model is integrated in the design process to adapt to multiple platforms multiplatform screen size limitations by designer intervention. The approach is more feasible than fully automatic approaches

from usability view point. The proposed methodology uses set of tools for model based UI development, storing the models in a model repository allows reusability of the models for new target devices. Future work will focus in combining these tools as a tool chain embedded in a modeling framework, also taking other parameters of the context of use model (environment and user) into account, and considering other modalities.

References:

[1] B. Michotte, and J. Vanderdonckt, "GrafiXML, A Multi-Target User Interface Builder based on UsiXML," Proc. of 4th International Conference on Autonomic and Autonomous Systems ICAS'2008 , *IEEE Computer Society Press,* Los Alamitos, 2008.

[2] CTTE: The ConcurTaskTrees Environment http://giove.cnuce.cnr.it/ctte.html, 2009

[3] D. Harel, *StateCharts: A Visual Formalism for Complex Systems*, Science of Comp. prog., 1987.

[4] E.Saleh, A. kamel, and A. Fahmy, "Dialog States a multi-Platform Dialog model", ECS journal, vol. 33, No. 2, Sep. 2009, pp 1-9.

[5] F. Montero, V. Víctor López Jaquero, J. Vanderdonckt, P. Gonzalez, M. Lozano, and Q. Limbourg, Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML, *Lecture Notes in Computer Science*, Vol. 3941, Springer-Verlag, Berlin, 2005, pp. 161-172.

[6] F. Paterno, *Model-Based design and Evaluation of Interactive Applications*. Springer-Verlag, London, 1999.

[7] F. Paterno, and C. Santoro, One model, many interfaces, In *Christophe Kolski and Jean Vanderdonckt, editors*, CADUI 2002, VOL 3, 2002, pp. 143-154.

[8] G. Calvary,J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouil-lon, and J. Vanderdonckt, A Unifying Reference Framework for Multi-Target User Interfaces, Interacting with

Computers, Vol. 15, No. 3, June 2003, pp. 289-308.

[9] J. Vanderdonckt, Model-Driven Engineering of User Interfaces: Promises, Successes, and Fail-ures, *Proc. of 5th Annual Romanian Conf. on Human-Computer Interaction ROCHI'2008* , Bucarest, 2008, pp. 1-10.

[10] K. Luyten, T. Clercks, K. Coninx, and J. Vanderdonckt, Derivation of a Dialog Model from a Task Model by Activity Chain Extraction, Proc. Of DSV-IS2003, Spriger-Verlag, 2003, pp. 203-217

[11] M. Florins, F. Montero, J. Vanderdonckt, and B. Michotte, Splitting Rules for Graceful Degradation of User Interfaces, In *Proc. of 10th ACM Int. Conf. on Intelligent User Interfaces IUI'2006*, ACM Press, New York, 2006, pp. 264–266.

[12] OMG: The object management Group: http://www.omg.org

[13] OMG: Model Driven Architecture available at: http://www.omg.org/mda/mda_files/02F-SIW-004-OMG.pdf .

[14] UsiXML documentation version 1.8.0, available at: http://www.usixml.org/index.php?mod=download&file=usixml-doc/UsiXML_v1.8.0-documentation.pdf

[15] Q. Limbourg, , J. Vanderdonckt, ,B. Michotte, and L. Bouillon and V. López , UsiXML: a Language Supporting Multi-Path Development of User Interfaces, *Lecture Notes in Computer Science,* VOL. 3425, Springer-Verlag, Berlin, 2005, pp. 200-220.

[16] Q. Limbourg, J. Vanderdonckt, Transformational Development of User Interfaces with Graph Transformations, *Proc. of the 5th International Conference on Computer-Aided Design of User Interfaces CADUI'2004*, Madeira, Kluwer Academics Publishers, Dordrecht, 2004.

[17] A. Mahfoudhi, W. Bouchelligua, M. Abed, and M. Abid, Towards a new approach of model-based HCI Conception, Proceedings of the 6th WSEAS International Conference on Multimedia, Internet & Video Technologies, Lisbon, Portugal, September 2006, pp. 22-24,

[18] A. MOHAMED, N. ARSHAD, N. HIDZIR Model-Based Computer Science Curricula Design, Proceedings of the 6th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, Corfu Island, Greece, February 2007.