Towards Conflict Management in User Interface Composition Driven by Business Needs

Audrey Occello, Anne-Marie Déry-Pinna, Michel Riveill

▶ To cite this version:

Audrey Occello, Anne-Marie Déry-Pinna, Michel Riveill. Towards Conflict Management in User Interface Composition Driven by Business Needs. 4th International Conference on Human-Centered Software Engineering (HCSE), Oct 2012, Toulouse, France. pp.233-250, 10.1007/978-3-642-34347-6_14 . hal-01342108

HAL Id: hal-01342108
https://hal.science/hal-01342108

Submitted on 5 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Towards Conflict Management in User Interface Composition Driven by Business Needs

Anne-Marie Dery-Pinna, Audrey Occello and Michel Riveill

Laboratoire I3S (Université de Nice - Sophia Antipolis - CNRS)
Bâtiment Polytech'Sophia – SI 930 route des Colles – B.P. 145
F-06903 Sophia Antipolis Cedex, France
{pinna, occello, riveill}@polytech.unice.fr

Abstract. This paper presents a composition engine that handles User interface (UI) in the context of application composition. The aim is to detect and manage conflicts that may arise when composing UI driven by business needs. The originality of this composition engine is to reason at an Abstract level which simplifies the composition algorithm and makes it reusable and oblivious to technology. The composition engine is the core of the Alias framework that reduces the re-engineering efforts needed to obtain the UI of an application built by composition of smaller ones following the "programming in the large" paradigm.

Keywords: User Interface composition, functional composition, composition conflicts.

1 Introduction

The "programming in the large" approach aims at constructing applications by composition of smaller ones. The presentation level is not considered in the paradigms that enable programming in the large such as Service Oriented Architectures (SOA) [1] or Component Based Software Engineering (CBSE) [2, 3]. In this context, developers usually need to apply a complete development cycle (from requirement analysis, task model, to tests through design and programming) to build the UI from scratch; they cannot reuse former UIs that come with the applications to be composed, pieces of them or even analysis elements.

Our goal is to reduce the re-engineering efforts needed to obtain the UI of an application built by composition. Application types are wide; we do not pretend to handle any kind of them. In fact we consider SOA applications built by composition of services. Even if the resulting application corresponding to a new service can be itself composed with other ones, this recursive composition process is not infinite. Hence, the granularity of the units to be composed is not coarse. This means that the corresponding UIs that we handle focus on rendering the use of service operations.

We propose a composition engine that exploits information about the functional composition (the business needs and then desired usages) to deduce which part of UIs

should be reused and which interaction links with the functionalities should be maintained. This deduction is based on an abstract representation which focuses on the UI composition needs in order to propose an algorithm reusable on several platforms. UIs, FCs and functional compositions are then expressed in a pivotal formalism [4].

The abstract representation used by the composition engine is based on an architectural decoupling derived from the separation of concerns principle. We assume that each application is made of two distinct parts: the functional core (FC), and an attached UI. This assumption is not very restrictive in the sense that many architectures prone for the separation of the UI from the FC like in the Arch model [5], in the PAC model [6] and in the MVC model [7] for example. The composition engine deduces UI composition based on functional composition provided that the functional composition (using orchestrations in SOA) is performed by the developers in response to end-user business needs after a requirement analysis: functional compositions may be seen as an implicit and partial incarnation of the desired usages expressed from the business point of view.

The composition engine output allows for the generation of a first sketch of the UI to the designer using transformation rules. This sketch reuses parts of former UIs, preserves the consistency between the functional level and the UI level respecting the composition needs and the previous interaction links between UIs and FCs in the composed applications. Then, the designer may concentrate on ergonomics. For this, the composition engine builds automatically an operational UI sketch for an application *A* as a function of: 1) the way FCs of smaller applications are composed to form *A* (functional composition); and 2) the interactions between each FC to be composed and their respective UIs. In this paper, we focus on the composition engine rules, on the UI composition conflicts that may arise, and on conflict management.

The remainder of this paper is organized as follows. Section 2 presents related word around composition. Section 3 describes a case study to illustrate the interest of using information about functional composition to drive the UI composition choices. Section 4 gives an outlook of the context of use of the composition engine: the Alias framework's process and the underlying models. Section 5 focuses on the composition engine and on composition conflict management. Section 6 evaluates the composition engine proposal. Last section concludes.

2 Related Work

We classify work around compositions according to the parts of applications they cover. We mentioned the functional core (FC) part and the UI part in the introduction. We add a third concern: Tasks are used in the HCI research field for user centered design and development in order to express end-user requirements. Task formalization allows designers, ergonomists end developers to agree and respect the user needs. Consequently it is important to consider works on Task composition as well as on FC or UI compositions. This classification highlights three categories of related work:

- Works only considering *functional composition* [8, 9, 10, 11] which results imply to build a new UI from scratch to use the composed application.

- Works only considering *UI composition*, either for defining specific toolkit for adaptive UI [12], either based on abstract definition of UIs [13, 14] or either adopting end-user programming [15] which results are UIs not linked to application functionalities (a non runnable application). Work around Mashups such as iGoogle and Netvibes enables end-user to create their own application. To achieve the composition, one can only juxtapose different applications in the same workspace leading to independent UIs.
- Works considering *several parts of applications* [16, 17, 18, 19, 20]. The interesting element in these approaches is that most of them exploit Task information either directly from Task analysis models either indirectly to propose a composition approach of entire applications.

Works of the two first categories are limited to truncated applications where a lot of additional development is needed. Our approach belongs to the third category of works that aim at composing entire applications to deliver runnable applications.

Yahoo!Pipes provides a Mashups environment to draw a workflow that aggregates information from multiple sources. However, users do not have any control on the UI produced by the workflow description. In [16], a planning problem describing user needs is transformed into a task representation, which is then transformed into a UI. A weak point of this approach is the independency of functional elements which are not composed. In [17, 18, 19, 20], composition only proposes to aggregate UIs without elements merging. [17, 18, 19] require a specific development for the UIs. [19, 20] do not use all the possibilities offered by the functional composition.

In contrast, our approach determines UI elements required in the composite application including the possibility to merge them. We do not have to realize a specific development to use our approach but only to follow a good separation of concerns.

Our originality is to establish a connection between the "FC domain" and the "UI domain" by deducing the UI composition from the functional composition and by conserving former associations between UI and FC. This connection preserves the consistency between the two main parts of an application. The reuse of former UIs assures a transfer of UI design analysis. Our proposition takes into account the fact that tasks are unfortunately not embedded in applications now but some of them are hidden in functional compositions which express business needs. Then a part of the desired usage is partially recoverable. For example, users' tasks chaining is injected as sequentiality or parallelism in the expression of the functional composition.

3 Building Application through Composition: The Human Resource System Case Study

In the rest of this paper, we focus on a Human Resource system case study. We consider two services: the first one provides access to Social Insurance account (e.g. French "carte vitale") and the other one provides access to a corporate directory (e.g. IBM blue pages). Users interact with these services through corresponding UIs.

For a user to interact with a new service built from these services, one must extract useful features from the existing services, to compose them using orchestrations and also to compose the relevant parts of the corresponding UIs consistently.

In the remainder of this section, we illustrate the interest of using information about functional composition to drive the UI composition choices using these services and two orchestration samples to show UI composition possibilities.

3.1 Service descriptions

Let *S1* be a service to access social insurance information. It exposes a *getByCard* operation with a *card identifier* input and with a *last name*, a *first name*, a *birthday*, a full social insurance *number*, a *medical referee* (the doctor that ensures a custom medical supervision), a *family status*, a *handicap* rate and an *address* outputs. Figure 1a outlines *UI1* the UI attached to the *S1* service.

Let *S2* be a service to access business information about an employee. It exposes a *getBusinessInfo* with a *full name* (last name plus first name) input and with a *full name*, a *position*, an *email* address, *office* numbers, *buildings* and *addresses* outputs. Figure 1b outlines *UI2* the UI attached to the *S2* service.

Insurance Card Id : 123456 [show insurance information]

Social Insurance Account

Last Name	Doe	**First Name**	John Peter
Birthday	1975-12-24		
Number	1751275056266	**Medical Referee**	Dr. Mabuse

Administrative Information

Family Status	married
Handicap	0%
Location	2010 promenade des Anglais 75000 PARIS FRANCE

(**a**) *UI1* attached to *S1*

Fellow : John Peter Doe [show fellow's information]

Full name: **John Peter Doe**
Email: John.Doe@jordan.eu
Position: Department Manager

Assignment(s)

Building:	MATISSE building	Building:	CHAGALL building
Office:	B455	Office:	C2412
Address:	9 Rue des Dames 75 009 Paris FRANCE	Address:	58 Avenue des Lys 75 022 Paris FRANCE

(**b**) *UI2* attached to *S2*

Fig. 1. Possible representation of UI

3.2 Composition cases

Let suppose now that the firm's Department of Human Resources needs some business and personal information about employees to manage sick leave for illness.

A first approach may be to use the two services simultaneously as in the mash up manner with the goal to group business and insurance information for a given person. The UIs are displayed side by side but there is no information exchange between the

two services. This kind of composition may engender a set of UI problems: The name is shown twice, addresses are dispatched in the two UIs and so on.

Another way to proceed is to compose the two services using orchestration mechanisms and reuse partially the former UIs to interact with the new service.

Let *Composition1* be the orchestration of *S1* and *S2* in sequence such that the data input of *S2* operation (the *full name*) is provided by the concatenation of the *first name* and *last name* outputs of S1 operation. *S3,* the new service resulting from this orchestration, exposes a *getEmployeeHRInfo* operation with a *card identifier* input (the data used to invoke *S1* in the orchestration). The outputs of *S3* operation are: the *first name*, the *last name*, the personal *address*, the professional *offices*, *buildings* and *addresses* the insurance *number*, the *medical referee*. This is not the union of *S1* and *S2* operation outputs because information such as the *handicap rate* are not needed by the HR Department and should not be disclosed for privacy considerations anyway. Let *UI3* be the UI attached to *S3* that reuses parts of *UI1* and *UI2* (Fig. 2).

Insurance Card Id : 123456 [show employee HR information]

Last Name Doe		**First Name**	John Peter
Number	1751275056266	**Medical Referee**	Dr. Mabuse
Location	2010 promenade des Anglais 75000 PARIS FRANCE		

Assignment(s)

Building:	MATISSE building	Building:	CHAGALL building
Office:	B455	Office:	C2412
Address:	9 Rue des Dames 75 009 Paris FRANCE	Address:	58 Avenue des Lys 75 022 Paris FRANCE

Fig. 2. A possible UI attached to S3

Let *Composition2* be a variation of *Composition1* where the personal *address* output from *S1* and the professional *addresses* output from *S2* are merged as they concern contact information in both cases. Then, *S4*, the service corresponding to this new composition, exposes an operation that holds a parameter less than the *S3* operation because of the merged outputs.

Figure 3 illustrates possible UIs that may be built from previous UIs reflecting the use of *S4*. In Figure 3a, addresses are stacked vertically and identified with a "location" label. In Figure 3b, addresses are laid out as a mosaic and identified with a "address" label.

(a) Addresses are represented like in *UI1*

(b) Addresses are represented like in *UI2*

Fig. 3. Possible UIs for *S4*

3.3 UI composition requirements

These two examples of composition highlight the fact that the way of composing *S1* and *S2* implies the creation of a new service which UI must be adapted to the new behavior. The analysis of the functional composition and of the former UIs should detect redundant display (e.g.: the name appearing twice in the composed UI) and not to display useless information (e.g. the family status). Moreover the new UI should ensure layout grouping for related information (e.g. contact information both present in *S1* and *S2* respective UIs). *S4* highlights a complexity in UI composition when the functional composition involves grouping data that have different presentation in the former UIs. This kind of conflict can be representative of incoherence in the way the user centered design analysis of the services has been done. How will the addresses be displayed? Following the recommendations in *UI1* like in Fig. 3a or following the recommendations in *UI2* like in Fig. 3b?

4 Context of Use of the Alias Composition Engine

This section gives an outlook of the context in which is used the proposed composition engine. We describe the composition process in which the engine is integrated, the underlying models on which the engine rely and an intuitive presentation of the proposed composition mechanisms.

The Alias process starts from a set of separate services with associated UIs and from a composition of these services. An automatic learning phase creates (1) abstract models of the relationships between UIs and FCs and between FCs. Then, the composition engine (2) analyzes all models with its set of composition rules and proposes (3a) potential conflicts or (3b) a possible UI for the service composition. The process ends with (4) the generation of code for the new UI and for the links to the FC part.

Figure 4 sums up the process steps. Steps 1 and 4 are platform dependent as we need to produce transformation rule for each technological space to be handled. Steps 2 and 3 are platform independent as the Alias composition engine is technological agnostic.

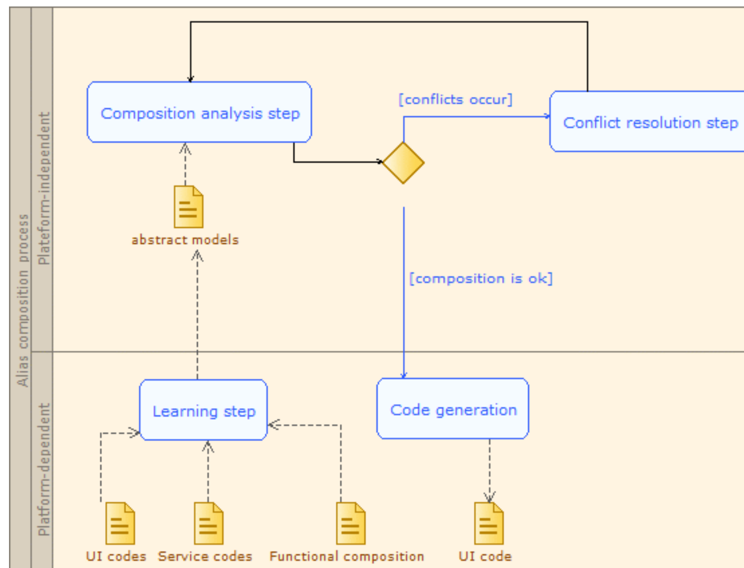

Fig. 4. Main composition steps of the Alias process

In Alias, each application is modeled as an assembly of a UI component and a FC component and compositions are expressed in a similar way to component assemblies such as in UML2.0 component diagram [21]. Describing formally the underlying models is not in the scope of this paper but they can be found in [4]. UI and FC parts of an application are represented as components with ports and compositions as bindings between components. However the granularity of the port is finer here: at the data or operation level not at the programming interface level. UI input ports correspond to user inputs (collected through widgets such as textfields, lists, checkboxes, …) and UI output ports correspond to system output (rendered through widgets such as labels, images, tables, …). FC intput ports correspond to operation input parameters and FC output ports correspond to operation output parameters. Trigger ports correspond to user actions in UI (fired through widgets such as buttons, menu items, …) and to operation calls in FC.

This abstract representation of applications is obtained by automatic transformations (see section 6). From this abstract representation, the composition engine produces a suggested UI (the first sketch of the composed UI). The suggested UI is then transformed into code making the application operational.

The bindings between a UI component and a FC component are very important for the UI composition computation as they help in identifying which functionality is related to which UI elements and vice versa. Searching for such bindings during the abstraction step makes it possible to differentiate functionalities related to user tasks

(functionalities that are called as a result from a user interactions and/or functionalities which results are presented to users) from system tasks (functionalities executed without user implication). During the abstraction step, only the first category of functionalities and only UI elements that are related to the FC are reified. As a consequence, we can observe an isomorphism between the ports of UI components and the ports of FC components in the Alias formalism.

The representation of the interactions between *S1* (resp. *S2*) and its attached UI in the Alias formalism is illustrated in Figure 5 which highlights the isomorphism between the components' ports. The downward arrows at the left of components connect user inputs (ex. *fullname*) to operation parameters and the upward arrows at the right of components connect operation results to their UI representation (*address*). The downward arrows at the middle of components link user actions to operation calls (*getBusinessInfo*).

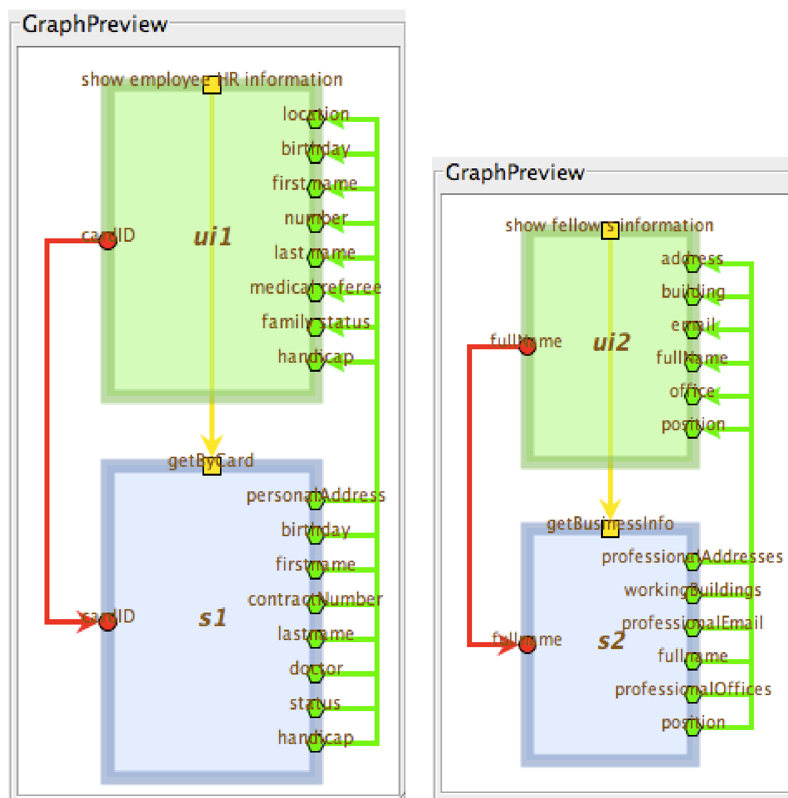

Fig. 5. Representation of S1/UI1 and S2/UI2 in Alias

In Alias, each functional composition is also modeled as an assembly of components. The bindings between FC components are very important for the UI composition computation as they help in identifying which functionalities and data are kept in the functional composition.

The representation of *Composition1* (the first orchestration of *S1* and *S2* services) in the Alias formalism is illustrated in the Figure 6. The *S3* component represented at the bottom of the figure is composed of the two former components: *S1* and *S2*.

The composition engine of Alias deduces which UI elements to reuse and which of them trigger conflicts by correlating information about: 1) functionalities and data that should be kept thanks to FC bindings and 2) their corresponding elements at the UI level thanks to UI to FC and FC to UI bindings. Potential conflicts may occur when two parts of former UIs are bound to the same operation of the new FC. Figure 6 shows the application corresponding to *Composition1*: UI3 is bound to *S3*.

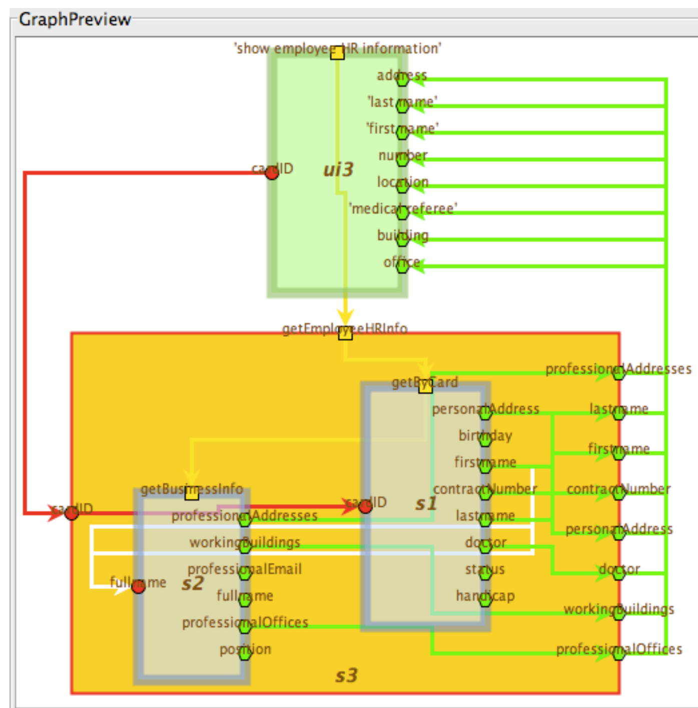

Fig. 6. Representation of S3 and its corresponding UI in Alias

The Alias framework has been implemented in Java for the abstraction and concretization steps (steps 1 and 4) and the composition engine has been implemented in Prolog (steps 2 and 3). The transformations involved in steps 1 and 4 have been written for Flex and SWING for the UI part, WSDL for the service part, and BPEL for the orchestration definition.

5 Alias Composition Engine

This section describes the composition engine which deduces a first sketch of the UI preserving the interaction consistency between the new UI and the new FC. The composition engine makes it possible to:

- **Identify** which UI elements of the former UIs to keep. In our example (see Figure 2), the *handicap* rate output is not exposed outside the "functional box" in the Figure 6. Then it is not represented in the UI component (*UI3*). The *medical referee* output is published (made available outside of the "functional box"). Then it is kept and represented in UI component (*UI3*).
- **Analyze** the data flows and reproduce them at the UI level.
- **Point out** the potential conflict points for UI composition.
- **Solve** the detected conflicts either automatically or interactively.

The different ways to solve conflicts are discussed informally in section 5.1. The different composition rules used by the composition engine to deduce the UI composition, to identify and solve the UI composition conflicts are described in section 5.2.

5.1 UI conflict management

a. UI Selection Solved implicitly by Functional Composition Choices. The functional composition indicates which UI elements to select automatically when: (1) an output of a FC is bound to the input of another one (workflow impact at the UI level) and (2) some outputs, inputs or functionalities are not kept in the final FC (implicit selection of UI elements). The following cases of UI element selection/ replacement /suppression is managed by the composition engine automatically (composition rule 1 in the section 5.2).

Case of UI selection deduced from input-output and input-input functional bindings. Using information from the functional compositions avoids reusing redundant UI elements. When FCs are bound, some inputs of a FC are automatically collected from outputs of other FCs. Such inputs are no more needed at the UI level. Then similar information may disappear implicitly. It avoids human mistakes or confusion caused by redundant information. When composed FCs are bound to internal FCs, the situation is slightly different: the inputs of the internal FC are filled with the inputs of the composed FC. Then the corresponding UI inputs must be kept.

In the example, the *fullname* input of *S2* is filled with the concatenation of the *firstname* and *lastname* outputs of *S1* then the input of *S2* is no more exposed by *S3* and *S4*. In consequence, the *fullname* is no more present in the new UI. The *contract number* input of *S1* is exposed by *S3* because it is needed by the *getEmployeeHRInfo* operation (the operation that calls the *getByCard* operation of *S1* in the orchestration). In consequence, the insurance *number* is kept in the new UI.

Case of UI selection/replacement deduced from the elements exposed in the functional composition (composite).

Using information from the functional compositions avoids reusing useless UI elements. When an input is not selected in the new behavior, the user should not have to give its useless value via the UI anymore. When a result is no more calculated in the new behavior, its visualization becomes obsolete. When functionalities are no more needed or when they shift from interactive tasks to system tasks (because of bindings), the composed FC does not exposed it anymore. Then, the UI elements corresponding to functionality triggers disappear as the user will not activate these functionalities anymore. On the other side, the internal functionalities not exposed but bound to a functionality of the composed FC are managed separately. Their corresponding UI element is replaced by that of the functionality of the composed FC.

In the example, UI elements like the *family status* are no more present in the new UI as *S3/S4* does not expose the corresponding outputs. Moreover, as the *getBusinessInfo* operation becomes a system task (*all* its inputs are automatically filled), the "show fellow's information" button is not kept in the new UI. Finally, the "show insurance information" button is replaced by a "show employee HR information" button in the new UI as the *getEmployeeHRInfo* operation of *S3/S4* calls the *getByCard* operation of *S1*.

b. Conflicts Solved explicitly by the Composition Engine. The composition engine cannot proceed to automatic UI element selection when several UI inputs (resp. outputs) are bound to a unique input (resp. output) of the composed FC (composition rules 2 and 3 in section 5.2). In such cases, the composition engine detects potential conflict points (composition rule 4 in section 5.2) and alerts the developers (Fig. 7). In the example, *Composition2* raises a conflict since there are two possibilities for displaying merged addresses.

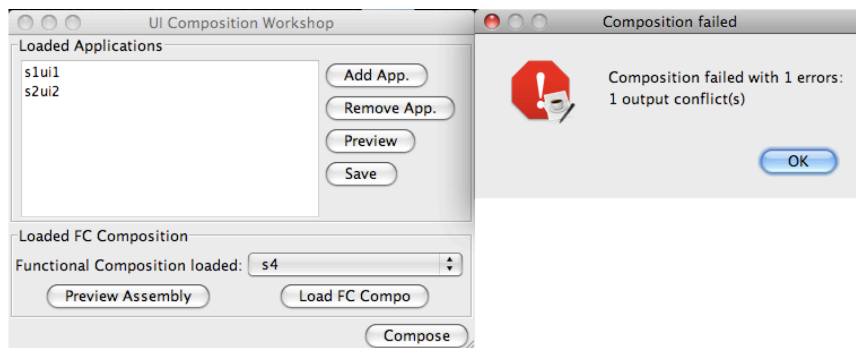

Fig. 7. Conflict detection in *UI4*

To proposed conflicts management methods are either to: 1) keep all UI elements, 2) keep one of the possible elements, or 3) create a new UI element. The framework may let the developers choose the conflict management method as shown in Figure 8. For example, the developer may decide between several former UI design in order to inform the engine about the best choice for usage.

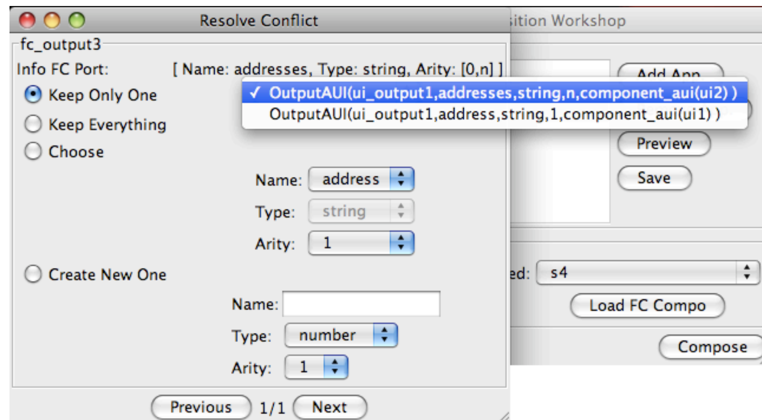

Fig. 8. Alias framework menu to choose the conflict resolution method (shown for *UI4* here)

In the example (*Composition2*), the representation of merged addresses may be (i) like in *UI1* or (ii) like in *UI2* or (iii) a new element may be created. This first possibility is illustrated in Figure 9 (the *location* element from *UI1* is kept whereas the *address* element from *UI2* is not).

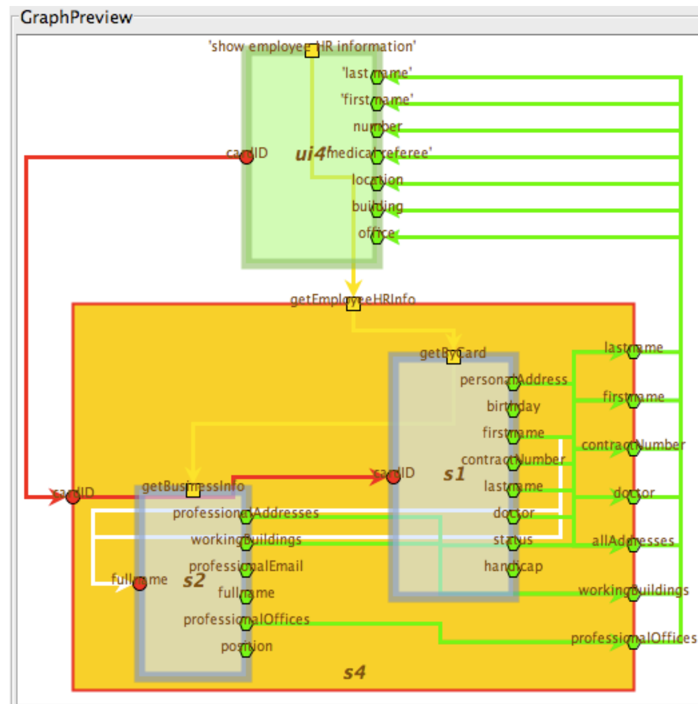

Fig. 9. A possible UI for S4 proposed by Alias as a function of the UI conflict resolution

5.2 Composition rules

The composition engine is based on a set of rules identifying the elements to keep from former UIs and detecting conflicts between UI elements.

Composition rule 1 - Potential Reused UI element detection.
This rule determines the set of UI elements which are not used in the new UI and thus the set of UI elements that may be reused in the new UI. This rule signals to the developers which UI elements are not kept by the functional composition. Then they may change the functional composition if they think that the removal of some UI elements highlights a consequence of a bad interpretation of the user requirements in term of functionalities.

Composition rule 2 - Potential conflicting UI output detection.
This rule determines the set of UI outputs which are potential conflict points for the new UI (bound to the same output of the new FC).

Composition rule 3 - Potential conflicting UI input detection.
This rule determines the set UI inputs which are potential conflict points for the new UI (bound to the same input of the new FC).

Composition rule 4 - Conflict resolution.
Several strategies could be adopted: 1) the engine may solve conflicts *automatically* using assumptions either provided by the developers or default choices; 2) the engine may also operate *interactively* asking the developers to remove the conflicts using an extra UI (the UI of the framework dedicated to conflict management).

Resolution algorithm or automatic solving. This algorithm takes in parameter the *predominant* UI and returns the set of kept UI elements. The predominant UI choice indicates which representation to advantage in case of ambiguities. This information may be given by the developers at the beginning of the process after an analysis of the hidden design choices made on the former UIs. The default value (the first UI declared in Alias) can be used to obtain a first sketch rapidly otherwise.

The conflict resolution algorithm uses information about UI elements: the kind of representation (multiple or single). A multiple representation serves to visualize a set of values (set of radio buttons, list box ...), a single representation serves to visualize a unique value (text field, label …). The conflict resolution algorithm also uses different UI composition operators to solve the conflicts: *selection*, *union* or *union without redundancy*. The underlying ergonomic rule is to minimize UI redundancy in order to avoid confusion in usage. The algorithm selects the best UI composition operator to apply depending on the kind of representation of the UI elements that are in conflict:

1. If there is only one “multiple” representation, then that representation is chosen to present the merged data
2. If there are several “multiple” representation, then the “multiple” representation of the predominant UI is chosen to present the merged data.

3. If all the representations are "single", then all the UI elements are kept. In this last case, the engine informs the developers of the semantic similarities between the kept UI elements. Such information helps the developers to select which UI elements to group when performing the layout of the new UI. It may also points out an inconsistency between the functional composition and user requirements that may imply revising the functional composition design.

Algorithm for interactive solving. Conflict resolution can be guided by the developers who can choose the composition operator they want to apply and the UI representation(s) they want to keep or the creation of a new UI element and its representation (this last choice is not possible in automatic mode) as shown in Figure 8.

Composition rule 5 - UI usability warning.

At this step, the labels associated with the elements of the new UI are compared. If labels with same names are associated with different UI elements, a warning is sent to the developers. The framework gives the possibility to the developers to rename some of them if they want.

Application to the case study.

In the second example of composition (*Composition2*), the composition engine deduces that the *fullname* input of *UI2* will not be reused in *UI3* as well as the *family status*, the *handicap* rate and the *birthday* outputs of *UI1* and the *email* and *position* outputs of *UI2* (composition rule 1). At this step, the developers can verify if the functional composition is well suited. If they feel that a UI element is missing and seems to lack to the usage (the worker *position* for example), they can decide to adjust the functional composition so as to keep it in the new UI.

The composition engine also deduces that there is a conflict point concerning the *location* output of *UI1* and the *address* output of *UI2* (composition rule 2) and that there are no conflict points on UI input elements (composition rule 3).

To solve the detected conflict in interactive mode, the developers must choose between the *location* output representation (single) of *UI1* and *address* output representation (multiple) and the operator to apply. They can also decide to create a new element and choose a new representation. In Figure 3a, they decide to keep the representation of *UI1* and to apply the union operator. In Figure 3b, they decide to keep the representation of *UI2* and to apply the selection operator.

To solve the detected conflict in automatic mode, the engine compares the representations of the UI elements that are in conflict and choose the best one. To understand the different solving methods, let consider three cases:

1. If the *professional addresses* are visualized in a list box ("multiple" representation) in *UI2* and the *personal address* in a text field ("single" representation) in *UI1*, then the engine chooses the UI element from *UI2* (the one with a multiple representation) containing the union of data from the *professional addresses* output of *S2* and from the *personal address* output of *S1*.

2. If the *professional addresses* and the *personal address* are both visualized in a list box ("multiple" representation) then the predominant UI representation is kept (*UI1* here).
3. If the *professional addresses* and the *personal address* are both visualized in labels ("single" representation) then the engine chooses to keep the two representations. The engine also informs the developers of the semantic similarities of the UI elements. The developers will then be able to place the UI elements side by side to group contact information.

In this example, no usability warning is detected by the engine because each UI element is associated with a unique label (there is no label having same names).

6 Evaluation of the Alias Approach

The Alias composition engine has been evaluated from different points of view: (i) modeling pertinence and coverage, (ii) composition engine coherence and (iii) user interaction with tool assessment. For evaluation purpose, several case studies have been implemented including the published ones: the mail/notepad scenario [22], the tour operator [23] and the human resource system (presented in this paper).

Modeling pertinence and coverage: In [23], we have shown how Alias exploits Model Driven Engineering (MDE) [24] transformations to fill the gap between the composition engine that is technological agnostic and the real world by: a) abstracting applications into the Alias formalism automatically (step 1 of the Alias process), and b) concretizing the sketch of the new UI deduced by the composition engine automatically as well as the interaction links with the FC part of the composed application (step 4 of the Alias process). The case studies have been useful to check the advantages of modeling architectural constraint as well as the possibility to abstract intrinsically different graphical toolkits such as Java SWING (desktop toolkit) and Flex (web toolkit).

Composition engine coherence: The composition rules described in this paper respect the formal properties listed in [25]. We checked the rules coherence by feeding the Prolog engine with a bunch of facts and by analyzing and comparing the inferred results with the expected ones. This work has consolidated the pivotal formalism and has been useful to check that the engine conflict detection covers a satisfying set of conflicts associated with the potential functional compositions.

User interaction with tool assessment: The framework prototype has been designed and continuously evaluated using the 9 heuristics of Nielsen and Molich [26]. Only the last heuristic "Help and documentation" is not covered as the tool is only at a stage of research prototype. Using such method is a first step toward assessment of user interaction with tool but it is not sufficient as end-users do not take part of the process in formative evaluation methods such as this one. Then, we are using a summative user-centered evaluation method called cooperative method [27] to collect application developers' feedbacks and to assess the usability of the framework and of the composition engine UI with real users. We are currently establishing evaluation protocols.

Our goal is to evaluate the ability of users (developers) to perform tasks such as: understanding conflict detection, selecting the best conflict resolution regarding a composition goal, etc. The protocols are also designed to allow us to compare what the developers expect as the UI composition result (they provide it as a diagram) and what they obtain by using the Alias framework which also provides diagrams (Figures 5, 6 and 9 are extracts of such diagrams). This cooperative evaluation is still in progress.

7 Conclusion

Alias is an approach for composing SOA applications including their user interfaces (UIs). The originality of Alias composition engine comes from the fact that UI composition is deduced from the way functional parts of applications are composed. This paper has shown: 1) how the Alias composition engine builds a first sketch of the UI by reusing and composing elements from former UIs and by maintaining the interaction links between the UI level and the functional level and 2) how it manages composition conflicts.

The goal is not to provide a UI directly usable by end-user in an ergonomic sense but to shorten UI development cycles by providing a consistent and automatic way to reuse former UIs while preserving the interaction links with their corresponding application functional part. To finalize the application, developers have to improve ergonomic aspects of the UI.

Apart from creating a UI sketch, the Alias composition engine offers additional usages. It can be used to simulate both UI composition and functional composition results prior to real developments. The composition engine feedbacks (missing UI elements, conflict points …) are clues making it possible to verify that the functional composition is such that developers expect it to be.

The experiments convince us about the pertinence of the Alias composition engine for the reuse of web application UIs developed with Flex and of desktop application UIs developed with SWING. Our short-term perspectives are to check the possibility to abstract intrinsically different SOA implementations. With the growing use of Restful services, we believe that providing abstraction transformations (step 1) for such SOA implementation is needed as well as for web service ones. Despite the fact that REST does not fit well with the message-oriented paradigm of the Web service, we claim that this has no impact on our modeling since the latter fits well with workflow as well as data flow compositional logic.

Finally, the underlying open issue is to manage divergent composition choices: between user interaction needs such as proposed in [16, 17] and functional requirements as we propose. One approach brings the user point of view and usability properties whereas the other one offers technical requirements on functionality implementation details (typing rules, semantics, etc.). Another challenge is to cope with compositions held at different time (runtime or design time) in a coherent way.

Acknowledgments. We thank the DGE M-Pub 08 2 93 0702 project for his funding.

References

1. Papazoglou, M. P., Heuvel, W. J. V. D.: Service oriented design and development methodology. Int. J. Web Eng. Technol. 2(4), pp. 412-442 (2006).
2. Szyperski C.: Component Software - Beyond Object-Oriented Programming, Addison-Wesley (1999).
3. Heineman, G., Councilln W., editors. Component-Based Software Engineering, Putting the Pieces Together. Addison-Westley, ISBN : 0-201-70485-4 (2001).
4. Occello, A., Joffroy, C., Pinna-Déry, A.-M., Renevier-Gonin, P., Riveill, M.: Meta-modeling user interfaces and services for composition considerations. In: SEDE'10, pp. 33-38, ISCA (2010).
5. Bass, L.J., Coutaz; J.: A Metamodel for the Runtime Architecture of an Interactive System. In: UIMS Tool Developers Workshop. SIGCHI Bull., vol 24(1), pp. 32-37, ACM (1992).
6. Coutaz, J.: PAC: An object oriented model for implementing user interfaces. SIGCHI Bull., 19(2), pp. 37-41 (1987).
7. Reenskaug, T. M. H.: MVC xerox parc. http://heim.ifi.uio.no/~trygver/themes/mvc/mvcindex.html (1979).
8. Marino J., Rowley M. Understanding SCA (Service Component Architecture). Addison-Wesley Professional, June 30, 360 pages (2009).
9. Objectweb Consortium: The Fractal Component Model: http://fractal.objectweb.org/ (2008).
10. Khalaf, R., Mukhi, N., Weerawarana, S.: Service-oriented composition in bpel4ws. In: WWW'03, Alternate Track Papers and Posters, Budapest, Hungary (2003).
11. Mosser, S., Blay-Fornarino, M., Riveill, M. Service Oriented Architecture Definition Using Composition of Business-Driven Fragments (workshop). In: MODSE'09, pp. 1-10, Denver, USA (2009).
12. Grundy J.C., Hosking J.G. Developing Adaptable User Interfaces for Component-based Systems. Interacting with Computers 14, 2, Elsevier Science Publishers, pp 175--194 (2002).
13. Dery, A.M., Fierstone, J.: Component model and programming: a first step to manage Human Computer Interaction Adaptation. In: Mobile HCI'03, Chittaro, L. (eds.) LNCS, vol. 2795, pp 456-460. Springer (2003).
14. Lepreux, S., Hariri, A., Rouillard, J., Tabary, J. Tarby, D., Kolski, C.: Towards Multimodal User Interfaces Composition Based on UsiXML and MBD Principles. In: HCII 2007. Jacko, J.A. (eds.) LNCS, vol. 4552, pp. 134-143. Springer, Heidelberg (2007).
15. Fujima, J., Lunzer, A., Hornbæk, K. and Tanaka, Y. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. In: UIST'04, Santa Fe, NM, pp. 175-184, (2004).
16. Gabillon, Y., Petit, M., Calvary, G. and Fiorino, H. Automated planning for userinterface composition. In Proc. of the 2nd Int. Wksp. on Semantic Models for Adaptive InteractiveSystems: SEMAIS'11, Springer HCI series, 5 pages, (2011).
17. Feldmann M., Hubsch G., Springer T., Schill A.: Improving Task-driven Software Development Approaches for Creating Service-Based Interactive Applications by Us-

ing Annotated Web Services. In Fifth International Conference on Next Generation Web Services Practices, pp. 94-97, (2009).
18. Nestler T., Feldmann M., Preußner A., Schill A.: Service Composition at the Presentation Layer using Web Service Annotations. In Proceedings of the First International Workshop on Lightweight Integration on the Web at ICWE 2009, (2009).
19. Tsai, W.T., Huang, Q., Elston J., Chen, Y.: Service-oriented user interface modeling and composition. In: ICEBE '08, pp. 21--28, IEEE Press, New York (2008).
20. Ginzburg, J., Rossi, G., Urbieta, M., Distante, D.: Transparent interface composition in Web Applications. Web Engineering, LNCS, vol. 4607, pp. 152-166. Springer, Heidelberg (2007).
21. Object Managemant Group: Unified Modeling Language Specification 2. OMG. Document formal/2009-02-02 (2009).
22. Pinna-Dery A.-M., Joffroy, C., Renevier, P., Riveill, R., Vergoni, C.: ALIAS: A Set of Abstract Languages for User Interface Assembly. In: IASTED SEA'08, pp. 77-82. ACTA Press (2008).
23. Occello A., Joffroy C., Dery-Pinna A.-M.: Experiments in Model Driven Composition of User Interfaces. In: DAIS'10, LNCS, Vol. 6115 (2010).
24. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer, 39(2), pp. 25-32, (2006).
25. Joffroy C., Caramel, B. Dery-Pinna A.-M., Riveill, M.: When the functional composition drives the user interfaces composition: process and formalization. In: EICS'11, ACM (2011).
26. Nielsen, J., and Molich, R. Heuristic evaluation of user interfaces, Proc. ACM CHI'90 Conf. (Seattle, WA, 1-5 April), pp. 249-256 (1990).
27. Monk, A., Wright, P., Haber, J., and Davenport, L.: Improving your human-computer interface: A practical technique. Prentice Hall International (UK) Ltd (1993).