

Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation

Arnaud Blouin
IRISA, Triskell, Rennes
arnaud.blouin@inria.fr

Grégory Nain
INRIA, Triskell, Rennes
gregory.nain@inria.fr

Brice Morin
SINTEF ICT, Oslo
brice.morin@sintef.no

Patrick Albers
ESEO-GRI, Angers
patrick.albers@eseo.fr

Olivier Beaudoux
ESEO-GRI, Angers
olivier.beaudoux@eseo.fr

Jean-Marc Jézéquel
IRISA, Triskell, Rennes
jezequel@irisa.fr

ABSTRACT

User interface adaptations can be performed at runtime to dynamically reflect any change of context. Complex user interfaces and contexts can lead to the combinatorial explosion of the number of possible adaptations. Thus, dynamic adaptations come across the issue of adapting user interfaces in a reasonable time-slot with limited resources. In this paper, we propose to combine aspect-oriented modeling with property-based reasoning to tame complex and dynamic user interfaces. At runtime and in a limited time-slot, this combination enables efficient reasoning on the current context and on the available user interface components to provide a well suited adaptation. The proposed approach has been evaluated through EnTiMid, a middleware for home automation.

Author Keywords

MDE, user interface, context, adaptation, aspect, runtime, malai

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces—*Theory and methods, User Interface Management Systems (UIMS)*; D.2.1 Software Engineering: Requirements/ Specifications—*Methodologies*; H.1.0 Information Systems: Models and Principles—*General*

General Terms

Design

INTRODUCTION

The number of platforms having various interaction modalities (*e.g.*, netbook and smart phone) unceasingly

increases over the last decade. Besides, user's preferences, characteristics and environment have to be considered by user interfaces (UI). This triplet $\langle \text{platform, user, environment} \rangle$, called context¹ [8], leads user interfaces to be dynamically (*i.e.* at runtime) adaptable to reflect any change of context.

UI components such as tasks and interactions, enabled for a given context but disabled for another one, cause a wide number of possible adaptations. For example, [14] describes an airport crisis management system that leads to 1,474,560 possible adaptations. Thus, an important challenge is to support UI adaptation of complex systems. This implies that dynamic adaptations must be performed in a minimal time, and respecting usability.

The contribution of this paper is to propose an approach that combines aspect-oriented modeling (AOM) with property-based reasoning to tackle the combinatorial explosion of UI adaptations. AOM approaches provide advanced mechanisms for encapsulating cross-cutting features and for composing them to form models [1]. AOM has been successfully applied for the dynamic adaptation of systems [20]. Property-based reasoning consists in tagging objects that compose the system with characterizing properties [14]. At runtime, these properties are used by a reasoner to perform the adaptation the best suited to the current context. Reasoning on a limited number of aspects combined with the use of properties avoids the combinatorial explosion issue. Although these works tackle system adaptation at runtime, they do not focus on the dynamic adaptation of UIs. Thus, we mixed these works with Malai, a modular architecture for interactive systems [4], to bring complex and dynamic user interface adaptations under control. We have applied our approach to EnTiMid, a middleware for home automation.

The paper is organized as follows. The next section introduces background research works used by our ap-

¹In this paper, the term "context" is used instead of "context of use" for conciseness.

proach. Then, the process to create an adaptive UI using our approach is explained. Next, the adaptation process that is automatically executed at runtime is detailed. Following, our approach is evaluated through EnTiMiD, a middleware for house automation. The paper ends with the related work and the conclusion.

BACKGROUND

The work presented in this paper brings an interactive system architecture and a software engineering approach together. Thus, this section starts with the presentation of the Malai architecture. The software engineering approach applied to Malai to allow complex UI adaptations at runtime is then introduced.

The Malai Architecture

The work presented in this paper is based on Malai, an architectural model for interactive systems [4]. In Malai a UI is composed of presentations and instruments (see Figure 1). A presentation is composed of an abstract presentation and a concrete presentation. An abstract presentation is a representation of source data created by a Malan mapping (link ①). A concrete presentation is the graphical representation of the abstract presentation. It is created and updated by another Malan mapping (link ②) [5]. An interaction consumes events produced by input devices (link ③). Instruments transform input interactions into output actions (link ④). An action is executed on the abstract presentation (link ⑤); source data and the concrete presentation are then updated throughout a Malan mapping (link ⑥).

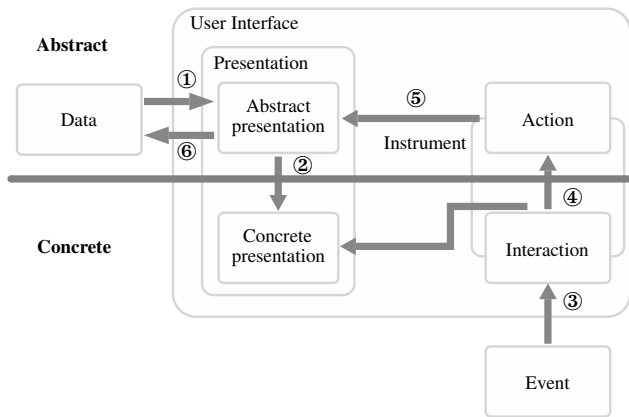


Figure 1. Organization of the architectural model Malai

Malai aims at improving: 1) the modularity by considering presentations, instruments, interactions, and actions, as reusable first-class objects; 2) the usability by being able to specify feedback provided to users within instruments, to abort interactions, and to undo/redo actions. Malai is well-suited for UI adaptation because of its modularity: depending on the context, interactions, instruments, and presentations, can be easily composed to form an adapted UI. However, Malai does not provide any runtime adaptation process. The next section

introduces the research work on dynamic adaptive system that has been applied to Malai for this purpose.

Dynamically Adaptive Systems

The DiVA consortium proposes an adaptation meta-model to describe and drive the adaptation logic of Dynamically Adaptive Systems (DAS) [14]. The core idea is to design DAS by focusing on the commonalities and variabilities of the system instead of analyzing on all the possible configurations of the system. The features of the system are refined into independent fragments called *aspect models*. On each context change, the aspect models well adapted to the new context are selected and woven together to form a new model of the system. This model is finally compared to the current model of the system and a safe migration is computed to adapt the running system [20].

The selection of the features adapted to the current context is performed by a reasoning mechanism based on multi-objective optimization using *QoS Properties*. QoS properties correspond to objectives that the reasoner must optimize. For example, the properties of the system described in [14] are security, CPU consumption, cost, performances, and disturbance. The importance of each property is balanced depending on the context. For instance, if the system is running on battery, minimizing CPU consumption will be more important than maximizing performances. The developer can specify the impact of the system's features on each property. For example, the video surveillance feature highly maximizes security but does not minimize CPU consuming. The reasoner analyzes these impacts to select features the best suited to the current context.

If DiVA proposes an approach that tames dynamic adaptations of complex systems, it lacks at considering UI adaptations. The following sections describe the combined use of Malai and DiVA to bring adaptations of complex interactive systems under control.

CONCEPTION PROCESS

This section describes the different steps that developers have to perform during the conception of adaptable UIs. The first step consists in defining the *context* and the *action* models. Then a *mapping* between these two models can be defined to specify which context elements disable actions. The last step consists in defining the *presentations* and the *instruments* that can be selected at runtime to compose the UI.

All these models are defined using Kermeta. Kermeta is a model-oriented language that allows developers to define both the structure and the behavior of models [21]. Kermeta is thus dedicated to the definition of executable models.

Context Definition

A context model is composed of the three class models *User*, *Platform*, and *Environment* that describe each

context component. Developers can thus define their own context triples without being limited to a specific context metamodel.

Each class of a class model can be tagged with QoS properties. These properties bring information about objectives that demand top, medium, or low priority during UIs adaptation. For instance, Listing 1 defines an excerpt of the user class model for a home automation system. Properties are defined as annotations on the targeted class. This class model specifies that a user can be an elderly person (line 3) or a nurse (line 6). Class *ElderlyPerson* is tagged with two properties. Property *readability* (line 1) concerns the simplicity of reading of UIs. Its value *high* states that the readability of UIs must be strongly considered during adaptations to elderly people. For instance, large buttons would be more convenient for elderly people than small ones. Property *simplicity* (line 2) specifies the simplicity of the UI. Since elderly people usually prefer simple interaction, this property is set to *high* on class *ElderlyPerson*.

```

1 @readability "high"
2 @simplicity "high"
3 class ElderlyPerson inherits User {
4 }
5
6 class Nurse inherits User {
7 }

```

Listing 1. Context excerpt tagged with QoS properties

By default properties are set to "low". For example with Listing 1, property *readability* is defined on class *ElderlyPerson* but not on class *Nurse*. It means that by default *Nurse* has property *readability* set to "low".

All the properties of the current context should be maximized. But adapting UIs is a multiobjective problem where all objectives (*i.e.* QoS properties) cannot be maximized together; a compromise must be found. For example, a developer may prefer productivity to the aesthetic quality of UIs even if maximizing both would be better. Values associated with properties aim at balancing these objectives.

Our approach does not provide predefined properties. Developers add their own properties on the UI components and the context. The unique constraint for the developers is to reuse in the context model properties defined in UI components and *vice versa*. Indeed, properties of the current context are gathered at runtime to then select the most respectfully UI components towards these properties. The efficiency of the reasoner thus depends on the appropriate definition of properties by the developers.

Actions Definition

Actions are objects created by instruments. Actions modify the source data or parameters of instruments. The main difference between actions and tasks, such

as CTT tasks [22], is that the Malai's action meta-model defines a life cycle composed of methods *do*, *canDo*, *undo*, and *redo*. These methods, that an action model must implement, bring executability to actions. Method *canDo* checks if the action can be executed. Methods *do*, *undo*, and *redo* respectively executes, cancels, and re-executes the action. An action is also associated to a class which defines the attributes of the action and relations with other actions.

```

1 abstract class NurseAction inherits Action { }
2
3 class AddNurseVisit inherits NurseAction, Undoable{
4   reference calendar : Calendar
5   attribute date      : Date
6   attribute title     : String
7   attribute event     : Event
8
9   method canDo() : Boolean is do
10    result := calendar.canAddEvent(date)
11  end
12  method do() : Void is do
13    event := calendar.addEvent(title, date)
14  end
15  method undo() : Void is do
16    calendar.removeEvent(title, date)
17  end
18  method redo() : Void is do
19    calendar.addEvent(event)
20  end
21 }
22
23 class CallEmergencyService inherits NurseAction{
24   // ...
25 }

```

Listing 2. Excerpt of nurse actions

Listings 2 defines an excerpt of the home automation action model in Kermeta. Abstract action *NurseAction* (line 1) defines the common part of actions that nurses can perform. Action *AddNurseVisit* (line 3) is a nurse action that adds an event into the nurse calendar (see method *do* line 12). Method *canDo* checks if the event can be added to the calendar (line 9). Methods *undo* and *redo* respectively remove and re-add the event to the calendar (lines 15 and 18). Action *CallEmergencyService* in another nurse action that calls the emergency service (line 23).

Mapping Context Model to Action Model

Actions can be disabled in certain contexts. For instance elderly people cannot perform actions specific to the nurse. Thus, action models must be constrained by context models. To do so we use Malan, a declarative mapping language [5]. Because it is used within the Malai architecture, the Malan language has been selected. Context-to-action models consists of a set of Malan expressions. For instance, one of the constraints of the home automation system states that elderly people cannot perform nurse actions. The Malan expression for this constraint is:

ElderlyPerson -> !NurseAction

where *NurseAction* means that all actions that inherit from action *NurseAction* are concerned by the mapping.

Another constraint states that nurses can call ambulances only if the house has a phone line. The corresponding Malan expression is:

```
House [! phoneLine] -> ! CallEmergencyService
```

where the expression between brackets (*i.e.*, *!phoneLine*) is a predicate that uses attributes and relations of the corresponding class of the context (*i.e.* *House* in the example) to refine the constraint.

By default all the actions are enabled. Only actions targeted by context-to-action mappings can be disabled: on each context change, mappings are re-evaluated to enable or disable their target action.

Presentation Definition

Developers can define several presentations for the same UI: several presentations can compose at runtime the same UI to provide users with different viewpoints on the manipulated data; defining several presentations allows to select at runtime the presentations the best suited to the current context. For instance, the calendar that the nurse uses to add visits can be presented through two presentations: 1) a 2D-based presentation that displays the events of the selected month or week; 2) a list-based presentation that shows the events into a list widget.

```
1 class Agenda {
2   attribute name      : String
3   attribute events    : Event[0..*]
4   attribute dates     : Date[0..*]
5 }
6 class Event {
7   attribute name      : String
8   attribute description : String
9   attribute place     : String
10  reference date      : Date
11  attribute start     : TimeSlot
12  attribute end       : TimeSlot
13 }
14 //...
```

Listing 3. Excerpt of the 2D-based abstract presentation

```
1 @aestheticQuality "high"
2 @space "low"
3 class AgendaUI {
4   attribute title      : String
5   attribute linesUI    : LineHourUI[0..*]
6   attribute handlerStart : Handler
7   attribute handlerEnd  : Handler
8   attribute eventsUI   : EventUI[0..*]
9   attribute datesUI    : DateUI[0..*]
10 }
11 class EventUI {
12   attribute x          : Real
13   attribute y          : Real
14   attribute width      : Real
15   attribute height     : Real
16 }
17 //...
```

Listing 4. Excerpt of the 2D-based concrete presentation and its QoS properties

Listings 3 and 4 describe parts of the 2D-based presentation of the nurse agenda. Its abstract presentation

defines the agenda model (see Listing 3). An *Agenda* has a name, contains *Event* and *Date* instances. An event has a name, a place, a description, a starting and an ending *Timeslot* instances. A time-slot specifies the hour and the minute. A date defines its day, month and year.

The concrete presentation defines the graphical representation of the nurse agenda (see Listing 4). The graphical representation of agendas (class *AgendaUI*) contains representations of days, events, and time-slot lines (respectively classes *DayUI*, *EventUI* and *LineHourUI*). These representations have coordinates x and y. Classes *DayUI* and *EventUI* also specify their width and height. An agenda has two handlers associated to the selected event. These handlers are used to change the time-slot of the selected event.

Similarly to context models, presentations can be tagged with QoS properties. These properties provide context reasoner with information about, for example, the easiness of use or the size of the presentation. For instance, the 2D-based and list-based presentations have characteristics well-suited for some platforms and users. Listing 4 shows the QoS properties of the 2D-based presentation defined as annotations: the 2D-based presentation optimizes the aesthetic quality (property *aestheticQuality "high"*) but not space (property *space "low"*). By contrast, the list-based presentation optimizes space to the detriment of the aesthetic quality.

While properties specified on contexts define objectives to optimize at runtime, properties on presentations declare characteristics used to select appropriated presentations depending on the current context and its objectives. For instance, if the current context states that the the aesthetic quality must be highly considered, the 2D-based presentation will be selected.

Instrument Definition

Instruments transform input interactions into output actions. Instruments are composed of links and of a class model. Each link maps an interaction to a resulting action. Instrument's class model defines attributes and relations the instrument needs. Widgets handled by instruments and that compose the UI are notably defined into the class model of instruments.

VisitTypeSelector is an instrument operating on the nurse agenda. This instrument defines the type of visit to add to the agenda. The selection of the type of visit can be performed using different widgets: several toggle buttons (one of each visit type) or a list can be used. While toggle buttons are simpler to use than a list (a single click to select a button against two clicks to select an item of a list), lists are usually smaller than a set of toggle buttons. The choice of using such or such widgets thus depends on the current context: if space is a prior objective, list should be privileged; otherwise, toggle buttons should be selected.

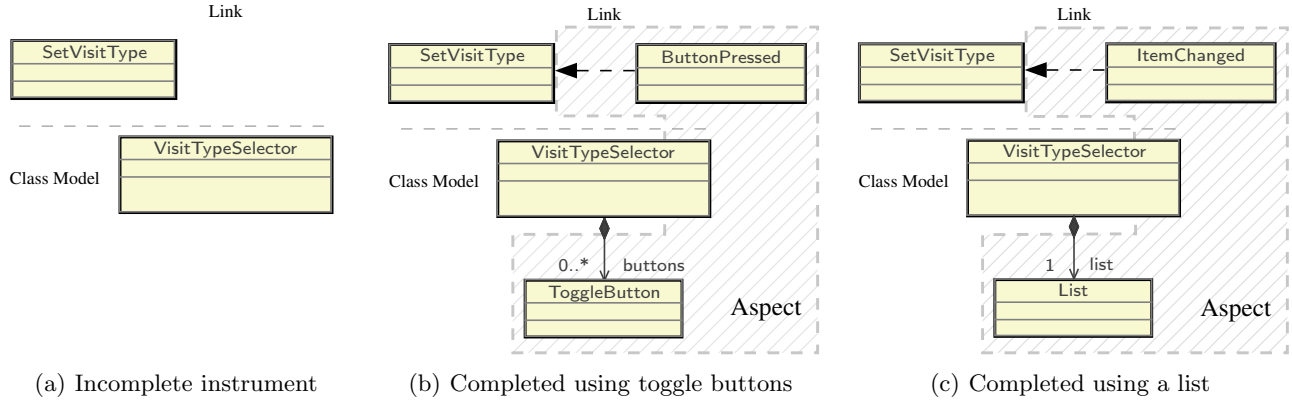


Figure 2. Instrument *VisitTypeSelector*

One of the contributions of our work consists of being able to choose the best suited interaction for a link at runtime: while defining instruments, developers can let interactions undefined. Interactions and widgets are automatically chosen and associated to instruments at runtime depending on the current context. For instance, Figure 2(a) describes the model of instrument *VisitTypeSelector* as defined by developers. This model is composed of an incomplete link that only specifies the produced action *SetVisitType*; the way this action is performed is let undefined. The class model of this instrument only defines a class corresponding to the instrument (class *VisitTypeSelector*). This class model will also be completed at runtime.

Figure 2(b) corresponds to the model of Figure 2(a) completed at runtime. Toggle buttons have been chosen to perform action *SetVisitType*. The interaction corresponding to the click on buttons (interaction *ButtonPressed*) is added to complete the link. A set of toggle buttons (class *ToggleButton*) is also added to the class model. This interaction and these widgets come from a predefined aspect encapsulating them. We defined a set of aspects for WIMP² interactions (*i.e.* based on widgets) that can automatically be used at runtime to complete instrument models.

Figure 2(c) corresponds to another completed model. This time, a list has been chosen. Interaction *ItemChanged*, dedicated to the handle of lists, completes the link. A list widget (class *List*) has been also added to the class model. This widget and its interaction also come from a predefined aspect.

Figure 3 presents an example of the instrument *TimeslotSetter* completed with interactions. This instrument changes the time-slot of events of the nurse agenda (action *SetTimeslotEvent*). Figure 3(a) shows this instrument completed with a drag-and-drop interaction (*DnD*) and handlers. Handlers surround the selected event. When users drag-and-drop one of these handlers the time-slot of the event is modified. This interaction

and these handlers were encapsulated into an aspect defined by the developer.

Figure 3(b) shows another aspect defined by the developer for instrument *TimeslotSetter*: when the current platform supports bi-manual interactions, such as smartphones or tabletops, time-slot setting can be performed using such interactions instead of using a DnD and handlers.

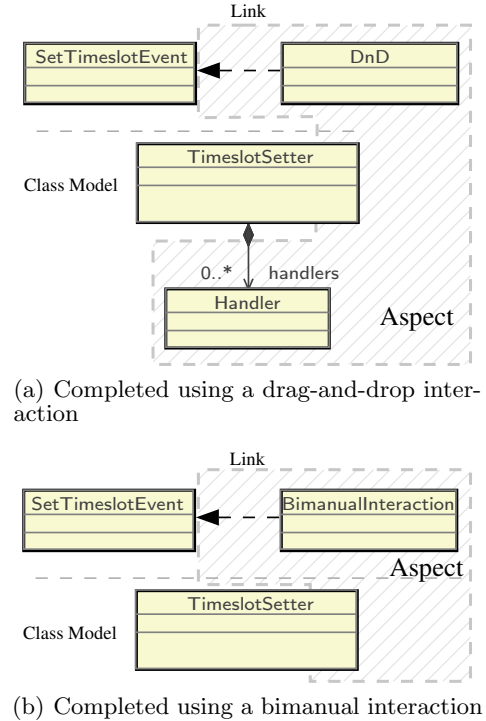


Figure 3. Instrument *TimeslotSetter*

Such flexibility on interactions and widgets is performed using QoS properties. Widgets and interactions are tagged with properties they maximize or minimize. Widgets are also tagged with properties corresponding to simple data type they are able to handle. For in-

²” Windows, Icons, Menus and Pointing device”

stance, the toggle button widget is tagged with four properties: property *simplicity high* means that toggle buttons are simple to use; property *space low* means that toggle buttons do not optimize space; properties *enum* and *boolean* mean that toggle buttons can be used to manipulate enumerations and booleans. At runtime, these properties are used to find widgets appropriate to the current context.

ADAPTATION PROCESS AT RUNTIME

This section details the adaptation process at runtime. This process begins when the current context is modified. The *context reasoner* analyzes the new context to determine actions, presentations, interactions, and widgets that will compose the adapted UI. The *weaver* associates WIMP interactions and widgets to instruments. The *UI composer* adapts the UI to reflect the modifications.

Reasoning on Context

The context reasoner is dynamically notified about modifications of the context. On each change, the reasoner follows these different steps to adapt actions, presentations, instruments, interactions, and widgets, to the new context:

```

1 foreach Context change do
2   Re-evaluate mappings to enable/disable
   actions
3   Disable instrument's links that use disabled
   actions
4   Enable instruments's links that use enabled
   actions
5   Disable instrument's links which interaction
   cannot be performed anymore
6   Disable instruments with no more link enabled
7   Select presentations by reasoning on
   properties
8   Select interactions/widgets for instruments by
   reasoning on properties
9 end

```

Algorithm 1. Context reasoner process

The process of enabling and disabling actions (line 2 of Algorithm 1) is performed thanks to the context-to-action mapping: if the change of context concerns a mapping, this last is re-evaluated. For instance with the home automation example, when the user switches from the nurse to the elderly person, mappings described in the previous section are re-evaluated. Actions that inherit from **NurseAction** are then disabled.

Once actions are updated, instruments are checked: instrument's links that use the disabled, respectively enabled, actions are also disabled, respectively enabled (lines 3 and 4). Links using interactions that cannot be performed anymore are also disabled (line 5). For example, vocal-based interactions can only work on platforms providing a microphone. Instruments with no more link enabled are disabled (line 6).

Presentations that will compose the UI can now be selected (line 7). This process selects presentations by aligning their properties with those of the current context. In the same way, WIMP interactions and widgets are selected for instruments (line 8) using properties. These selections can be performed by different kind of optimization algorithms such as genetic algorithms or Tabu search. These algorithms are themselves components of the system. That allows to change the algorithm at runtime when needed.

We perform this reasoning on properties using the *genetic algorithm* NSGA-II [12]. Genetic algorithms are heuristics that simulate the process of evolution. They are used to find solutions to optimization problems. Genetic algorithms represent a solution of a problem as a chromosome composed of a set of genes. Each gene corresponds to an object of the problem. A gene is a boolean that states if its corresponding object is selected. For example with our UI adaptation problem, each gene corresponds to a variable part of the UI (the nurse actions, the toggle button aspect, the list aspect, the different presentations, *etc.*). The principle of genetic algorithms is to randomly apply genetic operations (*e.g.* mutations) on a set of chromosomes. The best chromosomes are then selected to perform another genetic operations, and so on. The selection of chromosomes is performed using fitness functions that maximize or minimize objectives. In our case, objectives are properties defined by the developer. For instance readability is an objective to maximize. For each chromosome its readability is computed using the readability value of its selected gene:

$$f_{readability}(c) = \sum_{i=1}^n prop_{readability}(c_i)x_i$$

Where $f_{readability}(c)$ is the fitness function computing the readability of the chromosome c , c_i is the gene at the position i in the chromosome c , $prop_{readability}(c_i)$ the value of the property *readability* of the gene c_i , and x_i the boolean value that defines if the gene c_i is selected. For example :

$$f_{readability}(001100111001011) = 23$$

The fitness functions are automatically defined at design time from the properties used by the interactive system.

Chromosomes that optimize the result of fitness functions are selected. Constraints can be added to genetic algorithm problems. In our case a constraint can state that the gene corresponding to the calling emergency service action can be selected only if there is a line phone in the house.

When genetic algorithms are stopped, they provide a set of solutions that tend to be the best ones.

Weaving Aspects to Complete Models

Once interactions and widgets are selected, they must be associated with their instruments. To do so, we reuse the process proposed in the DiVA project to weave aspects with models. An aspect must specify where its content (in our case the interaction and possible widgets and components) must be inserted: this is the role of the *pointcut*. In our case pointcuts target instruments and more precisely an action and the main class of the instrument. An aspect must also define its *composition protocol* that describes how to integrate the content of the aspect into the pointcut.

Composing and Updating the User Interface

The goal of the UI composer is two-fold: 1) It composes the selected presentations and widgets at startup. 2) Once composed, the UI composer updates the UI on context changes if necessary. Because modifications of the UI must be smooth enough not to confuse the users, the UI must not be recomposed from scratch using 1). The existing UI must be updated to minimize graphical changes and to keep usability.

EVALUATION

Our proposal is based on two hypotheses: 1) it tames the combinatorial explosion of complex interactive systems adaptations; 2) adaptations performed using our proposal are well adapted to the current context. We evaluated these two hypotheses by applying our proposal to EnTiMid, a middleware for home automation. Each component of the UI of EnTiMid is developed with the Kermeta implementation of Malai. At the end of the conception time, executable models are compiled as OSGi components [25] to run on the top of DiVA. The use of OSGi permits instruments, actions, and presentations to be easily enabled and disabled at runtime.

The experiments described in this section have been performed on Linux using a laptop with a Core2Duo at 3.06GHz and 4Gb of RAM. Each result presented below is the average result of 1000 executions.

EnTiMid: a Middleware for Home Automation

EnTiMid is a middleware for home automation. It notably addresses two issues of the home automation domain, by providing a sufficient level of abstraction. The first issue is about interoperability of devices. Built by many manufacturers, devices are often not compatible with one another because of their communication protocol. EnTiMid offers a mean to abstract from these technical problems and consider only the product's functionalities.

The second issue is about adaptation. Changes in the deployed peripherals or in the user's habits imply changes in the interactive system dealing with the home. Moreover, many people with different skills will have to interact with the interactive system, and the UI must adapt to the user. Considering models at runtime, EnTiMid permits such dynamic adaptation.

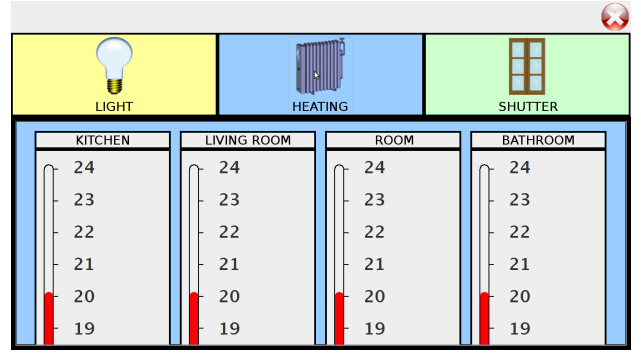


Figure 4. Part of the EnTiMid UI that controls the lights, the heaters, and the shutters of the home

Figure 4 shows a part of the EnTiMid's UI that manages home devices such as heaters, shutters, and lights. A possible adaptation is if the home does not have any shutter, related actions will be disabled and the UI adapted to not provide the shutter tab.

Hypothesis 1: Combinatorial explosion taming

We evaluate this hypothesis by measuring the adaptation time of five versions of EnTiMid, called v_1 to v_5 . These versions have an increasingly level of complexity, respectively around 0.262, 0.786, 4.7, 42.4, and 3822 millions of configurations. These different levels of complexity have been obtained by removing features from version v_5 . A configuration defines which components of the interactive system are enabled or disabled.

The adaptation time starts after a change of context and ends when the UI is adapted accordingly. The adaptation time is composed of: the time elapsed to select the optimal possible configuration in a limited time; the time elapsed to reconfigure the interactive system and its UI.

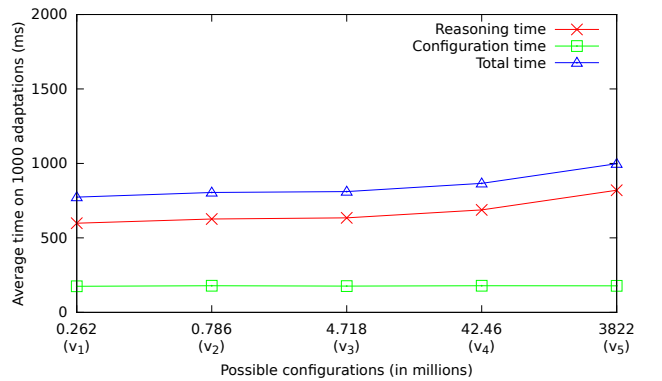


Figure 5. Average adaptation time of EnTiMid using an increasing number of possible configurations

Figure 5 presents our results using the reasoner based on the NSGA-II genetic algorithm. It shows that the reasoning time remains linear between 600 and 800ms. That because the parameters of the reasoner (*e.g.* the number of generations, the size of the population) are

automatically modified in function of the complexity of the system to run between 500 and 1000ms. Figure 5 also shows that the configuration time (*i.e.* when the system and its UI are modified) remains constant around 200ms. That brings the full adaptation time to around 1 second for the most complex version of EnTiMid.

Hypothesis 2: Adaptations quality

Finding a configuration in a limited time makes sense only if the configuration found is of good quality. Thus, we now evaluate the quality of the configurations found by the genetic reasoner in the limited time-slots described above. We compared these configurations with the optimal configurations. Optimal configurations are configurations giving the best results using the fitness functions. These optimal configurations have been computed by an algorithm exploring all the solutions. Such computations took 4.5s, 10s, 480s, and 7200s for respectively v_1 , v_2 , v_3 , and v_4 . We were not able to compute the optimal solutions of v_5 due to time and resource constraints.

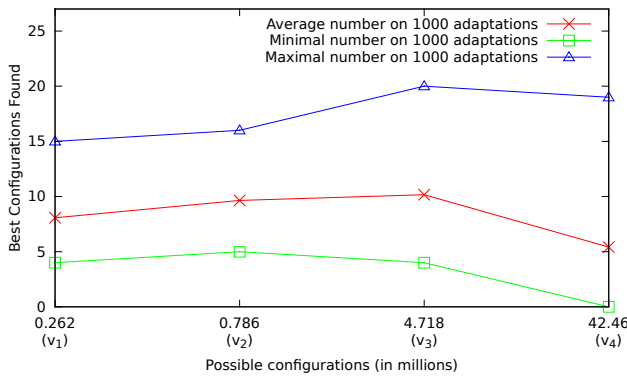


Figure 6. Comparison between the optimal solutions and solutions found by the genetic reasoner

Figure 6 presents the number of optimal configurations found by the genetic reasoner with v_1 , v_2 , v_3 , and v_4 . In average the reasoner always found optimal configurations for every version of EnTiMid tested. However, the performance slightly decreases while the complexity increases. For example with v_4 , several adaptations among the 1000 performed did not find some of the optimal configurations. This result is normal since we cannot obtain same quality results in the same limited time for problems whose complexity differ.

We can state that the genetic reasoner gives good results for EnTiMid. But it may not be the case for less complex or different interactive systems. One of the advantages of our proposal is that the reasoner is also a component that can be selected in function of the context. For instance with a simple interactive system (*e.g.* 10000 configurations), the selected reasoner should be a reasoner that explores all the configuration since it will not take more than 0.5s.

Threats to validity

An important remark on this evaluation is that in our current implementation the configuration quality does not include the evaluation of the usability of adaptations, nor the user's satisfaction. For example our process may perform two following adaptations provoking big changes in the UI, that may disturb the user. Such evaluations can be performed by:

- The reasoner while selecting a configuration. In this case, the previous UI will be integrated into the genetic algorithm under the form of fitness functions maximizing the consistency of the adapted UI.
- A configuration checker that would evaluate the best configuration among the best ones found by the reasoner.

The configurations found by the genetic reasoner mainly depend on the properties defined on the components of the interactive systems. The developers have to balance them through simulations to obtain good results [14].

This paper does not focus on the UI composition. The UI composer used in this evaluation is basic and takes a negligible amount of time during the reconfiguration. The use of a more complex composer will slow down the configuration process.

RELATED WORK

The conception of dynamic adaptable systems has been widely tackled in the software engineering domain [20]. Software engineering approaches use model-driven engineering (MDE) to describe the system as a set of models. These models are sustained at runtime to reflect the underlying system and to perform adaptations. This process thus bridges the gap between design time and runtime. Yet these approaches do not focus on the adaptation of UIs. For example in [9], Cetina *et al.* propose an approach to autonomic computing, and thus to dynamic adaptation, applied on home automation. This approach lacks at considering the system as an interactive system whose UI needs adaptations.

Based on MDE, UI adaptation has been firstly tackled during design time to face the increasing number of platforms (*e.g.*, Dygmes [11], TERESA [19] and Florins *et al.* [15]). These adaptation approaches mainly follow the CAMELEON top-down process composed of 1) the task model 2) the abstract UI 3) the concrete UI 4) the final UI [8]. Using the CAMELEON process, developers define several concrete UIs using one abstract UI to support different platforms. Users and the environment have been also considered as adaptation parameters, such as in UsiXML [18] and Contextual ConcurTaskTrees [3]. A need to adapt at runtime UIs thus appear to face to any change of user, environment and platform.

Approaches have been proposed to consider models of UIs at runtime [2, 24, 6]. In [7, 6], Blumendorf *et al.*

propose a framework for the development and execution of UIs for smart environments. Their proposal shares several points with ours: the use of a mapping metamodel to map models; they consider that bridging design time and runtime implies that models are executable. However, they focus on the link between the models and the underlying system while we focus on the adaptation of complex interactive systems.

In [24], Sottet *et al.* propose an approach to dynamically adapt plastic UI. To do so, a graph of models that describe the UI is sustained and updated at runtime. The adaptation is based on model transformations: in function of the context change, the appropriate transformation is identified and then applied to adapt the UI. This process follows the event-condition-action paradigm where the event is the context change and the action the corresponding transformation. The main drawbacks of this approach are that: transformations must be maintained when the interactive system evolves; the development of complex interactive systems will lead to the combinatorial explosion of the number of needed transformations.

CAMELEON-RT is a conceptual architecture reference model [2]. It allows the distribution, migration, and dynamic adaptation of interactive systems. Adaptations are performed using rules predefined by developers and users, or learned by the evolution engine at runtime. A graph of situations is used to perform adaptations: when the context changes, the corresponding situation is searched into the graph. The found situation is then provided to the evolution engine that performs the adaptation. This approach focuses on the usability of adaptations. However, it can hardly deal with complex systems because of the need to define a graph of situations.

ReWiRe is a framework dedicated to the dynamic adaptation of interactive systems [26]. As in our approach, ReWiRe's architecture uses a component-based system that facilitates the (de-)activation of the system's components. But ReWiRe suffers from the same main limitation than CAMELEON-RT: it can hardly deal with complex systems because of the increasing complexity of the ontology describing the whole runtime environment.

In [13], Demeure *et al.* propose a software architecture called COMETs. A COMET is a task-based interactor that encapsulates different presentations. It also embeds a reasoner engine that selects the presentation the more adapted to the current context. While we define a unique reasoner for the entire interactive system, COMETs defines one reasoner for each widget. We think that tagging widgets with properties that a global reasoner analyzes is a process that requires less effort than defining several reasoners.

The approach presented in [17] is close to COMETs where UI components can embed several presentations

and an inference engine deducing from the context the presentation to use.

In [23], Schwartz *et al.* propose an approach to adapt the layout of UIs at runtime. They show that the UI composer *must* also be context-aware to layout UIs in function of the current user and its environment. For example, our reasoner decides the components that will compose the UI, but not their disposition in the adapted UI. It is the job of the UI composer that analyzes the context to adapt the layout of the UI accordingly.

DYNAMO-AID is a framework dedicated to the development of context-aware UIs adaptable at runtime [10]. In this framework, a forest of tasks is generated from the main task model and its attached abstract description. Each task tree of this forest corresponds to the tasks possible for each possible context. Because of the combinatorial explosion, such process can be hardly scalable to complex interactive systems.

In [16], Gajos and Weld propose an approach, called Supple, that treat the generation of UIs as an optimization problem. Given a specific user and device, Supple computes the best UI to generate by minimizing the user effort and respecting constraints. This approach is close to our reasoning step. However, Supple is not MDE-driven and only consider user effort as objective while our approach allows developers to define their own objectives.

CONCLUSION

Adapting complex interactive systems at runtime is a key issue. The software engineering community has proposed approaches to dynamically adapt complex systems. However, they lack at considering the adaptation of the interactive part of systems. In this paper, we have described an approach based on the Malai architectural model and that combines aspect-oriented modeling with property-based reasoning. The encapsulation of variable parts of interactive systems into aspects permits the dynamic adaption of user interfaces. Tagging UI components and context models with QoS properties allows the reasoner to select the aspects the best suited to the current context. We applied the approach to a complex interactive system to evaluate: the time spent adapting UIs on context changes; the quality of the resulting adapted UIs.

Future work will focus on the consideration of adaptations quality during the reasoning process. It will assure consistency between two adapted UIs. Work on context-aware composition of UIs will be carried out as well.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Communitys Seventh Framework Program FP7 under grant agreements 215412

(<http://www.ict-diva.eu/>) and 215483 (<http://www.s-cube-network.eu/>).

REFERENCES

1. International Workshop on Aspect-Oriented Modeling. <http://www.aspect-modeling.org>.
2. L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary. CAMELEON-RT: A software architecture reference model for distributed, migratable, and plastic user interfaces. In *EUSAI*, pages 291–302, 2004.
3. J. V. d. Bergh and K. Coninx. Contextual concurtasktrees: Integrating dynamic contexts in task based design. In *Proc. of PERCOMW '04*, page 13, 2004.
4. A. Blouin and O. Beaudoux. Improving modularity and usability of interactive systems with Malai. In *Proc. of EICS'10*, 2010.
5. A. Blouin, O. Beaudoux, and S. Loiseau. Malan: A mapping language for the data manipulation. In *Proc. of DocEng '08*, pages 66–75, 2008.
6. M. Blumendorf, G. Lehmann, and S. Albayrak. Bridging models and systems at runtime to build adaptive user interfaces. In *Proc. of EICS'10*, 2010.
7. M. Blumendorf, G. Lehmann, S. Feuerstack, and S. Albayrak. Executable models for human-computer interaction. In *Proc. of DSV-IS'08*, 2008.
8. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting With Computers*, 15(3):289–308, 2003.
9. C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42:37–43, 2009.
10. T. Clerckx, K. Luyten, and K. Coninx. DynaMo-AID: A design process and a runtime architecture for dynamic model-based user interface development. In *Proc. of EIS'04*, 2004.
11. K. Coninx, K. Luyten, C. Vandervelpen, J. V. den Bergh, and B. Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In *Proc. of MobileHCI'03*, pages 256–270, 2003.
12. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2002.
13. A. Demeure, G. Calvary, and K. Coninx. COMET(s), a software architecture style and an interactors toolkit for plastic user interfaces. In *Proc. of DSV-IS'08*, pages 225–237, 2008.
14. F. Fleurey and A. Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proc. of MODELS'09*, 2009.
15. M. Florins and J. Vanderdonckt. Graceful degradation of user interfaces as a design method for multiplatform systems. In *Proc. of IUI '04*, pages 140–147, 2004.
16. K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *Proc. of IUI '04*, pages 93–100, 2004.
17. A. Hariri, D. Tabary, S. Lepreux, and C. Kolski. Context aware business adaptation toward user interface adaptation. *Communications of SIWN*, 3:46–52, 2008.
18. Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, and D. Trevisan. UsiXML: a user interface description language for specifying multimodal user interfaces. In *Proc of WMI'2004*, 2004.
19. G. Mori, F. Paternó, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30:507–520, 2004.
20. B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *Proc. of ICSE'09*, 2009.
21. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, pages 264–278, 2005.
22. F. Paternò, C. Mancini, and S. Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Proc. of INTERACT '97*, pages 362–369, 1997.
23. V. Schwartz, S. Feuerstack, and S. Albayrak. Behavior-sensitive user interfaces for smart environments. In *Proc of ICDHM '09*, pages 305–314, 2009.
24. J.-S. Sottet, V. Ganneau, G. Calvary, J. Coutaz, J.-M. Favre, and R. Demumieux. Model-driven adaptation for plastic user interfaces. In *Proc. Of INTERACT 2007*, pages 397–410, 2007.
25. The OSGi Alliance. OSGi service platform core specification, 2007. <http://www.osgi.org/Specifications/>.
26. G. Vanderhulst, K. Luyten, and K. Coninx. ReWiRe: Creating interactive pervasive systems that cope with changing environments by rewiring. In *Proc. of the 4th International Conference on Intelligent Environments*, pages 1–8, 2008.