ALIAS: A SET OF ABSTRACT LANGUAGES FOR USER INTERFACE ASSEMBLY

Anne-Marie Dery-Pinna, Cédric Joffroy, Philippe Renevier, Michel Riveill, Christophe Vergoni
Equipe Rainbow, Laboratoire I3S, CNRS
930, route des Colles BP 145
06903 Sophia Antipolis Cedex, FRANCE
(pinna, joffroy, renevier, riveill, vergoni}@polytech.unice.fr

ABSTRACT
Users’ devices and users’ web services constitute a changing environment, leading to needs for User Interface (UI) and service adaptations. As User Interface and services are separately developed, we propose an approach to abstract User Interface into a set of abstract languages (XML dialects) which describes different specific parts of a User Interface. The main goal of this set of languages is to compose heterogeneous User Interface in view to concretize the result in an existing language like Xaml or Flex.

KEY WORDS
Graphical User Interfaces, User Interface Composition, User Interface Description Language

1 Introduction

Ambient computing enables users to interact with their environments made of various and dynamic interacting devices (desktops, laptops, PDAs, smart phones, etc.) and services (timetable, diary, home automation, etc.). Software applications behavior and interactivity are then constantly changing because of the appearance and disappearance of services exposed by the different devices close to the user. Two complementary computer science research fields aim at taking into account these new usages and variation of context: service-oriented programming and plasticity (as part of the research in Human Computer Interaction (HCI)).

In one hand, the service approach [1] aims at developing independent and reusable entities, hiding middleware and facilitating components deployment. Thus, parts of applications are available on-line, ready to be used without specific development. Service approach focuses on the ability of the Functional Core (FC, the set of business services) of the application to react to changes in its environment. In this context, business services are first class entities that make possible to easily and modularly adapt the FC by services compositions explored with data flow compositions [2] or event flow compositions [3]. But, even if you know combining many FC services together, you do not know how to reuse their User Interface (UI), how to compose former UI components, so you have to develop a new UI from scratch (or with manual reuse of lines of code).

In the other hand, the plasticity [4] aims at enabling users to interact with computer tools on any device. The first issue of the plasticity was the projection on many platforms: specify once, generate many times. The second issue is to specify adaptive components [5], enable to react to a platform change or a usage change. The third issue is XML abstract definitions of UI ([6], [7], [8]) integrated into elaborate frameworks [9], [10], [11]. But the plasticity only focuses on the interactions with users. Environment changes adaptation plays an integral part in plasticity, but the emphasis preserves usability of the interface. So UI abstractions are based on the structure of the UI, in order to concretize it as the designer thought it. However, even if the composition of UI is possible [12], it is based on the structure and does not take into account the FC intelligence. Consequently, there is a lot of lost information useful to compose UI (interaction dialogs, interaction sequences, etc.).

Thus there is a need to construct an intermediary level between business assemblies and user interface assemblies. We focus on the composition of UI as first aim in order (i) to allow heterogeneous UI compositions and (ii) to generate UI compositions as a platform independent result intended by the FC for users.

We work on an intermediary level between services and UI in order to allow FC services adaptations and deducing information for resulting UI adaptations and vice-versa. One part of our work is inspired by MDE and we define a pivot model: ALIAS (Abstract Language for Interface Assemblies), composed at this time by three modeling languages ALIAS-Behavior (for modeling UI inputs, outputs and actions), ALIAS-Structure (for modeling more UI structures) and ALIAS-Layout (for modeling UI layout). In addition of purposes (i) and (ii) above, the ALIAS languages are design (iii) to be concertized in different abstract UI (Flex, XAML, UsiXML, etc.) and (iv) to be deduced from an abstract UI.

In the next sections, we illustrate our work in a case study and briefly describe our global framework. In the section 3, we describe the ALIAS languages and give an overview of a composition process. Finally, we discuss our results and conclude with perspectives.
2 Case Study and Framework

In this section, we first present the argument for UI Compositions through a case study. Then we present the global framework, in which ALIAS languages are inscribed in.

2.1 Case study

An application framework is a set of several services. Those services may be implemented as Web Services, rich interface on the Web or heavy clients. For example, there are a classical email service, a basic bookmark manager and a note editor. Sometimes, you may want to merge such applications in only one because your aim (task) needs the three applications. Typical case is when you are searching information on the Web, you may use any of the three applications to find again web addresses: you may send or receive web addresses by email; you obviously have access to your bookmarks with the manager; and you may note some web addresses on a memo during a meeting or a more informal talk. Even more when you want to send notes or bookmarks by email, if the screen you use is too small, it may be painful to switch from one application to another.

Moreover, the appearance of services is dynamic. The notes service may use a database server to store memos. So the service may be not always available: network access restriction, network or server troubles, etc. When the service appears, the UI has to be adapted by adding some UI components in order to enable users to use the "new" service. When it disappears, the UI has to be adapted by removing the UI component associated to the disappearing service in order to provide to users the honestest and the most efficient possible UI. To fulfill users' expectations, those adaptations must be performed dynamically in a reasonable time.

Despite our example simplicity, such requirements are concrete. An ubiquitous Information System in changing work environments (from a site to another, with different aims, with different networks, with different services) like those installed in universities or tourism guide systems are possible application domains.

So, this case study illustrates the needs of adaptation of the UI. Appearance and disappearance of services at runtime require a dynamic adaptation and from this adaptation can result a composition of UI. This composition can be an union of two or more UI or can be more complex like an extraction of some elements of UI.

In the next subsection we describe our global framework and compare it with the existing approaches for dynamic adaptations.

2.2 Framework

Our purposes are not only to compose User Interfaces (UI) but also to propagate compositions from the Functional Core (FC) level to the UI level. The ALIAS languages are used at different steps to facilitate the global process of composition. The framework is designed: (1) to transform ALIAS-Behavior into a service (or an orchestration of services). The objective is to find an available service corresponding to the behavior description. From composition of ALIAS-Behavior the framework creates the orchestration of the user needed services. (2) to propagate orchestration in the FC level to the UI level. We aim at deducing the ALIAS composition from the FC composition. Developers’ decisions are expected, but we want to provide at least an automated composition as possible. Thus, the reusing of UIs will be optimum. Here we will focus on the main role of the UIs and more specifically ALIAS-Behavior to facilitate composition.

In the Figure 1, you can see an overall of our framework. Arrows represent available transformations from a level to another. At the top of the Figure 1, there is the "final" UI, which is "really" in contact with users, like the UI representation of the note or email services. Below, there is the "concrete" or "abstract" UI, which are platform-independent UI descriptions. All terms "final", "concrete" and "abstract" are well-known levels of UI description in plasticity. Flex [8] and XaML [7] are "concrete" UI Languages, including layout and style information. Flex and XaML are provided with projection in the final UI level, respectively in Flash (.swf) and Win Form.

The bottom of the Figure 1 is the FC level. The FC is by definition composed of many services. As a result of research in Service-Oriented Architecture, those services may be combined in orchestration [13]. That is why services are represented by a gear on a puzzle piece that can be "added" (composed) with other services.

The middle of the Figure 1 is the ALIAS Level. ALIAS-Behavior only contains inputs of the UI that are parameters of the FC, outputs of the UI that are the results of the FC and actions that are events in both way (new information to display, new click of the user, etc.). The Figure 1 focuses on UI composition, but you can see in the dashed frame the connection between a service and its UI thanks to the ALIAS level. A service may express its parameters and results with ALIAS-Behavior without any dependencies of the final UI. This correspondence between services and ALIAS-Behavior is visible in the Figure 1 through dashed lines connecting puzzle pieces. A UI component may express what users enter and what kind of information it display to users with ALIAS-Behavior without any dependency of the service from which arrives information. Of course, that intermediate level may require some adapters (in reference in the design pattern "Adapter"). This use of ALIAS-Behavior is inspired by the PAC model [14]. The transformations between ALIAS-Behavior and abstract/concrete UI are in both ways. We generate ALIAS-Behavior description from a UI specified in Flex or XaML. Moreover, we can transform ALIAS-Behavior description in Flex or XaML UI. Without further information, that transformation is simple, with a basic layout. So, in order to work with more organized UI, we use the other ALIAS-languages. The ALIAS-Layout de-
scription for example can be deduced or can be designed apart from an existing description in Flex or XaML. Another strong point of the ALIAS level is the composition. Since we can compose ALIAS description and transform Flex and XaML to ALIAS and vice versa, we can compose UIs specified in different languages and finally generate a new UI without code writing.

Now you know our global framework, ALIAS languages and composition are fully described in the next section.

3 ALIAS: a set of languages for User Interfaces composition

In this section, we are going to present you how the set of ALIAS languages divides compositions of User Interfaces (UI) in several "smallest" and probably simplest compositions, and then the ALIAS languages.

3.1 UI Compositions

Today compose heterogeneous Functional Core is possible however nothing is done to do it with final UI. The latter is complex according to all preoccupations that UIs assemble (tasks, interaction, style...). A way to compose UIs is to separate all these different aspects. To compose heterogeneous UIs, no matter what is the description, we need "something" to unify them in view to re-transform them in one or other UI description. To do that in a general case, we propose an abstraction of different existing aspects of an UI: ALIAS, a set of languages. Each language has its own specificity to address only one preoccupation.

In case of applications designed by composition of services, the UI must be adapted. For example, if the developer integrates a new service (i.e. email service), its UI has to be added to the previous UI to allow the usage of sending and receiving mail. If that service is removed, its usage must be removed from the UI. To solve such evolutions, it is important to define composition rules on UI, which concerns different aspects:

- The structure of the UI; how merging two labels, buttons or menus. In the given example of emails, bookmarks and notes in the sub-section 2.1 and illustrated in the Figure 2, there are two "Content:" textArea in Mail and Note. The composition rule must answer to the following question: must the composed UI still have two Content textAreas or only one?

- The behavior of the UI, its connection with the functional core; how merging two actions? In our examples, a click on "create" button must create a Note or a Bookmark? If the FC assembly is to create one or other feature or to create both at the same time, we
will have different composition.

- The layout of the UIs, which rules must be applied to organize the structure on the final UI? In the example, the three main frames will be placed following a simple vertical layout (one above, the second below the first, the third below the second)? Another more complex layout? in which order?
- The style (color, image, font, etc.). Which one must be kept? Which one must be changed?

Some of our past researching work on such rules [15], and recent works on UsiXML [6] develop the compositions. Once the behavior composition is determined, structure composition should be in part deduced (two labels are common and only one is sufficient because they reference the same FC data) and then specialized. The others aspects allow to reuse more part of existing UI (layout or style) as possible, even if we do not work on "style" yet.

In the following we will illustrate ALIAS with the examples presented in the subsection 2.1 and shown in the Figure 2. Suppose that we have three different applications: one for mail, one for bookmarks and one for notes. Each application has its own User Interface written in different languages (Xaml, Flex). So, we are in the case where we have heterogeneous UI that we would like to combine to make only one application.

We present here the two UI descriptions of Mail (in Flex, Figure 3) and Notes (in Xaml, Figure 4) to briefly compare them.

![Figure 3. Flex source code of Mail UI](image)

![Figure 4. Xaml source code of Mail UI](image)

In those samples, we can notice that Flex and Xaml describe structures such as label mx:TextArea (in Flex) and Textbox (in Xaml) and buttons. Our objectives are to identify similar structures and behaviors to facilitate composition.

### 3.2 ALIAS Languages

For the moment, ALIAS only abstract the Presentation part of a UI. To do that, we put in place three languages:

- **ALIAS-Behavior**: this language describes with what the user can interact. It also makes the correspondence with the functional core. Thus, it keeps coherence when we would like to compose UI with the functional core,

- **ALIAS-Structure**: this language specify the ALIAS-Behavior. With this language, it is possible to know more about elements present in the UI,

- **ALIAS-Layout**: this language organizes all elements we have in ALIAS-Behavior. It manages the placement. It is more useful at the concretization step and at the abstraction step to keep more information from the original UI.

In those samples, we can notice that Flex and Xaml describe structures such as label mx:TextArea (in Flex) and Textbox (in Xaml) and buttons. Our objectives are to identify similar structures and behaviors to facilitate composition.

#### 3.2.1 ALIAS-Behavior

As we said before, ALIAS-Behavior is a language that describes each element that the user can manipulate via the UI. As depicted in the Figure 5, we identify three kinds of elements which are: Input, Output and Action.

Input element corresponds to User Input. They are used to give information, select items... It also does the correspondence with the functional core parameters (input). Output element corresponds to information given by the functional core that we would like to display. Thus, we can see that we can keep coherence between the UI and the functional core and even more at the composition time.
Finally, Action element corresponds to elements that user can use to trigger event and so call operations from the functional core.

If we transform the Mail Flex code into ALIAS-Behavior, we obtain the code given in Figure 6.

We can see here four Input elements which correspond to inputs to send a mail. We abstract `mx:TextInput` of Flex in Input. It will also the same thing with Xaml, we will transform `TextBox` in Input. The button (`mx:Button`) that trigger the action is abstracted into an Action element.

### 3.2.2 ALIAS-Structure

ALIAS-Structure is here to specify elements present in ALIAS-Behavior. It describes Input and Output elements by adding some attributes. ALIAS-structure is defined by the class diagram of the Figure 7. With these attributes it is possible to say that an input (resp. output) is a string or an integer with inputContentType (resp. outputContentType). It also possible to abstract a List specifying the min/maxSelection.

ALIAS-Structure can also group ALIAS-Behavior elements. For example, if we have a menu with a lot of items, we can group these items to keep coherence.

Abstraction of this information is done at the same time that ALIAS-Behavior. We just extract more information about a concrete element to make possible a better concretization.

A more complete transformation of the Mail Flex code in ALIAS is illustrated in Figure 8 which is the generated code for ALIAS-Structure.

We can see here that we specify each Input by adding a caption which represent the label associated to the Input. We also specify the inputDataType which specify the type of the Input. ForceDefault attribute specify if we can change the value of an Input in the case that Input has some default values.

### 3.2.3 ALIAS-Layout

ALIAS-Layout keeps information about organization of UI elements into a hierarchy. Thus, it is possible to know to which layout belongs an UI element.

When we detect a layout container, we create a new layout element inside. Then we place inside this layout all element that there are. A layout can also contain another layout. Thus, we obtain a layout tree with ALIAS-Behavior elements as leafs.

We do not give here an example of layout. For the moment, the result of abstraction will be only one layout which contains all elements, because the source code in Flex contains only one layout (the main layout).

### 3.3 Process of composition

The basic process of composition is the following. From two existing UI (Mail and Notes for example), we trans-
form the two concrete UIs into ALIAS-Behavior trees. This point is illustrated in the previous sections. An application of a composition operator on the two ALIAS-Behavior trees is done. This operator can be deduced from the FC orchestration or given by a UI developer. The operators are: sum, intersection, union...

For the moment, the union operation compose two ALIAS-Behavior (and ALIAS-Structure) trees keeping all elements. If some elements have the same semantic, another operator union-minus keeps only one of the two elements.

In case of a more global reuse of the UI, we have to compose also ALIAS-Structure, and ALIAS-Layout.

4 Conclusion and Perspectives

In this paper, we show more specifically the ALIAS languages we have defined to manage in part UI composition. There are a lot of ways to describe UI like Xaml, MXML, XUL based on XML or Java Swing, Qt based on Object Oriented Languages. Our experimentation concerning UI and ALIAS transformations is only proposed on UI based on XML because those languages facilitate structuring and separation of the UI description, events and functional core. The difference with other works on UI description like UsiXML is that we have a separation of concern which is not the case in UsiXML. Another particularity of our works is to reuse different levels of UI to avoid a complete rewritten of an application. We manage in priority the behavior composition to reuse and to assure a good connection between FC and final UI. Finally, the UI developers who respect a good separation allow further to reuse UI at different levels.

Concerning ALIAS, we have two principal perspectives. First, to add more operations of composition and the second one, to concretize the different ALIAS languages in others target languages. This concretization is an important part to check if the composition is correct or not and if the different languages of ALIAS are sufficient.

Concerning more specifically ALIAS-Layout, for the moment, we do not add information about the positioning inside a layout. So in a future work, we have to abstract these layouts and positioning associated to go further.

The complete framework is still in progress with two main axes: the deduction of UI composition from business service orchestration and inversely the generation of a user dedicated application build of services orchestration from UI composition.

An open perspective, it is to add a new model that makes the link with the Functional Core. With this new model, we could react to elements composition of the Functional Core to compose UI Elements, and vice-versa. Thus, we will have a complete composition of all the different level that we have in the Figure 1.

References