Metamodeling Functional and Interactive Parts of Systems for Composition Considerations

Audrey Occello, Cédric Joffroy, Anne-Marie Déry-Pinna, Philippe Renevier-Gonin, Michel Riveill

Metamodeling Functional and Interactive Parts of Systems for Composition Considerations

Audrey Occello, Cédric Joffroy, Anne-Marie Dery-Pinna,
Philippe Renevier-Gonin, Michel Riveill
University of Nice Sophia-Antipolis/CNRS/I3S,
930, Route des Colles,
B.P. 145, F-06903 Sophia Antipolis cedex, France
{occello, joffroy, pinna, renevier, riveill}@polytech.unice.fr

Abstract: Software Composition techniques improve the reusability of systems built by composing smaller software artifacts. Composition is a challenge for Human Computer Interaction and Software Engineering communities. These communities have proposed metamodeling approaches in order to address composition at a high level and to overcome technological diversity as advocated by the Model Driven paradigm. However, such metamodels cover only one aspect of system composition. This leads to build incomplete systems. To tackle this problem, we propose a global composition approach that takes into account the functional composition choices and that maintains the interaction links between interactive and functional parts of systems. This paper presents the metamodeling on which relies the proposed composition approach.

Key words: User interface and functional composition, metamodeling, pivotal formalism.

1 Introduction

Software Composition techniques improve the reusability of systems built by composing smaller software artifacts. Composition is a challenge for the Human Computer Interaction (HCI) community and also for the Software Engineering (SE) one. In HCI [1], composition is addressed from an ergonomic and usability point of view, whereas composition in SE is focused on functional consistency. Both communities try metamodeling in order to handle composition at a high level and to overcome technological diversity as advocated by the Model Driven paradigm [2]. However, their metamodels cover only one aspect of system composition.

On the one hand, composition in HCI is based on metamodeling user interactions. For instance, the Cameleon framework [3] defines four metamodels: (i) tasks and concepts, (ii) abstract user interface (AUI), (iii) concrete user interface (CUI) and (iv) final user interface (FUI). Mappings enable traceability between these metamodels. [4] and [5] use these metamodels to compose UI descriptions but the functional part is not handled.

On the other hand, composition in SE focuses on metamodeling of the entities that hold the functionalities of a system. For instance, UML2.0 component diagram [6] or SCA (Service Component Architecture) [7] metamodels reify components or services and express functional composition (i.e. service orchestration or component assembly) as composite entities. However, UIs are not treated as services or components leading to loss in the composition process.

In both cases, the interaction links between UIs and functional parts are lost and the composition is done only on a subpart of the systems to be composed. This leads to build incomplete and unoperational systems: either systems without UIs and not able to ensure user interactions, or systems with fake UIs which are not linked to functionalities. Whenever a composition is performed, developers need to build new UIs or new functionalities from scratch and need to recover the interaction links; which is time consuming and error prone.

Some recent work aims to reconcile these both aspects. For example, Servface [8], a European project, enhances service descriptions with UI annotations to generate a high quality UI

linked to services. The generation process is based on the refinement of the Cameleon models mentioned previously. Servface composition relies on a task tree based description (service operations are bound to system tasks). In practice, this approach is not actually usable, principally because: 1) most of applications do not provide task tree descriptions and 2) UI annotations have to be written to generate new UI instead of reusing existing UI.

To propose a more pragmatic UI composition approach, we extract the task sequences in UI interactions from dataflow and workflow information present in functional composition. The originalities of our proposal are also to reuse existing UI and to maintain the interaction links with the functional part of the systems. To simplify the vocabulary, we focus on service oriented applications in the rest of the paper. Hence, the functional parts of systems are referred as services.

The remainder of the paper is organized as follows: Section 2 presents a case study that illustrates the proposed approach. Section 3 details the metamodeling proposition used to describe UIs and services as well as their compositions. Finally, Section 4 presents our conclusions.

2 Alias Approach Outlines Illustrated by a Tour Operator Case Study

In this section, we illustrate our composition approach called Alias thanks to a tour operator case study. The only requirement of the Alias approach is to respect a separation of concerns between the different parts of an application. More precisely, UIs and services need to be clearly recognizable as well as the interaction links between the two parts (i.e. triggering a given operation on event handling is considered as an interaction link between the UI and the service). The Alias composition process deduces a UI for an application *A* as a function of: 1) the service composition choices that lead to produce *A*; and 2) the interaction links between the services that compose *A* and their corresponding UIs. Thus, the new UI is built from several parts of former UIs.

We exemplify the Alias approach with a Hotel Booking and Flight Reservation composition scenario. Suppose we want to build a new service that enables a user to book a hotel and a flight simultaneously as would happen in a tour operator company. With such a service, the user would plan a trip faster. To obtain this new service, we compose two existing services: a Hotel Booking service and a Flight Reservation service. To illustrate our proposal, we only focus on the availability checking operation of these services and the corresponding UIs. Extensions of the example can be found on the Alias website (http://users.polytech.unice.fr/~joffroy/ALIAS/).

2.1 Service Description

The Flight Reservation Service provides two operations (Figure 1(a)): (i) *getAvailableFlights* returns a list of flights, (ii) *reserveAFlight* makes a flight reservation. To view available flights, a user chooses a country and a city to select departure/destination airports and departure/return dates. This action calls *getAvailableFlights*.

Similarly, the Hotel Booking service provides two operations (Figure 1(b)): (i) *getAvailableHotels* returns a list of available hotels for a given quadruplet (country, city, check-in and check-out dates) and (ii) *bookARoom* books a room. To view available hotels, a user chooses a country, a city and check-in/check-out dates. This action calls *getAvailableHotels*.

2.2 Composition Scenario

The strength of Alias is to prevent developers from implementing UI from scratch. Instead, the developer focuses on composing business services as usual. The UI of the resulting composite application is deduced from: 1) the interaction links between each UI and its corresponding service and 2) the service composition choices. The composition result, at the UI level, depends on the choices that are made by the developer during service composition.

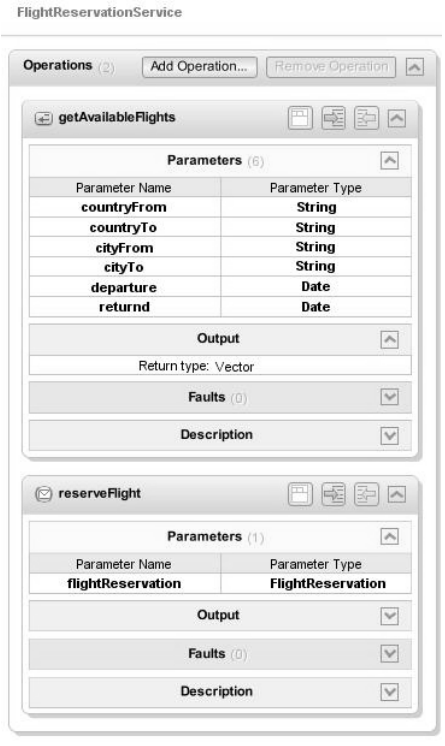


Figure 1(a): Flight Reservation service

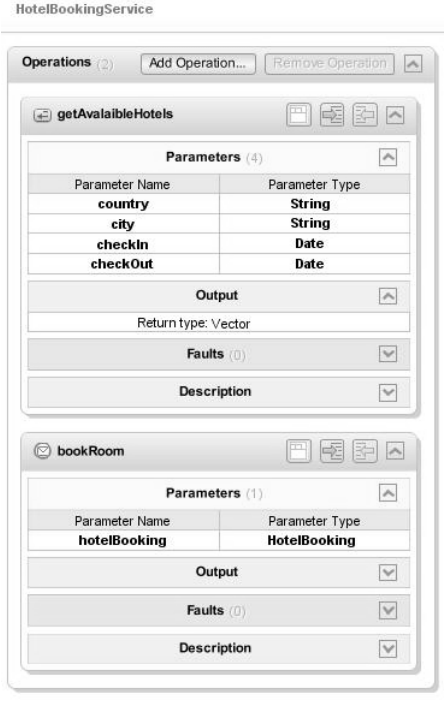


Figure 1(b): Hotel Booking service

In this case study, the two services are composed in the orchestration depicted in Figure 2. The two operations are called in parallel in the workflow and some input data are merged (Figure 3). Other service composition choices may lead to call operations in sequence or to keep all inputs (resp. outputs). Figures 1, 2 and 3 are screenshots from Netbeans IDE.

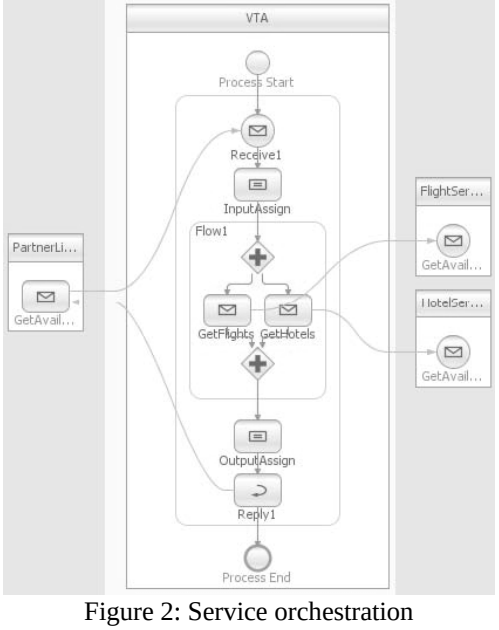


Figure 2: Service orchestration

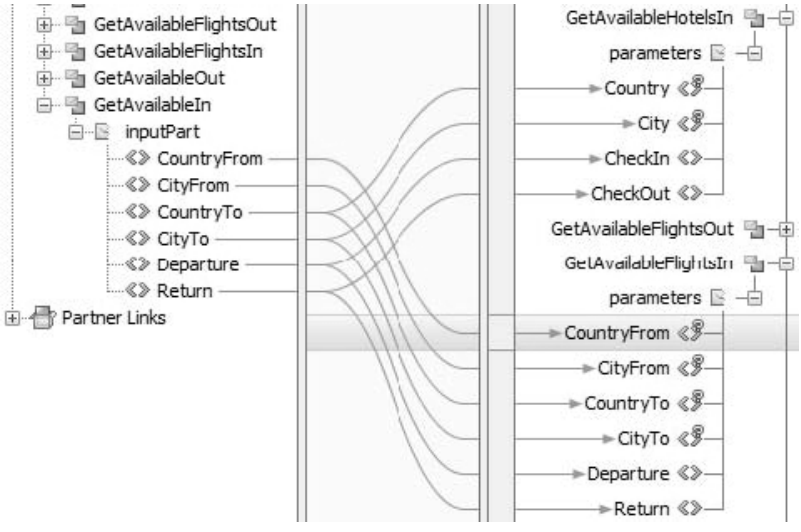
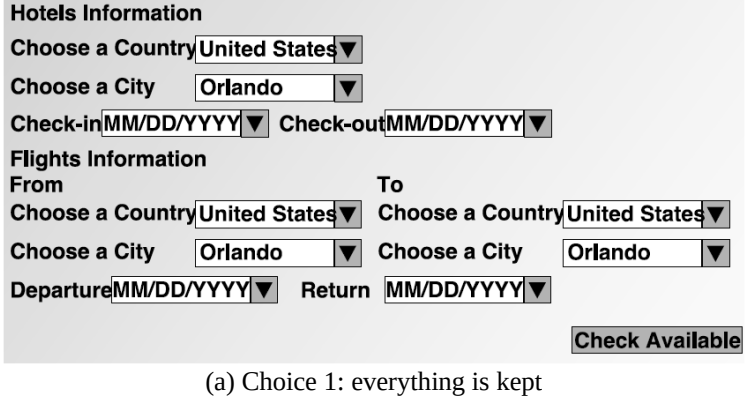


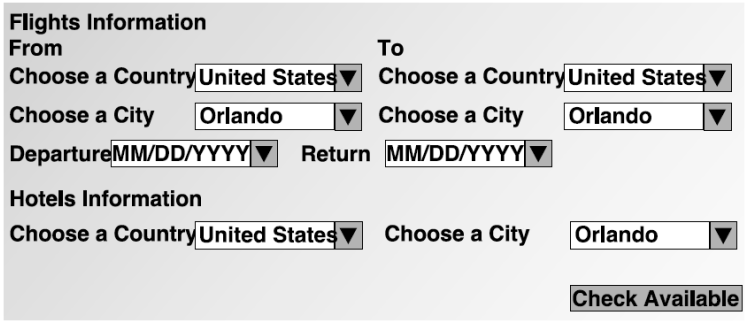
Figure 3: Input merging in service composition

Alias does not use an opportunistic algorithm which takes elements from anywhere, trying to compose them in a specific way. Instead, the composition is expressed as a logical expression using first order logic. Actually, the composition engine's algorithm (described as PROLOG facts) focuses on merge rules in order to decide whether to merge UI elements or not and then, to remove duplicate elements for equivalent functionalities. The choice of merging UI elements is deduced from the choice of merging inputs (resp. outputs) of services. The composition rules and their formalization are detailed in [9].

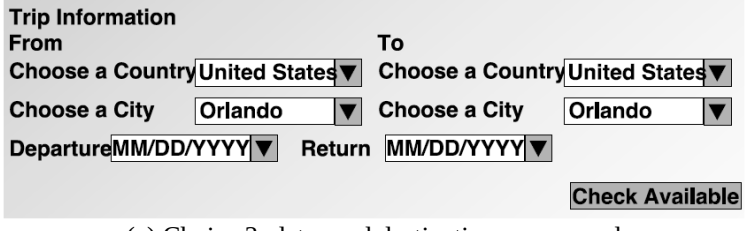
In this case study, keeping all inputs from both services leads to a resulting UI, where all UI inputs are kept (illustrated in figure 4(a)). Keeping only one date input for the departure and for the return in the service composition leads to a UI result where date inputs are merged (Fig. 4(b) - the user fills departure and return dates only once). Here, the last composition alternative reflects the merging choices that have been done on the service composition (Figure 3). This last composition alternative consists in merging date inputs as previously and also destination inputs (Fig. 4(c)): the flight destination is the same for the hotel and the user fills departure /return dates and city only once.



(a) Choice 1: everything is kept



(b) Choice 2: dates are merged



(c) Choice 3: dates and destination are merged

Figure 4: Possible UI composition results

To make such composition as generic as possible, UIs, services and the way they are composed need to be explicited in a pivotal formalism which is presented in next section.

3 Metamodels for UI Composition

In previous work [10], we defined three languages in order to describe UIs at different levels of abstraction: *ALIAS-Behavior* for modeling UI elements at a very high level, *ALIAS-Structure* for modeling more concrete aspects of UI structure and *ALIAS-Layout* for modeling UI layout. The main goal of this set of languages was to compose heterogeneous UI directly.

In recent work, we moved from a pure UI composition to a UI composition deduced from service composition. As a matter of fact, we have to manage information from both UIs and services. We could have distinguished the formalism describing UIs from the one describing services. However, the degree of abstraction that we chose allows for manipulating UI and services in the same way (we do not need to take into account information about UI concrete structure and UI layout in this kind of composition). This point facilitates the reasoning on composition and simplifies the formalism.

In the main, modeling is always done with a particular goal; then, we split our formalism design into two metamodels, each one having the most suitable representation to achieve its goal easily. The first one deals with service and UI information extraction and exchange (see Section 3.1) and the second one deals with the composition itself (Section 3.2). Since their respective structures and their goals differ, merging these two metamodels would have obliged us to privilege one representation in an unified metamodel and would have led to weaken one of them. The metamodels we described in this article are at the AUI level of the Cameleon framework [3]: *AliasExchange* is a subset of the AUI metamodel, *AliasCompose* reifies additional information about composition/interaction links.

3.1 AliasExchange Metamodel

The *AliasExchange* metamodel characterizes UIs and services. This metamodel describes information that needs to be reified for a UI concerns: the data inputs independently of the widget chosen to retrieve data (textfields, lists, trees, etc), the data outputs independently of the widget chosen to display such data (labels, etc) and action triggers (user interactions) independently of the widget chosen to trigger actions (buttons, menu items, etc). Also, this metamodel describes information that needs to be reified for a service concerns , essentially the signature of the operations implemented by a service: input parameters, output parameters and the name of the operations. Only functionalities related to user tasks (functionalities that are called as a result from a user interaction and/or functionalities whose results are presented to users) are reified. As a result, we can identify an isomorphism between the two sets of information: 1) UI data inputs correspond to service operation parameters, 2) service operation results correspond to UI data outputs and 3) interactions with users are located in UI action elements and service operation calls. Thereby a unique representation for both UI and services is possible.

To sum-up, the metamodel (Fig. 5) defines the *Entity* class, representing UIs and services. It is composed of the *Element* class and the *Input*, *Output* and *Action* classes inheriting from the *Element* class. UI elements and service operations are characterized by an *id*, a *name* and a *type* (except for actions). Each *Action* element is associated with *Input* and *Output* elements which may be the parameters/return of an operation or the input/output widgets of a UI.

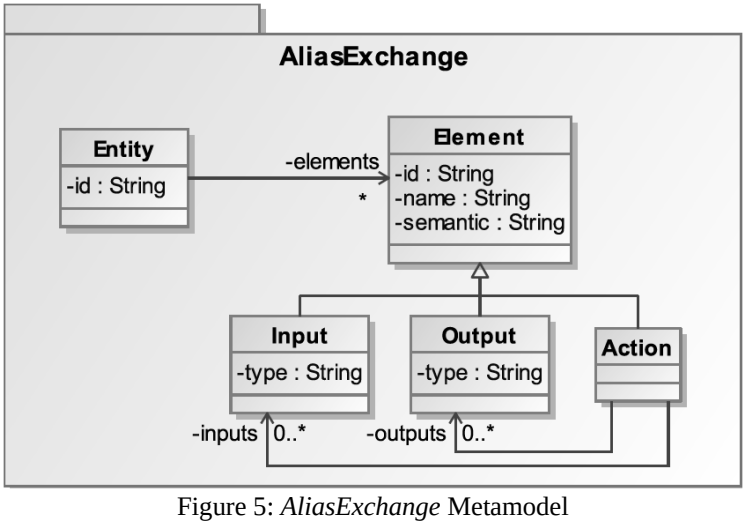


Figure 5: *AliasExchange* Metamodel

In this metamodel, each entity is considered individually. We reify neither the interactions between the UI and the service nor the interactions between services. The semantic attribute is a composition parameter determining if two UI elements are equivalent or belong to a same family of information for the user (i.e. contact information). Currently, it is filled manually and enables us to validate the composition algorithm and to compare different merging alternatives. Ultimately, an ontological model should be used instead of such an attribute.

In the tour operator scenario, the Flight Reservation service and its UI are represented as *AliasExchange* models which highlight the isomorphism between the two sets of information. In Figure 6, we present the user interface of the Flight Reservation service for the availability checking. There are six inputs: departure and destination country names, departure and destination city names, check-in and check-out dates). Also, there is one output: a list of fights; and one action: users query for available fights.

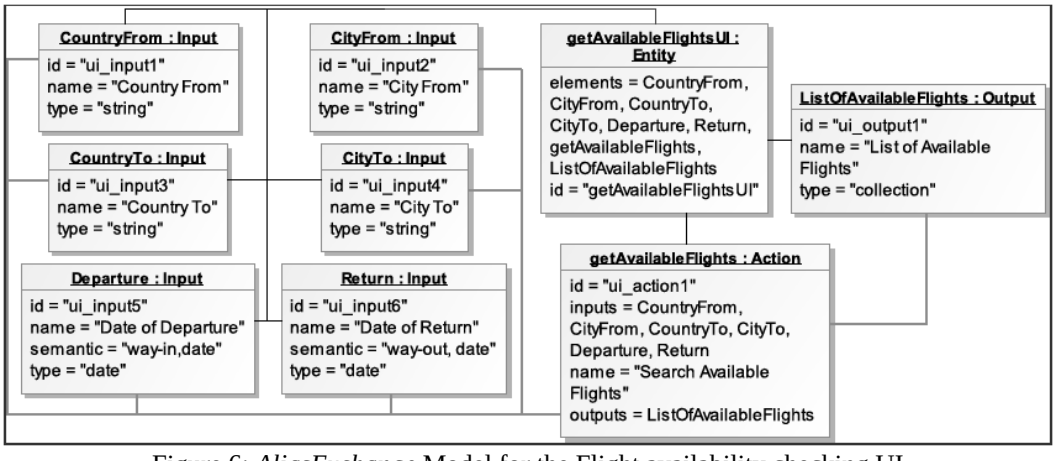


Figure 6: *AliasExchange* Model for the Flight availability checking UI

Figure 7 illustrates the Flight Reservation service. We recognize the data shared with the UI component for availability checking as well as the data relative to the other UI (this UI is not depicted in this paper). In this service, inputs are operation parameters, outputs are operation results and actions are operations such as *getAvailableFlights*. Types are not shown to avoid overloading the figure.

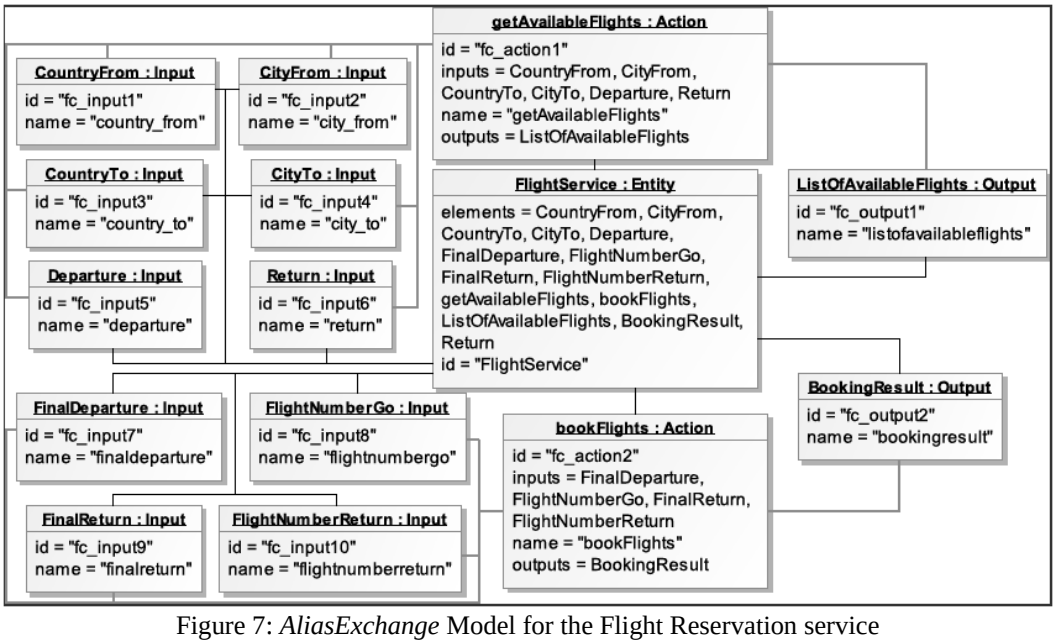


Figure 7: *AliasExchange* Model for the Flight Reservation service

This metamodel facilitates the service and UI descriptions exchange between service compositors and UI designers; specifying the new UI as a function of the composition deduction results.

3.2 AliasCompose Metamodel

The *AliasCompose* metamodel (Fig. 8) describes the interactions involved in compositions of a service and its UIs, and also in compositions of several services. To express the interactions, we use two different binding types: one for dataflows and one for control flows (events).

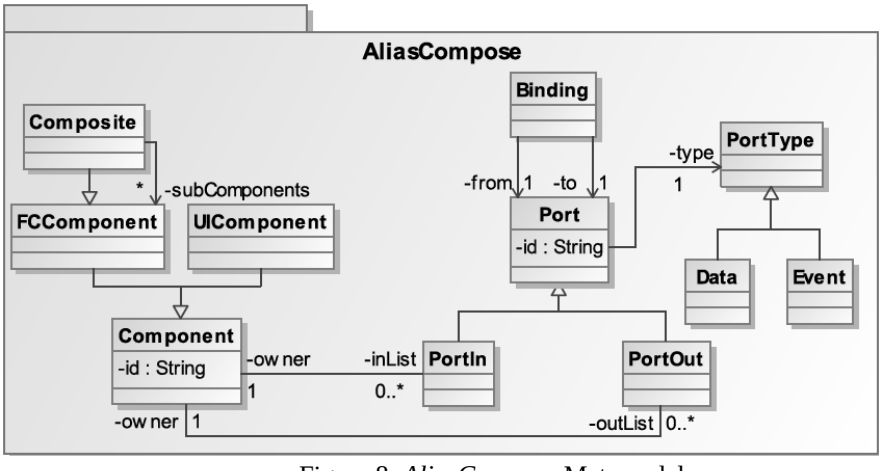


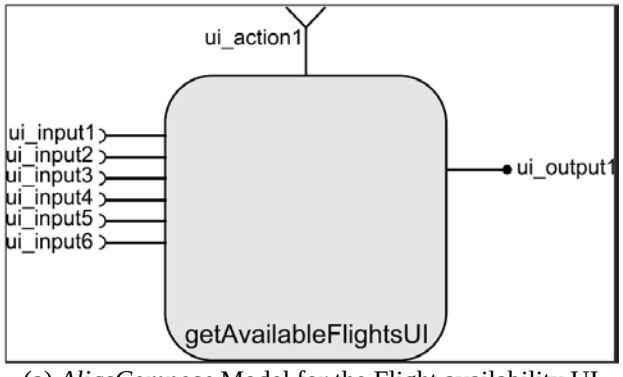
Figure 8: *AliasCompose* Metamodel

This metamodel is related to the component metamodels (UML2.0 component diagram [6] or SCA [7]), where UIs and services are represented as components with ports. However, in our metamodel the granularity of the port is finer (at data or operation level, not at programming interface level). Therefore, as this is a defacto standard to express bindings, we adopt the component metaphor.

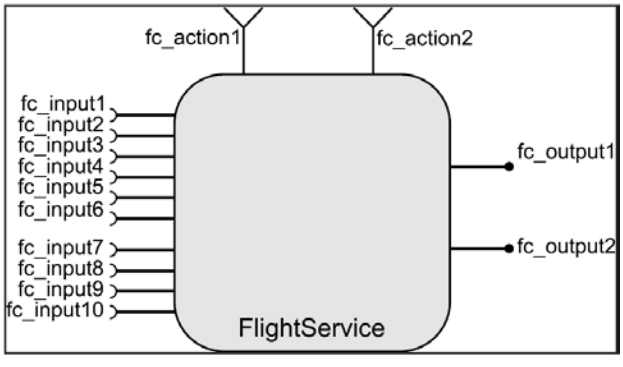
AliasCompose shares information about each individual UI and service with *AliasExchange*. However, not all information is kept here as none precise definition of each individual service/UI is needed to express the interaction and composition links. In addition, *AliasCompose* contains OCL rules that constrain the *Binding* class. For example, bindings between inputs link from a *UIComponent* to a *FCComponent* or from a *Composite* to a *FCComponent*:

```
context Binding inv DataLinkII : (self.from.oclIsTypeOf(PortIn) and
self.from.oclIsTypeOf(PortIn) and self.from.porttype.oclIsTypeOf(Data)
and self.to.porttype.oclIsTypeOf(Data)) implies
((self.from.oclAsType(PortIn).component.oclIsTypeOf(UIComponent) and
self.to.oclAsType(PortIn).component.oclIsTypeOf(FCComponent)) or
(self.from.oclAsType(PortIn).component.oclIsTypeOf(Composite) and
self.to.oclAsType(PortIn).component.oclIsTypeOf(FCComponent)))
```

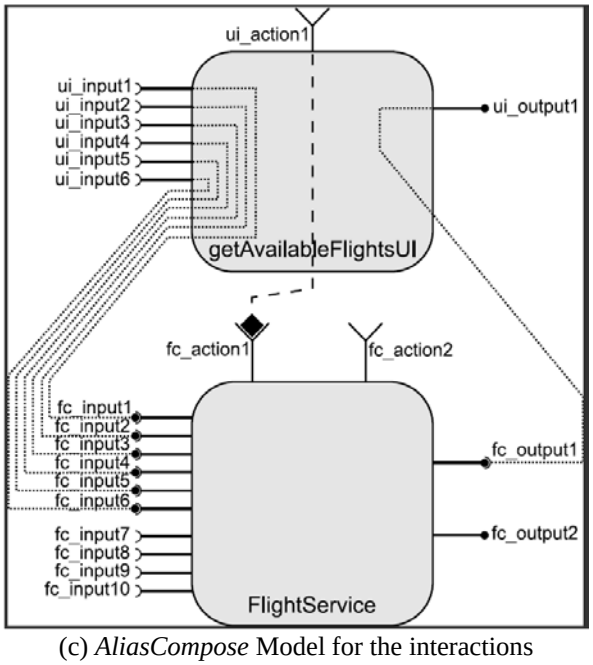
The availability checking IU and the Flight service are represented as two *AliasCompose* models (Fig. 9(a) and 9(b)). Each box represents a component. Data inputs (resp. outputs) are represented at the left (resp. right) side of the box. Triggers are on the top and bottom sides of the box. To express interactions between the UI component and the service component, a new *AliasCompose* model is defined (Fig. 9(c)). The bindings between ports are added (see the Binding class in Fig. 8).



(a) *AliasCompose* Model for the Flight availability UI



(b) *AliasCompose* Model for the Flight service



(c) *AliasCompose* Model for the interactions

Figure 9: *AliasCompose* models: UI and service for Flight reservation

The *Composite* class (see figure 8) reifies the service composition as a new component. It expresses which ports are kept and which ones are left (we have chosen to merge city, country, departure, return inputs as presented in section 2), and also it helps in deducing UI element merging. Figure 10 shows the *AliasCompose* model corresponding to the composition of the Hotel Booking and the Flight Reservation services. If the service composition changes, the result in the *AliasCompose* model changes accordingly.

According to the ports selection (ports kept and left), the composition engine deduces which UI elements may be merged in the resulting UI. Figure 11 shows the UI deduced from the service composition model and from the UI/service interaction models. It illustrates also how this UI is linked to the new service model (composite).

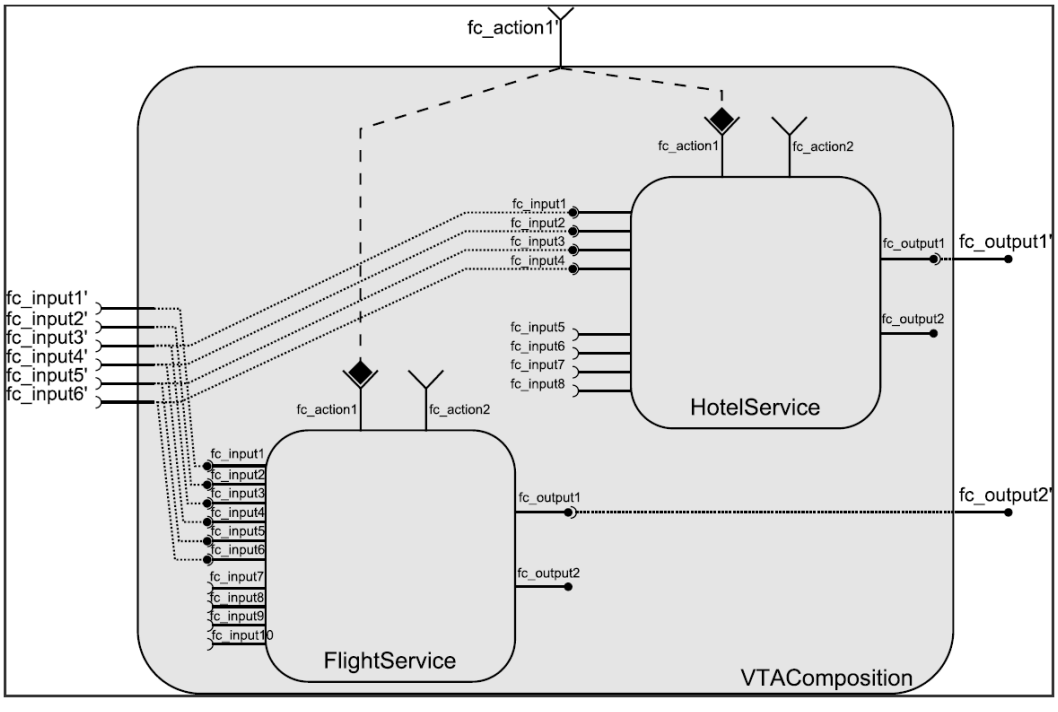


Figure 10: *AliasCompose* model: service composition in the tour operator case study

Reasoning at an abstract level, it is not enough to have an operational composition. Composition inputs have to be retrieved from real services/UIs. The resulting UI has to be concretized as a real UI and Prolog facts have to be fed automatically. To obtain an end-to-end process, we use a transformation chain.

In [11], we have shown how Alias exploits Model Driven Engineering (MDE) [2] transformations in order to: 1) abstract applications into the Alias formalism automatically, and 2) generate code for the sketch of the UI provided automatically by the composition engine and for the interaction links with the composed application functional part). These works have consolidated the pivotal formalism.

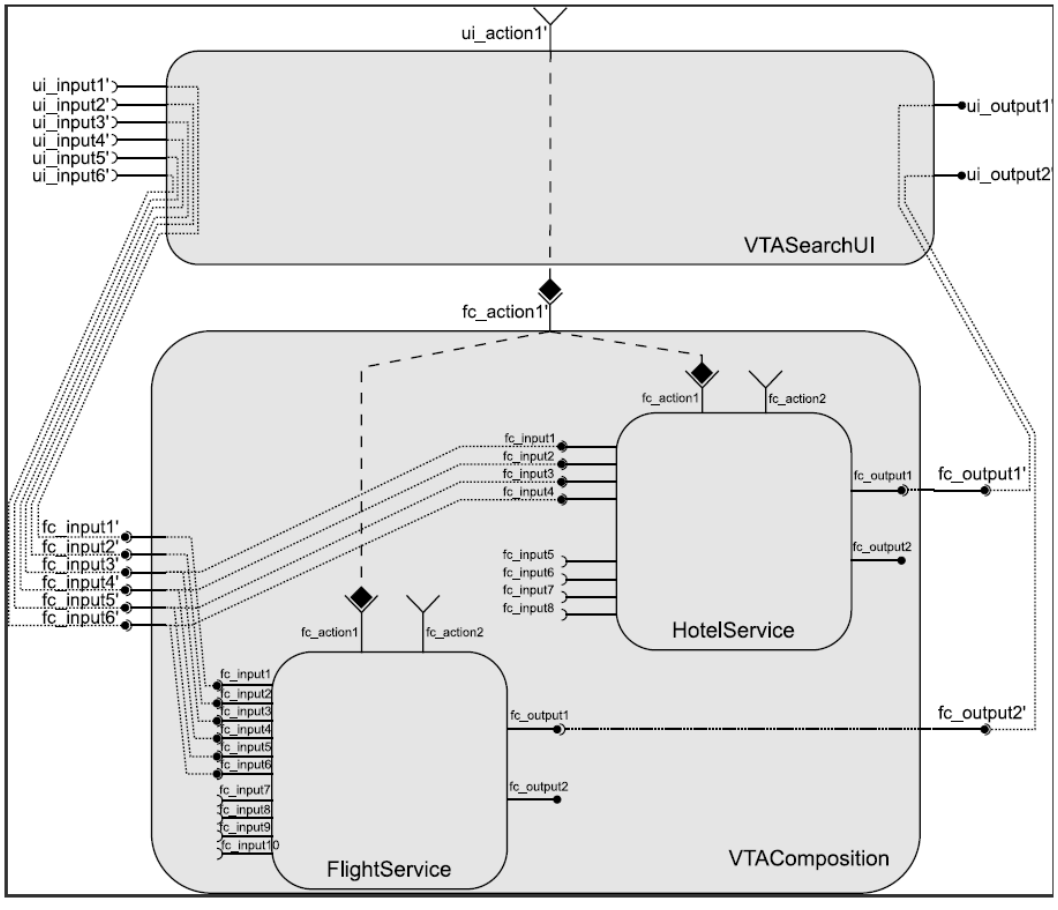


Figure 11: *AliasCompose* model: resulting UI for the tour operator application

4 Conclusion

In this paper, we have presented a logical approach for composing UIs called *Alias*. Its originality comes from the fact that the UI composition is deduced from the functional composition. In order to decouple the composition from technological spaces, the proposed metamodeling approach raises the abstraction level of descriptions for interactive and functional parts of systems.

The Alias approach has been evaluated using several criteria as: the Alias metamodels validation, the cost estimation of the architectural constraints, the set of supported graphical toolkits, or the component platforms that can be reached.

As well, several case studies have been implemented including the published ones: the mail/notepad scenario [10], the tour operator (mentioned in this paper) and the human resource system [9]. These case studies have been useful to verify the pertinence of the Alias metamodels. These case studies involve form-based UI. In order to evaluate the Alias metamodels over more sophisticated UI, we are currently working on the handling of tactile and vocal applications with multimodal interactions.

The Alias composition engine builds a first sketch of the final UI by reusing elements of former UI. Our underlying goal is to reach a final UI with ergonomic properties. This means that

details concerning UI concrete structure and layout choices need to be reintroduced. For this, we plan to integrate work concerning plasticity in the Human Computer Interaction community. Such a work is also based on models at different abstraction levels: The UI resulting from the Alias composition, expressed as an Alias model, needs to be transformed into plasticity models (such as Teresa [12]) using annotations to describe UI elements more precisely with widget-specific characteristics (lists, check-boxes, buttons, menu items . . .) and the layout of UI elements.

Acknowledgements

This work was partially funded by the DGE M-Pub 08 2 93 0702 project.

References

[1] D.L. Scapin and J.M.C. Bastien. Ergonomic criteria for evaluating the ergonomic quality of interactive systems. Behaviour & Information Technology, 16(4):220–231, 1997.

[2] D. C. Schmidt. Model-Driven Engineering. IEEE Computer, 39(2):25–32, 2006.

[3] J.S Sottet, G. Calvary, J.M. Favre, and Jolle Coutaz. Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: Mega-UI. 2007.

[4] S. Lepreux, A. Hariri, J. Rouillard, D. Tabary, J. Tarby, and C. Kolski. Towards Multimodal User Interfaces Composition Based on UsiXML and MBD Principles. Lecture Notes in Computer Science, 4552:134, 2007.

[5] A.M. Pinna-Déry and J. Fierstone. Component model and programming: a first step to manage Human Computer Interaction Adaptation. In Mobile HCI, volume LNCS 2795, pages 456–460, 2003. Springer Verlag.

[6] The Object Managemant Group. UML Specification 2. OMG Document formal/2009 -02-02, 2009.

[7] J. Marino and M. Rowley. Understanding SCA (Service Component Architecture). Addison-Wesley Professional, 2009.

[8] Servface Project. Service annotation for user interface composition (7th Framework European Programme Project). http://www.servface.org, 2008.

[9] C. Joffroy, A.M. Dery-Pinna, B. Caramel, M. Riveill: When the functional composition drives the user interfaces composition: process and formalization. In ACM, Engineering Interactive.

[10] A.M. Pinna-Déry, C. Joffroy, P. Renevier, M. Riveill, and C. Vergoni. ALIAS: A Set of Abstract Languages for User Interface Assembly. In SEA'08, pages 77–82, USA, 2008. IASTED, ACTA Press.

[11] A. Occello, C. Joffroy, A.M. Pinna-Déry, P. Renevier, and M. Riveill. Experiments in Model Driven Composition of User Interfaces. In DAIS'10, volume 6115 of LNCS. Springer-Verlag, 2010.

[12] G. Mori, F. Paterno, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. IEEE Transactions on Software Engineering, 30(8):507–520, 2004.