Systematic Generation of Abstract User Interfaces

Vi Tran[1], Jean Vanderdonckt[1], Ricardo Tesoriero[1,2], and François Beuvens[1]
[1]Université catholique de Louvain, Louvain School of Management
Louvain Interaction Laboratory, Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium)
[2]University of Castilla-La Mancha, Albacete (Spain)
{vi.tran, jean.vanderdonckt, ricardo.tesoriero, francois.beuvens}@uclouvain.be, ricardo.tesoriero@uclm.es

ABSTRACT
An abstract user interface is defined according the Cameleon Reference Framework as a user interface supporting an interactive task abstracted from its implementation, independently of any target computing platform and interaction modality. While an abstract user interface could be specified in isolation, it could also be produced from various models such as a task model, a domain model, or a combination of both, possibly based on information describing the context of use (i.e., the user, the platform, and the environment). This paper presents a general-purpose algorithm that systematically generates all potential abstract user interfaces from a task model as candidates that could then be refined in two ways: removing irrelevant candidates based on constraints imposed by the temporal operators and grouping or ungrouping candidates according to constraints imposed by the context of use. A model-driven engineering environment has been developed that applies this general-purpose algorithm with multiple levels of refinement ranging from no contextual consideration to full-context consideration. This algorithm is exemplified on a some sample interactive application to be executed in various contexts of use, such as different categories of users using different platforms for the same task.

Categories and subject descriptors
D.2.2 [**Software Engineering**]: Design tools and techniques – *User Interfaces*. H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical User Interfaces*. I.7.2 [**Document and text processing**]: Document preparation – *Markup languages*.

General Terms
Algorithms, Human Factors.

Keywords
Abstract User Interface, Concrete User Interface, Graphical User Interface, Model-based User Interface Design, Model-Driven Engineering, User Interface Description Language, User Interface eXtensible Markup Language.

INTRODUCTION
The Cameleon Reference Framework (CRF) [3] is a conceptual and methodological framework that structures the User Interface (UI) development life-cycle according to four levels: task and domain, abstract user interface, concrete user interface, and final user interface. In this CRF, an *Abstract User Interface* (AUI) is defined as a UI supporting an interactive task that is specified in a way that does not refer to any peculiarity belonging to the implementation world. The AUI is specified independently of any target computing platform and interaction modality that could be used for such a UI. More recently, the final report of the W3C Incubator Group on Model-Based User Interface Design [4] agreed upon the following definition:

"The Abstract User Interface (AUI) (corresponding to the Platform-Independent Model– PIM– in Model-Driven Engineering) is an expression of the UI in terms of interaction spaces (or presentation units [1]), independently of which interactors are available and even independently of the modality of interaction (e.g., graphical, vocal, haptic …). An interaction space is a grouping unit that supports the execution of a set of logically connected tasks." [19]

In order to adhere to this standard definition, UsiXML V2.1 [19] instantiates this definition by defining an AUI as a hierarchy of *Abstract Interaction Units* (AIUs), each AIU expressing the input/output required to conduct a particular task or set of semantically related sub-tasks of a task over a given domain of discourse. For this purpose, different types of AIUs are defined and may contain any AIU type.

When interested in generating a AUI from a task model and a domain model, we are generally confronted with a dilemma: on the one hand, the AUI definition should remain independent of any platform and interaction modality in order to preserve this property of independence (otherwise, the AUI is no longer called abstract); on the other hand, a general trend consists in trying to optimize the definition of potential AUIs having already in mind the constraints imposed by the target platform and interaction modality. For instance, when one desires to produce an AUI for a smartphone, consciously or unconsciously, the AUI being defined is already taking into account the constraints imposed by the target computing platform (e.g., a particular operating system on a mobile phone having a specific screen resolution) and/or the intended interaction modalities (e.g., a graphical user interface equipped with bi-touch capabilities). When the target computing platform is already decided, going through the AUI step is no longer required. Therefore, the phase of defining an AUI could be skipped. However, it may still be interesting to identify all potential AUIs that could result from a task model and then

EICS'12, June 25–26, 2012, Copenhagen, Denmark.

decide by progressive refinement which ones could be the most suitable for a certain context of use for a given development environment.

This paper addresses the problem of systematic generation of abstract user interfaces: first, an algorithm is provided that automatically generates all potential AUIs from a same task model; then, only AUIs relevant for a given context of use could be kept by comparing different candidates against various criteria constrained by the context of use.

Section 2 reviews the work related to the problem of identifying AUIs from a task model. Section 3 defines the three meta-models that will govern the process of systematic generation of AUIs; the task meta-model as a starting point, perhaps with the domain meta-model, and the AUI meta-model as a target point. The process of systematic generation of AUIs is detailed in Section 4, first at the outline level, then at a detailed level. The software tool that supports this process is described in Section 5. Section 6 concludes this work by summarizing the main aspects of this process and by presenting some avenues of this work.

RELATED WORK

The problem of automatically generating a UI from one or many models has been addressed extensively by many different approaches [5,7,11,13,16,17], but the problem of determining AUIs [1,8,10,12,15,18] from one or many initial models so as to initiate this process has been partially addressed. In *forward engineering*, significant work has been produced to generate one or many UIs from initial models for multiple contexts of use. Only a few of them go through the AUI level: IKnowU [8] fires rules for generating AUIs for different platforms based on task, domain, and context models. In *reverse engineering*, significant work has been produced to recover a CUI model from code (e.g., from HTML). Few of them goes until the AUI level: ReversiXML proceeds with the abstraction process until the AUI level. The observation is similar for UI retargeting, UI adaptation, etc. [20]

Bogdan *et al.* [2,7] take benefit of a discourse model to generate different AUIs for different target devices such as mobile phone, PC or PDA. However, these abstract user interfaces are completely independent of the programming languages such as Java Swing, AWT, or Windows-Forms. Discourse models are created based on human communication theories, in other words they are used to describe the human communicative acts. This approach generates UIs for multiple devices, but unfortunately its determination of the control types is not detailed. For example, one communicative act can be mapped to many control types such as a *Closed Question* communicative act mapped to a radio buttons, or a check box, or a menu.

ROAM [5] automatically generates UIs for heterogeneous platforms classified based on their capabilities such as processors, memory, screen size, and software libraries. It allows the user to migrate a UI from a specified platform to another one, provided that ROAM is installed on both.

SUPPLE [9] automatically generates a UI that is adapted to a person's devices, task, preference and abilities. This system uses various input materials to generate UIs including: user, device and task models. Moreover, it also uses a *rendering and optimization algorithm* to search the space of possible interface objects to adapt its container once the container's size has been changed by the user; and uses the *cost function* to generate styles of final user interfaces based on different parameters. Supple considers both user and device models and use well the optimization algorithms to generate user interface addressing user's subjective preferences and device capabilities.

Most approaches discussed above attempt generating a complete and executable UI for multiple platforms by using different algorithms from simple ones such as in Desktop-to-Mobile [13] or ROAM [5] to complex ones such as in SUPPLE [9]. ROAM requires that the UI designer provides various implementations for multi-platforms. In order to generate the UIs for different platforms, the UI designer has to create the different layout structures; one layout structure created is suitable to a device. Then she has to link these layout structures to the task model manually. Most steps of the UI generation process in the Supple system are performed automatically but it has also some limits e.g., this system directly generates CUIs instead of AUIs.

TOPCASED AUI [14] is an Eclipse-based modelling editor enabling designers to specify AUIs directly, without necessarily taking a task model as a starting point. This freedom has a cost: there is no analysis of the quality of AUIs resulting from this manual process.

In some aforementioned works, the use of temporal operators to link the tasks at the same abstract level has not been discussed yet. For example, in [8], the containers are specified based on the groups of tasks, but these tasks are grouped together without checking the operators between them; in practice, tasks that can also exclude each other so they cannot be grouped together. The ROAM system does not exploit the semantics of temporal operators in the generation process.

META-MODELS

This section describes the main models used in this paper including task, domain and AUI models. These models are defined within the UsiXML V2.1 framework [19].

Task Meta-Model

The task meta-model (Figure 1) represents the task decomposition view of the application in the Tasks & Concepts layer of the UsiXML framework. Inspired by the Hierarchical Task Approach (HTA), the task decomposition is defined for the tasks that can be performed independently of the situation in which a task is performed. The temporal relationship is also provided in this model. However, more information to the relationship can be added by a context model. Thus, the temporal relationship among tasks depends on the context and varies according to the context situation in which a task is performed.

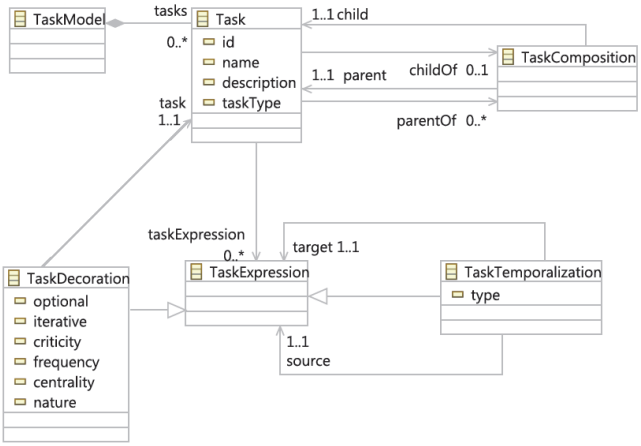

Figure 1: Task meta-model overview.

Domain Meta-Model

The UsiXML domain model describes the various entities manipulated by a user while interacting with the system. This model specifies the main concepts of a User Interface by identifying the relationships among all the entities within the scope of the User Interface, their attributes and the methods encapsulated within the entities. The UsiXML domain model uses the UML V2.0 class diagram to describe the different entities manipulated by a UI. A UML class diagram is a type of static structure diagram which provides a rich expressiveness to describe the structure of a system by using the classes, their attributes, and the relationships between the classes.

Abstract User Interface Meta-Model

The AUI model is aimed at specifying the end user interaction in terms of concepts that do not make any reference to any concrete platform or modality, which is done via the *Concrete User Interface* (CUI) [3,11,12]. Usually, AUIs are specified independently of platform and devices so that the various CUIs can be created from a single AUI. The AUI meta-model used in this paper is depicted in Figure 2 [19]. The AUI, corresponding to the Platform-Independent Model (PIM) in Model-Driven Engineering (MDE) is an expression of the UI in terms of interaction spaces (or presentation units), independently from interaction units available and from the interaction modality (e.g., graphical, vocal, haptic). An interaction unit is a grouping unit that supports the execution of a set of logically connected tasks.

ABSTRACT USER INTERFACE GENERATION

This section describes the AUI systematic generation process from task and domain models at the outline and detailed levels. At the outline level, we discuss the engineering process, the role of its components and the resources used in this process in order to identify the responsibilities to be taken for ensuring this step. At the detailed level, the main steps of the process will be overviewed as well as the mapping rules and the algorithms used so as to identify the actions that will be needed by the responsible entities of the outline level.

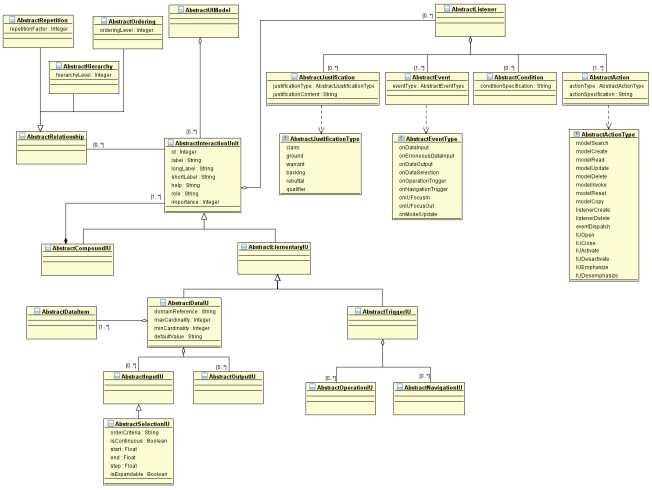

Figure 2: AUI meta-model.

Process at the Outline Level

AUI generation process and its main components are presented with respect to the UML profile for SPEM V2.0 (Software & Systems Process Engineering Metamodel specification - http://www.omg.org/spec/SPEM/2.0/). After describing components of the AUI generation process (drawn as a package), a workflow is presented and specified that details package activities and work-products.

AUI Generation

Figure 3 shows the principal components of the AUI generation process including three process-roles and six work-products. These work-products are the three UsiXML models (i.e., task model, domain model and AUI model) and three text documents (i.e., mapping rules, platform information and algorithm document).

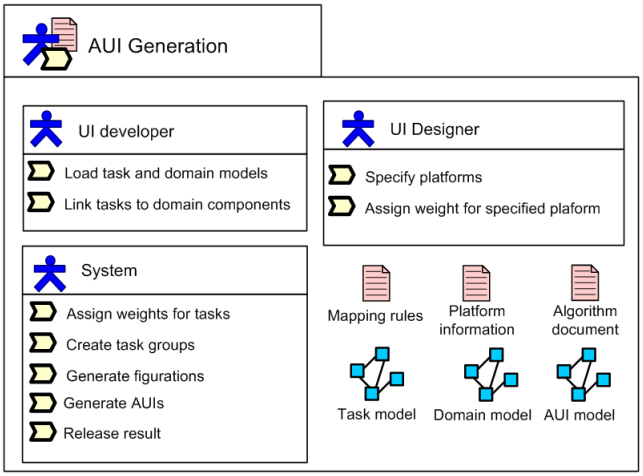

Figure 3: AUI generation and its principal components.

Workflow

Our process for generating the abstract user interface is depicted in Figure 4. This process stars with loading the task and domain models and finishes with returning the AUI model stored in terms of UsiXML specifications. The activities of the AUIs generation are depicted in order of their performance. These activities are described in Table 1. Table 2 describes the work-products used in our process.

Process at the Detailed Level

In this section, we discuss the engineering process and its main steps for the generation of AUIs from task and domain models. There are five steps namely: link tasks to domain components, assign weights for the tasks, create task groups, specify configuration and generate AUIs. In order to demonstrate the steps in this process, we use *Contact* task of eHealth application. *Contact* task describes how the information of a contact is displayed to the user and how the user can add a new contact and modify an existing one. This task has three sub-tasks: *View a contact, Search contact,* and *change phone number* (Figure 5).

Step 1: Link tasks to domain components and relation between the task and domain models

As discussed above, AUIs are generated based on the task and domain models. These models are considered for the following reasons:

- *The task model constraint the AUI.* It expresses how the UI provides information to the user and how the actions that users and system perform can be sequenced. This expression is intended to be independent of any particular implementation or technology. This explains why a same model or set of models could initiate several different UIs, whether they are abstract or concrete. However, the amount of possible AUIs that could be generated from a same task model depending on its configuration is not infinite.
- *The domain model provides the special features needed for creating a user interface*. These features are the attributes of the objects in the domain model, the relationships between these objects and prototype of the generic application functions. The domain model is used to specify the control of this user interface – at this level the user interface is specified more in details.

Two these models are related to each other. The relationship between the task and domain models can be described like the connections between the domain components and the tasks themselves that enable the user to perform operations on the domain objects. These operations are creating, deleting, modifying or selecting the objects in domain model. The UI is specified by selecting the elements of a domain model for the relevant tasks [19]. Before generating different abstract user interfaces for the different platforms, the tasks in task model will be linked to the components in domain model by the developer. The leaf tasks are linked to the components in domain model; these components maybe attributes, classes, and operations (Figure 6).

Task type	Weight	Description
1	0	Unknown task
2	1	Action task
3	2	Application task
4	3	Interaction task

Table 3: Weight of tasks

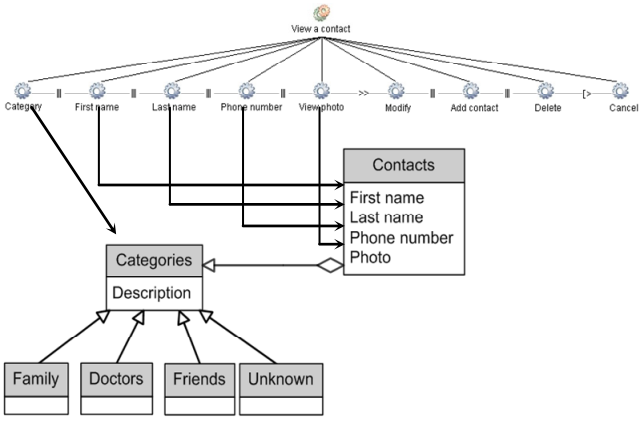

Figure 6: An example of linking tasks to domain components.

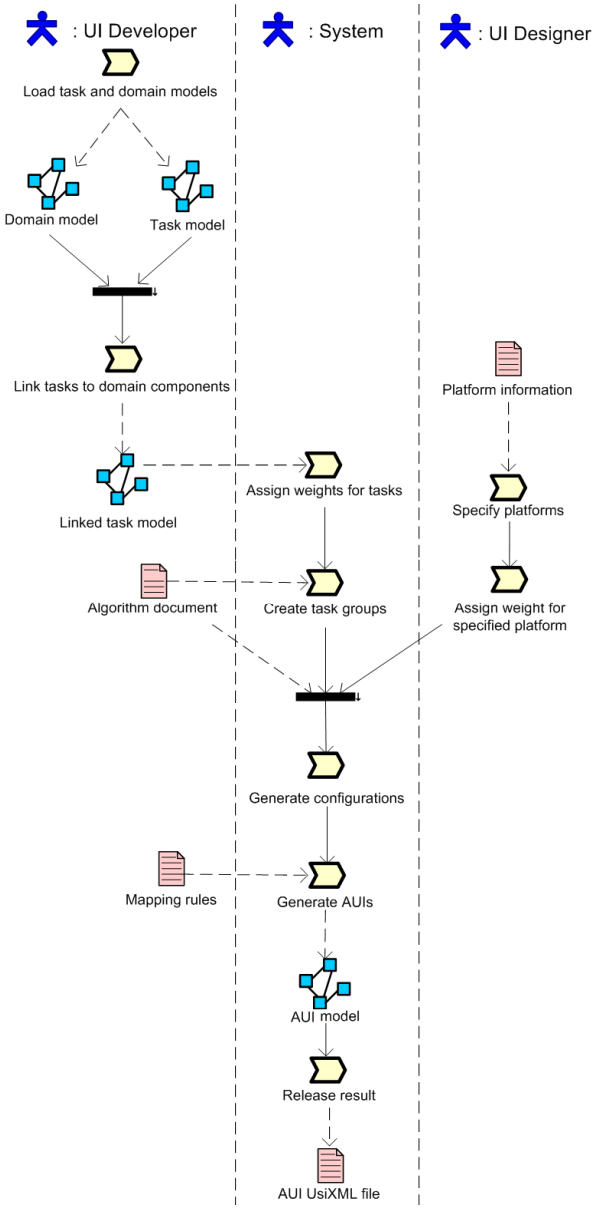

Figure 4: AUI generation workflow.

Table 1: Activities description.

No	Activity name	Goal	Process role	Input	Output
1	Load task and domain models	The goal of this activity is to load task model and domain model from XML files. These files are created by UsiXML tool.	UI Developer	XML files	Task and domain models
2	Link tasks to domain components	Once task and domain models have been loaded, the tasks in task model are linked to the components in the domain model by the UI developer.	UI Developer	Task and domain models	Linked task model
3	Specify platforms	The platform on which generated user interfaces will be run is specified based on a text document. The aim of this activity is to provide the platform information such as screen size, type, …	UI Designer	Text document	
4	Assign weight for specified platform	The different platforms are assigned different weights by the UI designer.	UI Designer		
5	Assign weights for tasks	The tasks in task model will be assigned weights based on the task types. For example the weight of an action task is 1 and the one of an application task is 2.	System		
6	Create task groups	Tasks will be grouped together to create all possible combinations.	System	Linked task model	Task groups
7	Generate configurations	Once the tasks have been grouped and the platform specified, the system generates configurations suitable to this platform by selecting task groups created during the previous activity.	System		
8	Generate AUIs	AUIs are generated automatically based on the configurations and the mapping rules.	System		AUIs model
9	Release result	The AUI is stored in terms of UsiXML specifications.	System		XML files

Table 2: Work-products description.

Work-product name	Type	Description
Task model	UsiXML model	The task model is a model structured in Figure 1. This model is used to describe the user's tasks.
Domain model	UsiXML model	The domain model is a model structured in Figure 2. This model is used to describe data objects and the associations between these objects
AUI model	UsiXML model	The AUI model describes the AUIs defined by UsiXML.
Mapping rules	Text document	The mapping rule list describes the rules that are used for specifying the AUI's types.
Platform information	Text document	The platform information describes the platform's parameters.
Algorithm document	Text document	The algorithm document describes all algorithms used in AUI generation process.

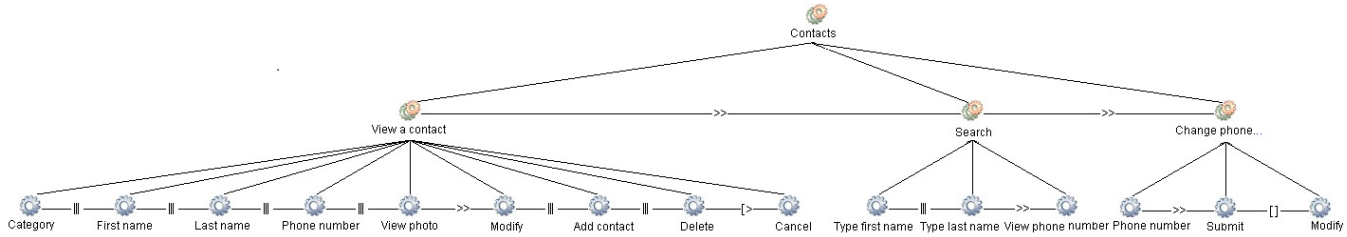

Figure 5: Contacts task model.

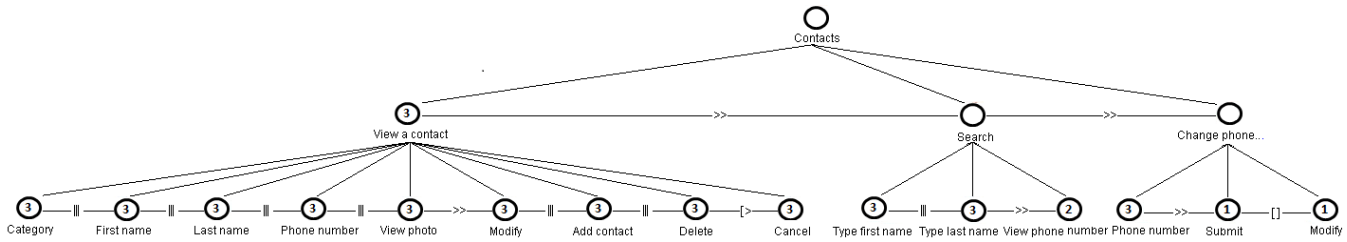

Figure 7: Tasks in task tree are assigned weights based on the task types (See Table 3).

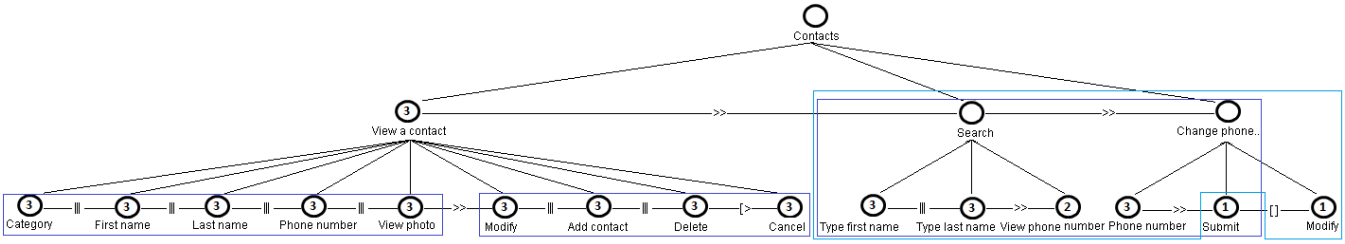

Figure 10: The configuration is based on the task weight and device weight.

The algorithm for specifying task weights is depicted as follows:

```
FOR each task of task model
   IF task type is action task THEN
      SET task weight to 1
   ELSE IF task type is application task THEN
      SET task weight to 2
   ELSE IF task type is interaction task THEN
      SET task weight to 3
   ELSE
      SET task weight to 0
   END IF
END FOR
```

Figure 7 depicts the result of running the algorithm with the task tree from Figure 6.

Step 3: Create task groups

This phase is decomposed in two sub-steps. The first one tries to find all of the possible groups of tasks without examining operators between these tasks (Figure 8). The second one will reject the unsuitable task groups from the ones created in the first sub-step (Figure 9). These are the ones that contain at least two adjacent tasks that the operator placed between these tasks is not suitable to the original task mode.

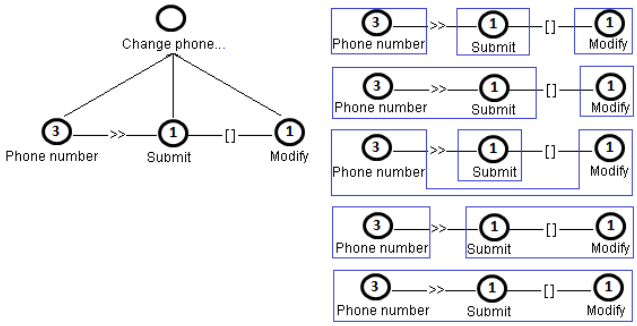

Figure 8: An example of task grouping.

The algorithm for creating task groups is depicted as follows:

```
RESET Vector taskGroups
INITIALIZE numOfTasks
FOR N = 1 to numOfTasks
   CALL Create_Group_N_Tasks(N)
END FOR
CREATE FUNCTION Create_Group_N_Tasks(number N)
   INITIALIZE taskGroup[N]
   CALL Combination(0 ,0 ,N)
END FUNCTION
CREATE FUNCTION Combination(number startIndex,
number currentIndex, number numOfTasksInGroup )
   FOR j = startIndex TO numOfTasks - numOf-
TasksInGroup + currentIndex - 1
      SET taskGroup[currentIndex] as taskList[j]
      IF currentIndex is equal to N - 1
         taskGroups.Add(taskGroup)
      ELSE
         Combination(j+1, currentIndex + 1, N)
      END IF
   END FOR
END FUNCTION
```

The main part of the source code in Java could be implemented as follows;

```
private void Combination(int startIndex, int
              currentIndex, int numOf-
              TasksInGroup){
   for(int j = startIndex; j <= numOfTasks -
        numOfTasksInGroup + currentIndex; j++){
      taskGroup[currentIndex] = taskList[j];
      if(currentIndex == numOfTasksInGroup - 1)
         taskGroups.Add(taskGroup)
      else
         Combination (j + 1, currentIndex + 1);
   }
}
```

One example for combination function is:

```
Vector taskGroups = new Vector ();
int numOfTasks = 3;
int[] taskList = {1, 2, 3};
private void Create_Group_N_Tasks (int N){
   int[]taskGroup = new int[N];
   Combination(0, 0, N);
}
for(int N = 0; N <=; N++)
   Create_Group_N_Tasks(N);
```

The result of this program is a combination of tasks 1, 2 and 3: {1, 2, 3, 12, 13, 23, 123}

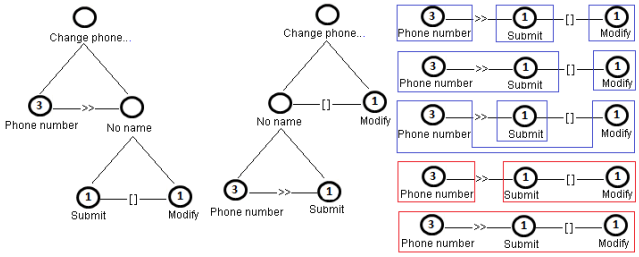

Figure 9: An example of operator check.

In order to generate valid sequences of tasks, we have formalized the definition of the task model using the Haskell programming language; which is based on lambda-calculus [6]. Lambda-calculus formalizes the function definition, application and recursion employing *Beta reduction*. Thus, we have defined a TaskExpression (TaskExp) as:

```
data TaskExp a = At a
   | En (TaskExp a) (TaskExp  a)
   | Ch (TaskExp a) (TaskExp  a)
   | Di (TaskExp a) (TaskExp  a)
   | Co (TaskExp a) (TaskExp  a)
   | Oi (TaskExp a) (TaskExp  a)
   | Su (TaskExp a) (TaskExp  a)
   | C (TaskExp  a) a
```

where a task expression is defined as an atomic task (At), a compose task (C) or any temporal relationship between temporal operations between task expressions: Enabling (En), Choice (Ch), Disabling (Di), Concurrency (Co), Order independence (Oi) and Suspend-Resume (Su). Thus, the "Change phone…" task can be expressed as:

```
C (En (At "Phone Number") (Ch (At "Submit") (At
                "Modify"))) "Change Phone"
```

To find out the valid sequences (traces) of tasks for this expression we employ the "trace" function. The following code represents part of its definition:

```
trace :: TaskExp a -> [[a]]
trace (At a)    = [[a]]
trace (C x a)   = trace x
trace (Ch x y) = trace x ++  trace y
trace (En x y) = [a ++ b |
                  a<-(trace x), b <- (trace y)]
```

```
trace (Co x y) = foldr (++) []
                 [interleaving xs ys |
                 xs <- trace x, ys <- trace y]
trace (Oi x y) =
   flatten (map bt (perms (lot (Oi x y))))
```

The "trace" function takes a task expression as input parameter, and returns a list of lists of atomic task names. Each list of atomic task names represents a valid trace of tasks. The definition of the trace function is trivial for the following expressions: At, C, Ch and En. However, the rest of them require auxiliary functions, such as, "perms", "between", "interleaving", "lot", "bt". The "perms" function returns the permutation of a list of atomic task names as a list of atomic task name lists. It is defined as follows:

```
perms [] = [ [] ]
perms (x:xs) = concat (map (between x) (perms
               xs))
```

To perform the permutation, the "between" function is defined to generate all traces resulting from the insertion of an atomic task at any position of an already defined trace.

```
between e [] = [ [e] ]
between e (y:ys) = (e:y:ys) : map (y:) (between
                e ys)
```

The "interleaving" function is linked to the concurrency operation. It takes two traces as input parameters to return a list of traces representing the concurrent execution of both traces.

```
interleaving :: [a]->[a]->[[a]]
interleaving (x:xs) [] = [x:xs]
interleaving [] (y:ys) = [y:ys]
interleaving (x:xs) (y:ys) = map (y:) (inter-
                leaving (x:xs) (ys)) ++ map (x:)
                (interleaving (xs) (y:ys))
```

The "lot" function takes a task expression, and returns the list of list of traces that represent the "composed traces" derived from the task expression passed as parameter.

```
lot:: TaskExp a -> [[[a]]]
lot (Oi x y) = lot x ++ lot y
lot t = trace t :[]
```

To generate the trace resulting from the order independence temporal operator, we apply the "bt" function on each of the list of traces generated from the result of applying the "lot" function. The "bt" function is defined as follows:

```
bt::[[[a]]]->[[a]]
bt (xss:[]) = xss
bt (xss:xsss) = [a ++ b | a<-xss, b<-(bt xsss)]
```

The result of applying the trace function to the "Change Phone…" task is: `[["Phone Number","Submit"], ["Phone Number","Modify"]];` which reveals the valid traces only for the temporal expression. Note that "[" and "]" denote a list and "," denote sequence in this case.

Step 4: Configuration is specified automatically.

Once the task groups have been created, the system automatically specifies the configurations by selecting one or more task groups. The system generates the different configurations for the different platforms based on characteristics such as screen size, processors, memory of devices. In order to do that, each platform will be assigned a maximum weight by the UI designer. Usually, this maximum weight is in direct ratio to the screen's size, the power of processor and memory, screen type ... The maximum weight is used to specify the number of task groups which are suitable to determined platform. The formulas for selecting task groups based on weight of task group and maximum weight are as follows:

```
Weight of a task group = ∑ Weight of its tasks
Maximum weight >= ∑ Weight of selected task
groups
```

The algorithm for selecting task groups based on the task weight and device weight is defined as follows:

```
SET current weight to 0
SET maximum weight to 15
SET current group to null
WHILE current weight is less than maximum weight
      AND group count is more than 0
   FOR each group in taskGroups
      IF current weight + group weight is less
         than or equal maximum weight AND group
         weight is less than current group
         weight
         SET current group weight to group
weight
         STORE current group as group
      END IF
   END FOR
   IF current group is NOT NULL
      COMPUTE current weight as current weight +
            group weight
      FOR each task in current group
         FOR each group in taskGroups
            IF group contains task
               REMOVE this group from taskGroups
            END IF
         END FOR
      END FOR
   ELSE
      BREAK WHILE
   END IF
END WHILE
```

Figure 10 depicts the configuration of *contacts* for a task model with a device weight of 15. In this configuration, two choices for creating the container are possible: the first one that contains: Type first name, Type last name, View phone number, Phone number and Submit; the other one that contains: Type first name, Type last name, View phone number, Phone number, and Modify

Step 5: Generating abstract user interface from task and domain models

AUIs are generated based on the mapping rules and the task groups specified manually by the developer or automatically by the system. For each created group, the system generates an AbstractCompoundIU; this AbstractCompoundIU will contain all of AUIs generated for the tasks belonging to this group. The rules for determining the AUI type are:

- **Rule1:** An AbstractSelectionUI is considered when a task derives from an attribute of a domain class which is not the edited class and the relationships between the edited class and another one is '1-1' or 'n-1'and .
- **Rule2:** An AbstractInputUI is considered when a task derives from the attributes of the classes that these classes are the edited classes.

- **Rule3:** An AbstractOutputUI is considered when an abstract user interaction has been created and its label is the task name of the task related to this abstract user interaction.
- **Rule4:** An AbstractDataItemUI is considered when a task derives from the attributes of the classes.
- **Rule5:** An AbstractTriggerUI is considered when a task derives from an operation of a class. For example. Once the tasks have been grouped by the developer based on the screen size of devices, the AUIs are generated automatically by system.

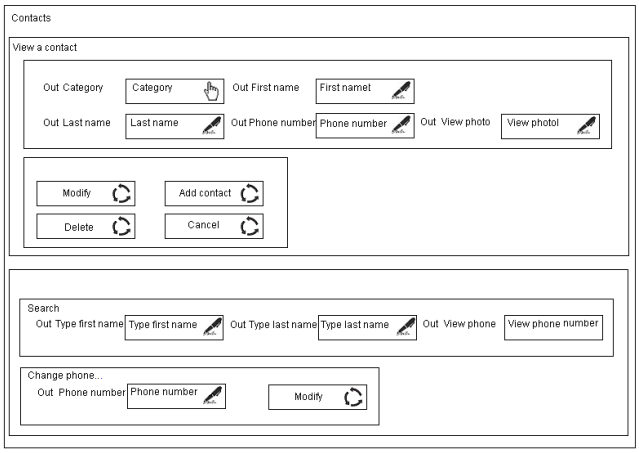

Figure 11: AUI is generated from configuration specified in Figure 10.

One of the AUI specifications generated from the configuration above is shown in Figure 11. The abstract user interface units are specified based on the following algorithm:

```
FOR each task in task list
   IF task has sub tasks
      CREATE an AbstractCompoundIU
   ELSE
      IF task type is action task OR task is
         linked to operation of class
         CREATE an AbstractTriggerIU
      ELSE IF task type is application task
         CREATE an AbstractOutputIU
      ELSE IF task type is interaction task
         IF task is linked to attributes of
class
            CREATE an AbstractInputIU
         ELSE IF task is linked to class
            CREATE an AbstractSelectionIU
         ELSE
            CREATE an AbstractInputIU
         END IF
      ELSE
         CREATE an AbstractInputIU
      END IF
   END IF
END FOR
```

SOFTWARE SUPPORT

Integrated software

The software developed to support the aforementioned process and that implements the algorithms outlined in the discussion all at once has been implemented in Java. The main purpose of the tool is to help designers to generate AUIs from the task and domain models.

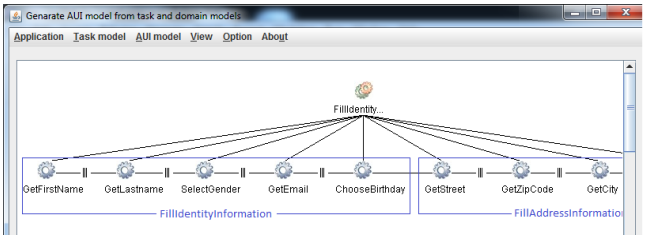

Figure 12: Task model editor.

The task model is loaded from a UsiXML file (Figure 12). Once tasks contained in the task model have been loaded, they are collected into the different groups by the designer depending on the concrete platform. The number of tasks in our task model is unlimited. There are two ways to group the tasks; the first one is that the tasks are grouped manually by the designer; and the second one is that they are grouped automatically based on the device selected by designer (Figure 13). With this tool, the designer can manipulate tasks easily with the mouse buttons and Ctrl key. In order to observe task model easily and clearly, the designer can decide which task attributes to display in the task model by using *Task model view* dialog (Figure 14).

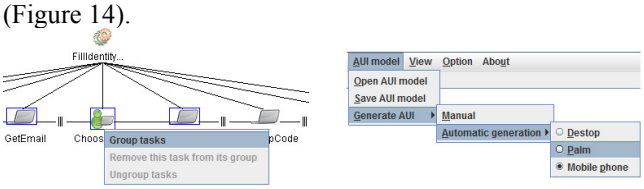

Figure 13: Grouping tasks.

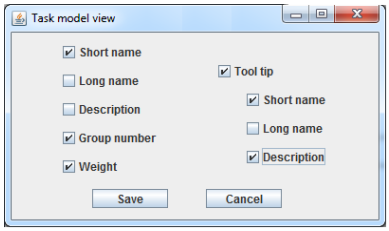

Figure 14: Selecting task attributes

The number of tasks in our task model is unlimited. In this current version, the tasks are manually linked to the components of domain model by the designer. Abstract user interfaces are automatically generated based on grouped tasks and the attributes of domain's components. Generated AUIs are stored in terms of UsiXML specification. In this version, the designer cannot directly modify automatically generated AUIs in order to preserve the rules that have been fired to obtain these results. Indeed, it the resulting AUIs are modified manually, they will no longer be consistent with the rules that were used for this generation. If the designer wants to change the AUIs, she has to modify the task group or the relation between tasks and domain's components. Our generated AUIs are depicted in Figure 15.

AUI Generation as service

The systematic generation of AUIs from a task model could be also invoked as a service from any other UsiXML compatible software.

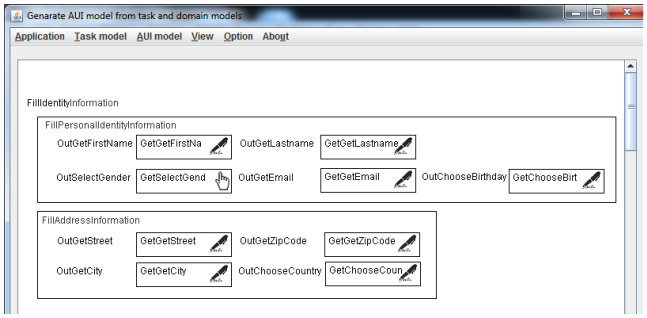

Figure 15: An example of AUI generated by our software.

For instance, Figure 16 reproduces an Eclipse-based task model editor specifying tasks according to the UsiXML V2.1 meta-model outlined in Figure 1. In this example, a simple car rental task model is depicted, with more detailed information about the sub-task FillIdentityInformation. Figure 17 details the first AUI candidates for this task by decreasing order of amount of interaction units contained and complexity, while Figure 18 shows some rendering in the integrated software.

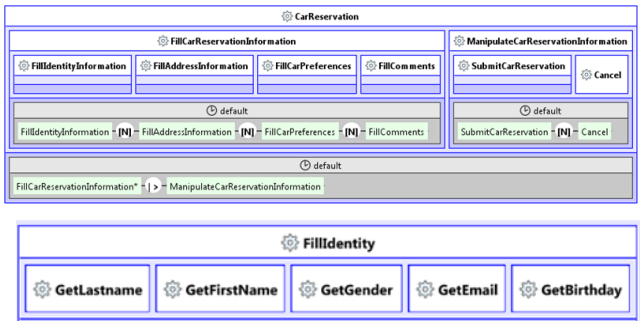

Figure 15: Task model in Eclipse-based task model editor.

Figure 16: AUI candidates from the same task model.

CONCLUSION

In this paper, we have presented an algorithm for systematically generating all potential abstract user interfaces from a task model. Our AUI generation process has been discussed at the comprehensive and detailed levels. At the comprehensive level, we have discussed its main tasks and the resources used in the process. At the detailed level, the process is represented step by step with the rules for specifying the abstract user interaction types and the algorithms used in each step. More specially, this paper has provided a number of necessary algorithms for an AUI generation process. In order to explore these algorithms, an editor tool has been implemented to evaluate the cost and performance of this method.

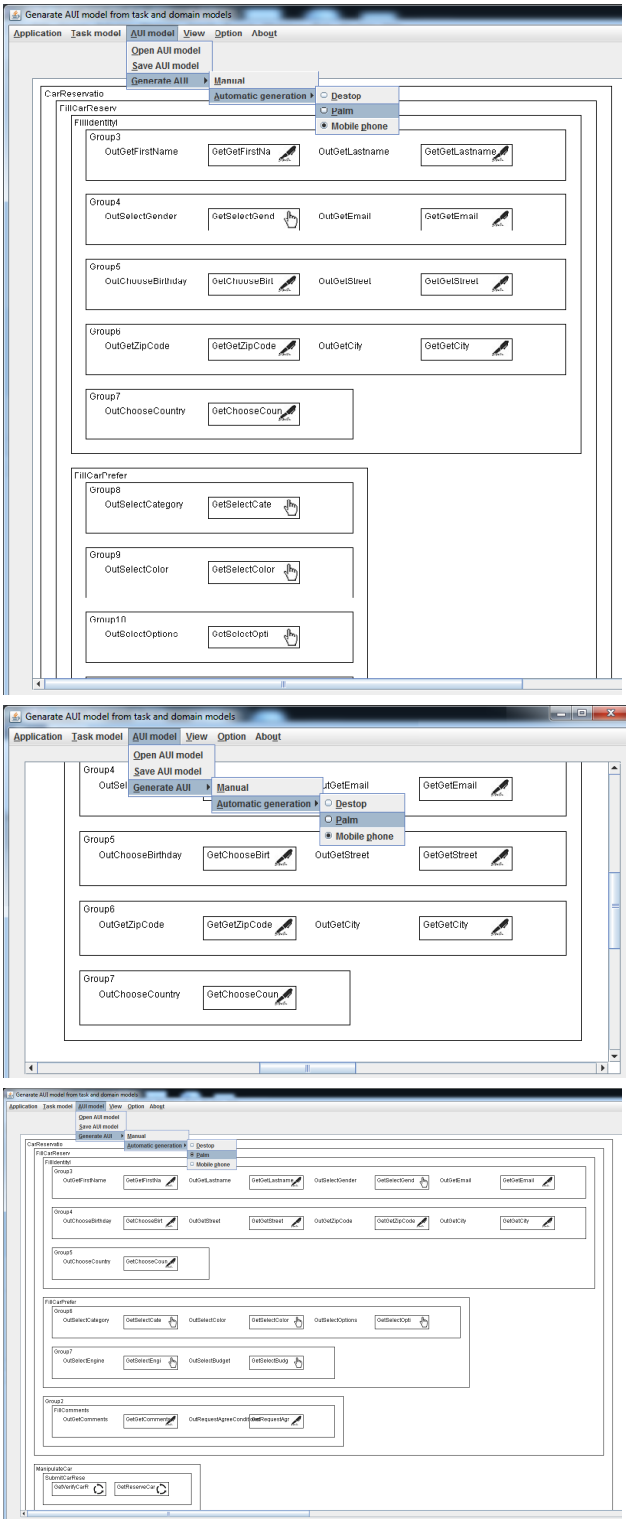

Figure 17: AUI rendered in the integrated software.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the FEDER eHealth project and the ITEA2-Call3-2008026 USIXML European project (User Interface eXtensible markup language – http://www.usixml.eu, http://www.usixml.org) and its support by Région Wallonne DGO6.

REFERENCES

1. Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., and Vanderdonckt, J. Computer-aided window identification in TRIDENT, in *Proc. of IFIP TC13 Int. Conf. on Human-Computer Interaction Interact'95* (Lillehammer, June 27-29, 1995). Chapman & Hall, London, 1995, 331–336.
2. Bogdan, C., Falb, J., Kaindl, H., Kavaldjian, S., Popp, R., Horacek, H., Arnautovic, E., and Szep, A. Generating an Abstract User Interface from a Discourse Model Inspired by Human Communication, in *Proc. of the 41st Annual Hawaii Int. Conf. on System Sciences HICSS'2008*. IEEE Computer Society, Los Alamitos, 2008, 1–10.
3. 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.
4. Cantera Fonseca, J.M., González Calleros, J.M., Meixner, G., Paternò, F., Pullmann, J., Raggett, D., Schwabe, D., and Vanderdonckt, J. Model-Based User Interface Incubator Group, Final Report. 4 May 2010. Available at http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui/.
5. Chu, H., Song, H., Wong, C., Kurakake, S., and Katagiri, M. Roam, a seamless application framework. *J. of Systems and Software 69*, 3 (2004), 209–226.
6. Church, A. A set of postulates for the foundation of logic. *Annals of Mathematics 39*, 2 (1932), 346-366.
7. Falb, J., Popp, R., Röck, T., Jelinek, H., Arnautovic, E. and Kaindl, H. Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts, in *Proc. of the 40th Annual Hawaii Int. Conf. on System Sciences HICSS-40* (Waikoloa, 3-6 January 2007). IEEE Computer Society, Los Alamitos, 2007, Track 26.
8. Furtado, E., Furtado, V., Sousa, K., Vanderdonckt, J., and Limbourg, Q. KnowiXML: A Knowledge-Based System Generating Multiple Abstract User Interfaces in UsiXML, in *Proc. of 3rd Int. Workshop on Task Models and Diagrams for User Interface Design-Tamodia'2004* (Prague, November 15-16, 2004). ACM Press, New York, 2004, 121–128.
9. Gajos, K.Z., Weld, D.S., and Wobbrock, J.O. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence 174*, 12-13 (2010), 910–950.
10. González Calleros, J.M., Stanciulescu, A., Vanderdonckt, J., Delacre, J.P., and Winckler, M. A Comparative Analysis of Transformation Engines for User Interface Development, in *Proc. of 4th Int. Workshop on Model-Driven Web Engineering MDWE'2008* (Toulouse, 1 October 2008). N. Koch, G.-J. Houben, A. Vallecillo (Eds.), CEUR Workshop Proceedings, Vol. 389, 2008, 16–30.
11. Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction, in *Proc. of 10th Int. Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2003* (Funchal, June 11-13, 2003). Lecture Notes in Computer Science, Vol. 2844. Springer, Berlin, 2003, 203–217.
12. Martínez-Ruiz, F. J., Vanderdonckt, J., and Arteaga, J. M. Web User Interface Generation for Multiple Platforms, in *Proc. of 7th Int. Workshop on Web-Oriented Software Technologies IWWOST'2008* (Yorktown Heights, July 14, 2008). L. Olsina, O. Pastor, D. Schwabe, G. Rossi, M. Winckler (Eds.), CEUR Workshop Proceedings, Vol. 445, 2008, 63–68.
13. Paternò, F. and Zichittella, G. Desktop-to-Mobile Web Adaptation through Customizable Two-Dimensional Semantic Redesign, in *Proc. of 3rd Int. IFIP Conf. on Human-Centred Software Engineering HCSE'2010* (Reykjavik, October 14-15, 2010). Lecture Notes in Computer Science, Vol. 6409. Springer, Berlin, 2010, 79–94.
14. Perico, E., TopCased User Interface generator, Atos origin, 4 May 2005. http://www.topcased.org/index.php?id_projet_pere=114
15. Plomp, C. and Mayora-Ibarra, O. A generic widget vocabulary for the generation of graphical and speech-driven user interfaces. *International Journal of Speech Technology 5* (2002) 39–47.
16. Pribeanu, C. An Approach to Task Modeling for User Interface Design, in *Proc. of World Enformatika Conf. WEC'2005* (Istanbul, April 27-29, 2005). C. Ardil (Ed.). *Enformatika 5*, 2005. 5–8.
17. Schneider, K.A. and Cordy, J. Abstract User interfaces: A model and notation to support plasticity in interactive systems, in *Proc. of the 8th Int. Workshop of Design, Specification and Verification of Interactive Systems DSV-IS'2001*. Springer, Berlin, 2001, 40–58.
18. Van den Bergh, J., Luyten, K., and Coninx, K. CAP3: Context-Sensitive Abstract User Interface Specification, in *Proc. of ACM Symposium on Engineering Interactive Systems EICS'2011* (Pisa, June 13-16, 2011). ACM Press, New York, 2011, 31–40.
19. Vanderdonckt, J., Tesoriero, R., Beuvens, F., and Melchior, J. Towards a Fifth-Generation User Interface Description Language with UsiXML V2.1. Submitted to *Science of Computer Programming*, June 2012.
20. Vanderdonckt, J. Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures, in *Proc. of 5th Annual Romanian Conf. on Human-Computer Interaction ROCHI'2008* (Iasi, September 18-19, 2008). Matrix ROM, Bucharest, 2008, 1–10.