Experiments in Model Driven Composition of User Interfaces

Audrey Occello, Cedric Joffroy, and Anne-Marie Dery-Pinna

Université de Nice Sophia-Antipolis, Polytech'Nice Sophia,
930, Route des Colles, B.P. 145,
F-06903 Sophia Antipolis cedex, France

Abstract. Reusing and composing pieces of software is a common practice in software engineering. However, reusing the user interfaces that come with software systems is still an ongoing work. The Alias framework helps developers to reuse and compose user interfaces according to the way they are composing new systems from smaller units as a mean of speeding up the design process. In this paper we describe how we rely on Model Driven Engineering to operationalize our composition process.

Keywords: User interface composition, metamodeling, transformations.

1 Introduction

Software Composition is about reuse of software artifacts in order to construct larger systems from smaller ones such as with Service Oriented Architecture (SOA) [1] or Component-Based Software Engineering (CBSE) [2] paradigms. Techniques evolve to ever improve the reusability, customizability and maintainability of such composed systems. However composition is often focused on the functional part of a system and not on its interactive part. Hence, the User Interface (UI) often has to be built from scratch each time a new system is composed from a set of services or components for example.

Based on the hypothesis that each service comes along with a UI, we propose to exploit the relationships between them to deduce the UI of the application resulting from a composition of services. The Alias framework[1] builds a UI for an application A as a function of: 1) the way services are composed to form A; and 2) the interactions between such services and their corresponding UIs.

The originality of Alias is to reason at the Abstract User Interfaces (AUI) level which simplifies the composition algorithm and makes it reusable and oblivious to heterogeneity:

– algorithm simplification: the AUI level enables us to focus on the composition rules without burdening ourselves with widget type and style,
– reuse: the same algorithm can be used to deduce composition of Swing UIs, Flex UIs, Ajax UIs and so on,
– obliviousness to heterogeneity: we can deduce the composition of UIs written in different languages, as is done in the plasticity research area [3].

[1] `http://users.polytech.unice.fr/~joffroy/ALIAS/`

F. Eliassen and R. Kapitza (Eds.): DAIS 2010, LNCS 6115, pp. 98–111, 2010.
© IFIP International Federation for Information Processing 2010

Given these facts, we believe in the legitimacy of adopting a Model Driven Engineering (MDE) [4] approach. In this paper, we describe how we experiment with MDE to operationalize our composition process. As there are various ways of implementing MDE, we also discuss how we deal with modeling and tool choices and the questions our experiment raises.

The remainder of this paper is organized as follows. Section 2 illustrates the Alias composition process using a tour operator scenario. Section 3 introduces the metamodels on which the framework is based to describe user interfaces and services as well as their compositions. Section 4 gives an overview of the transformations that operationalize the Alias composition process. Section 5 compares Alias with related work. Section 6 concludes.

2 Alias Composition Process Overview

The strength of the Alias approach is to prevent the developer from implementing user interfaces from scratch. Instead, the developer focuses on composing business components or services as usual. Then, the UI of the resulting composite application made of services is deduced from: 1) the interaction links between each user interface and its corresponding service and 2) the way services are composed to form the composite application.

The only requirement to use the Alias framework is to respect a separation of concerns between the different elements that compose an application: the UI part and the service part need to be clearly identifiable as well as the interaction links between the two parts (triggering a given operation on event handling is considered as an interaction link between the UI and the service).

Section 2.1 presents a scenario to illustrate the proposal. Section 2.2 gives some details about the composition engine and its role in the overall process.

2.1 Tour Operator Scenario

We illustrate the Alias approach with a Hotel Booking and Flight Reservation composition scenario. We want to reuse these two services and their corresponding user interfaces in order to build a new service that enables users to book a hotel and a flight simultaneously, as would happen in a tour operator company. With such a service, the user would be able to plan a trip faster. To illustrate our proposal, we only focus on the search part of these services. Extensions of the example can be found on the Alias website.

The Hotel Booking service. This service handles two operations: (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 in a hotel. To use this service, there are different user interfaces (as proposed by `http://www.travelocity.com/Hotels`). We only focus on the UI that enables the user to check the availability of hotels. To view available hotels, a user has to follow the steps above: 1) choose a country, a city and check-in/check-out dates; 2) search for available hotels (`getAvailableHotels` operation call).

The Flight Reservation service. This service handles two operations: (i) `getAvailableFlights` returns a list of flights and (ii) `reserveAFlight` makes the reservation of the flight. To use this service, there are different user interfaces (as proposed by `http://www.airfrance.us/`). We focus only on the UI that checks available flights). To view available flights, a user has to follow the steps above: 1) choose a country and a city to select a departure and a destination airport, a departure and a return date; 2) search for available flights (`getAvailableFlights` operation call).

2.2 Alias Framework Steps and Role of the Composition Engine

The composition engine aims at deducing which elements of the existing UIs to keep, which ones to leave and what to do in case of duplicated elements in the UI corresponding to the service composition. The expression of the resulting UI structure as a composition of the existing UI is not written by the developer. The composition rules that give the resulting UI structure are generated by the engine as a function of composition inputs (the interaction links between each user interface and its corresponding service and the way services are composed).

Alias uses first order logic to generate such composition rules. The composition engine is described as PROLOG predicates, the composition inputs are generated as PROLOG facts and the composition rules to generate the resulting UI are obtained by inference. We do not detail the composition engine further as it is not in the scope of this paper.

Using the Alias framework can be divided in 5 steps and implies the manipulation of services, UIs and compositions at two levels of representation: a concrete level that corresponds to the Platform Specific Model (PSM) of the MDA layer modeling stack [5] and an abstract level that corresponds to the Platform Independant Model (PIM).

1. The developer has to select the services to compose (Hotel Booking service and a Flight Reservation service in the tour operator scenario);
2. The framework collects information about each service and their UIs to create abstract representations;
3. The developer makes explicit the composition links between the different services of the composite application and the interaction links between services and UIs at the abstract representation level;
4. The composition engine computes the resulting user interface as a set of element reused from the existing UIs abstract representation;
5. The information of the resulting UI abstract representation is used to generate a first sketch of the Concrete User Interface (CUI) at the platform level.

Steps 1, 2 and 5 imply being capable to obtain and interpret differents PSMs corresponding to various UI description languages (Flex, XUL, Swing, etc), service description languages (SCA, OSGI, WS-*, etc) and composition formalisms (BPEL orchestration, component assembly, etc). Steps 2, 3 and 4 are necessary to perform the composition in a generic way: UIs, services and the way they are composed need to be explicit in a pivotal formalism, the PIM, which is presented in section 3. An overview of the model transformations that operationalize this process is described in section 4.

3 Metamodels Involved in the UI Composition Process

In previous work [6], 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 the UI structure and ALIAS-Layout for modeling the UI layout. The main goal of this set of languages was to compose heterogeneous UIs directly: we experimented with various composition algorithms.

This first step led us to the conclusion that we do not need to take into account all these aspects in the composition rules. Composing the style or the layout is not pertinent when considering heterogeneous UIs: for the composite UI to look coherent, it is necessary to keep the style and the layout of only one of the UIs to be composed. Such information is reintroduced after the composition during the transformation to the concrete UI level. Ongoing work is focused on ALIAS-Behavior, the pivotal formalism upon which the composition reasoning is done. This section presents how we metamodeled this pivotal formalism.

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 not only about UIs but also about 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 UIs and services in the same way (see section 3.1). This point facilitates the reasoning about composition and simplifies the formalism.

We applied separation of concerns and split our formalism design into two metamodels (PIMs), each one having the most suitable representation to achieve its goal easily. A first one deals with service and UI information extraction and exchange (section 3.1) and the second one deals with the composition itself (section 3.2). As their structures goals differ, merging the two would have led us to privilege one representation in such a unified metamodel, making the other less efficient. Moreover, there is only a small subset of information in common between the two metamodels.

3.1 AliasExchange Metamodel

The AliasExchange metamodel represents both UIs and services. Information that needs to be reified for a UI concerns: the data inputs independently of the widget chosen to retrieve this data (text fields, lists, trees, etc), the data outputs again independently of the widget chosen to display this data (labels, etc) and action triggers (user interactions) independently of the widget chosen to trigger actions (buttons, menu items, etc). 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 operation. Then we can identify an isomorphism between the two sets of information and that is why a unique representation for both UI and services is possible: 1) UI data inputs correspond to service operation parameters, 2) service operation

results correspond to UI data outputs, 3) interactions with the user are located in UI *action* elements and service operation calls.

To sum-up, the metamodel (Fig. 1) defines the **Entity** class, representing UIs and services which are made of a set of **AliasElement** and the **Input**, **Output** and **Action** classes which inherit from the **Element** class describing UI elements and service operations through an *id*, a *name* and a *type*. Each **Action** element is associated with **Input** and **Output** elements in order to find the parameters and return of an operation and to associate user interactions to input and output widgets. In this metamodel, each entity is considered individually: we reify neither the interactions between the UI and the service nor the interactions between services.

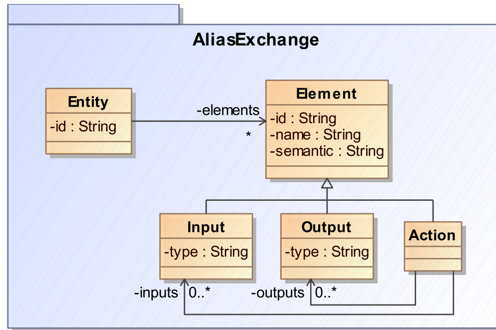

Fig. 1. UML Class diagram of the AliasExchange Metamodel

The *semantic* attribute is a parameter of the composition algorithm used in case of conflicts. It helps in deciding to merge or group UI elements if they are equivalent or if they belong to the same family of information for the user (e.g. two UI elements that denote contact information). It also makes it possible to prevent the merging when UI elements own the same structure (name, type) but differ in their semantics (e.g. hotel check-in and flight departure are dates but the first one means “way-out” and the second one “way-in”).

Currently, this attribute corresponds to some key words (e.g. “way-in”, “way-out”, “contact information”) and is filled manually in order to validate the composition algorithm and to compare different merging alternatives. Ultimately, we plan to decorate AliasExchange models with ontology annotations to enhance the conflict detections. The challenge is then to decorate the models automatically. As a proof of concept, we plan to extract such information from Web Services ontology languages such as OWL-S (`http://www.w3.org/Submission/OWL-S/`).

Figure 2 shows the AliasExchange model for the flight availability checking UI of the tour operator scenario. There are six inputs (names of departure/destination country and city, check-in and check-out dates), one output (a list of available flights) and one action (users search for flights).

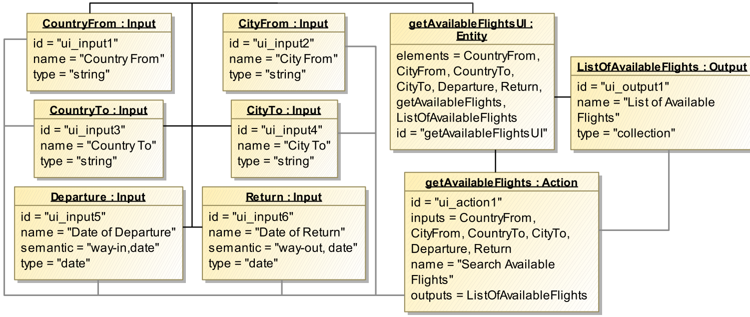

Fig. 2. Model for the user interface of the Flight Reservation service

Figure 3 represents the Flight Reservation service and highlights the isomorphism between the two sets of information: We recognize the data shared with the UI component for availibility checking as well as the data relative to the other UI not depicted in this paper. Inputs are operation parameters, outputs are operation results and actions are operations such as `getAvailableFlights`. Types are not shown to avoid overloading the figure.

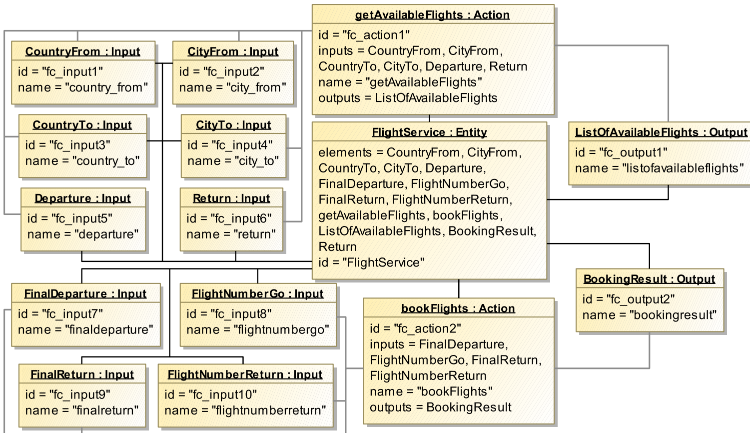

Fig. 3. Model for the Flight Reservation service

This metamodel eases the exchange of service and UI descriptions between developers, composing services, and UI designers, creating the concrete UI in function of the abstract UI deduced from such composition.

3.2 AliasCompose Metamodel

The AliasCompose metamodel represents interactions involved in compositions between a service and its UIs and between several services. To express the interactions, we use two different binding types: data links and event links. Event links represent control flows (between two operations or between an operation and the UI element that triggers the call). Data links represent dataflows (between UI elements and operation or between operations).

The metamodel (Fig. 4) looks similar to component metamodels such as UML2.0 component diagram [7] or SCA [8] because UIs and services are represented as components with ports. However the granularity of our port is finer: at the data or operation level not at the programming interface level. We adopt the component metaphor as this has become a defacto standard to express bindings. AliasCompose shares some information about each individual UI and service with AliasExchange. However, it does not keep neither the name and type of inputs/outputs nor the relationships between actions and relative input and output elements as it does not need a precise definition of each individual service/UI to express the interaction and composition links.

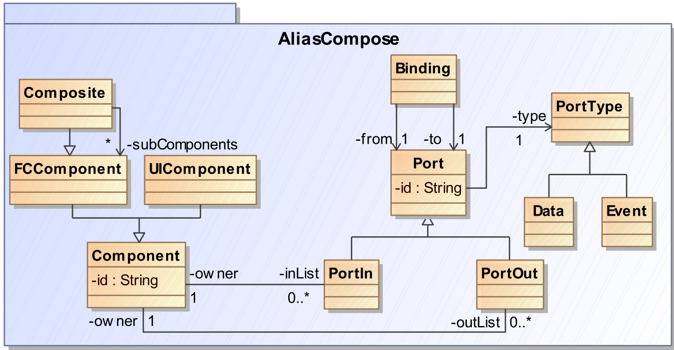

Fig. 4. UML Class diagram of the AliasCompose Metamodel

From AliasExchange to AliasCompose: The AliasCompose models for the Flight Reservation service and the flight availability checking UI are obtained from the AliasExchange models illustrated in section 3.1. The service and the UI are components represented as boxes; inputs and outputs are represented at the left and right sides of the box; and triggers are represented on the top side of the box. The AliasCompose model for the service is depicted in the lower part of Figure 5 and the AliasCompose model for its UI in the upper part of this figure.

First refinement step: To express the interaction links between the service and the UI, the two corresponding models need to be refined as a third one where bindings (see the *Binding* class in figure 4) between component ports are added. Such bindings are illustrated in Figure 5 as dashed lines. The overall figure depicts the AliasCompose model corresponding to this refinement.

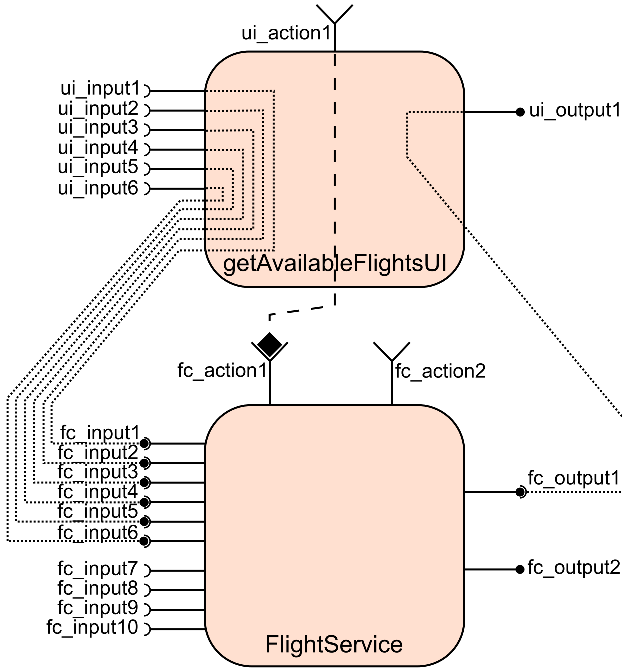

Fig. 5. AliasCompose model showing the interactions between the UI and the service

Second refinement step: The *Composite* class (see figure 4) is used to reify the result of the service composition as a new component. It expresses which ports are kept and which ones are left and makes it possible to deduce which UI elements may be merged in the resulting UI. For example, the lower part of figure 6 shows the AliasCompose model resulting from the refinement of the two AliasCompose models corresponding to the Hotel Booking service and the Flight Reservation service with city input merging.

Results of the composition engine: The upper part of Figure 6 shows the AliasCompose model for the UI computed by the composition engine. The bindings between the upper part and the lower part of the figure describe the interactions between the resulting UI and the composition of services.

Exploitation of the resulting UI abstract description: When the composition engine has computed the abstract description of the UI for a given composition of services, the resulting AliasCompose model is translated back to AliasExchange models in order to generate code for specific platforms.

At this step, details concerning UI structure and layout choices need to be reintroduced. Hence AliasExchange models which correspond to UIs are annotated with extra information to describe UI elements more precisely with widget-specific characteristics (lists or check-boxes, buttons or menu items, ...) and the position of UI elements. For this, we would reuse our work around ALIAS-Structure and ALIAS-Layout.

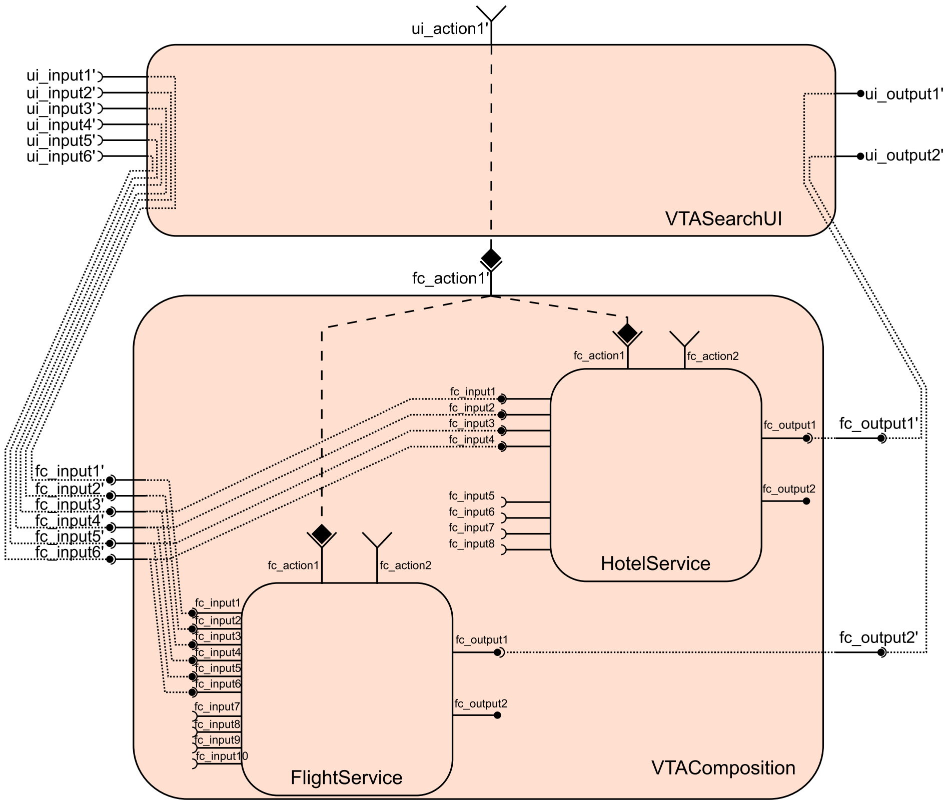

Fig. 6. AliasCompose model for the composition of the Hotel Booking service and the Flight Reservation service and the deduced UI

Annotations would be added using the decorator design pattern. This part of the work is still under progress and what we hope is to be able to transform a decorated AliasExchange model into an existing model of abstract UI dedicated to plasticity (such as Teresa [9] for example) to address an ergonomic final UI. We do not discuss this point further in this paper.

4 Overview of the Transformations Involved in the Composition Process

This section describes the end-to-end transformation chain that operationalizes the composition process steps described in section 2.2. The overview of the transformation chain is depicted in Figures 7 and 8. Full arrows represent automated transformations and dashed arrows represent hand-crafted transformations.

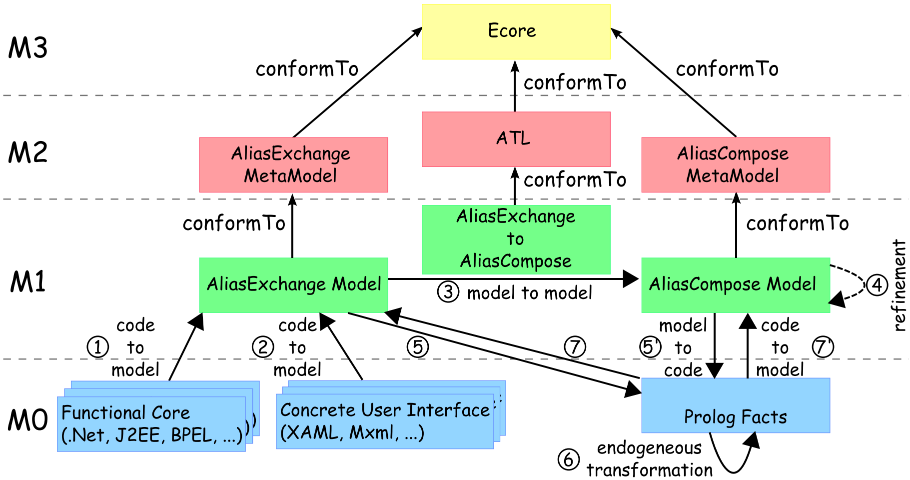

Fig. 7. Abstraction and composition related transformations

- Transformations 1 and 2 correspond to "step 2" and consist in the extraction of information about each existing concrete service and UI and its expression into the AliasExchange formalism.
- Transformation 3 is a pre-processing task of "step 3" to obtain UI and services in the AliasCompose formalism. Transformation 4 corresponds to "step 3" in which the developer specifies: (i) the interaction links that exist between existing concrete UI and services in the AliasCompose formalism (first refinement step) and (ii) the composition links between services in the AliasCompose formalism (second refinement step).
- Transformations 5, 5', 6, 7 and 7' are related to "step 4" (the use of the composition engine) and consist in a technological space shift. They correspond respectively to: the generation of prolog facts from alias models, the internal rule inference, the translation of prolog results back to alias models specifying the structure of the resulting UI (AliasExchange model) and its links to the composition of services (AliasCompose).
- Transformation 8 and 8' correspond to "step 5" and give feedback to application developers and UI designers on the UI composition. These transformations are very important because one can see whether the result of the composition is correct or not. It is easier to deal with a concrete UI than with Prolog facts or with an abstract UI.

Each transformation has been denoted based on the classifications of Czarnecki [10] and of Mens [11]. The figure outlines the fact that our composition

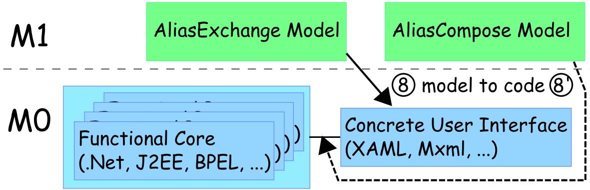

Fig. 8. Concretization related transformations

process covers a large spectrum of transformations: 1) endogenous and exogenous, 2) vertical and horizontal, 3) reverse engineering (abstraction from PSM to PIM), synthesis (concretization from PIM to PSM), migration as well as internal refinement.

The diversity of transformations has lead us to experiment with different tools and mechanisms. Then we use pure MDE tools such as ATL [12] and Acceleo[2] as well as ad-hoc techniques such as implementing visitor design patterns, conditional Transformations [13] (which are based on a description of a model inside Prolog facts - with some rules of transformations we can describe how we would like to transform a source model into another one or into a source code).

For the time being, we are not sure of which combination of tools best fits. Some questions are still open such as:

– How to make a transformation bidirectional? We would like to use only one implementation in order to automate transformations 4 and 8' because the latter is really the inverse of the former concerning the first refinement.
– Which technique better deals with runtime and instance level transformation? We plan to use Alias at runtime in order to adapt the UI as a function of service discovery and disappearance in the context of ubiquitous computing. Then, transformations would occur at runtime and data transformation would be at an instance level not at a class/type level.

Lastly, limits of the current transformation chain essentially concern reverse engineering of UIs and services. Transformations 1 and 2 are implemented as a visitor. However, this approach is only possible if we work on source code. We plan to try out an extraction technique based on language reflection such as proposed by Bezivin et al [14]. Transformation 4 is handwritten because it is too complicated to extract interaction links using a visitor or transformation rules in the case of the first refinement but it can be automated easily for the second refinement (composition links). The automation of transformation 4 is not essential in a design approach but is crucial if we want to use Alias at runtime. We plan to test the extraction technique based on language reflection also for the automated discovery of composition links by exploiting introspection over composition formalisms such as component assemblies [8], [15], [16] or service orchestrations [17], [18].

[2] http://www.acceleo.org

5 Related Work

Models have long been used in Human-Computer Interaction (HCI) to make knowledge explicit. Rapidly, efforts have been put on UI code generation to make explicit and to reuse the know-how in HCI. Nowadays, models and transformations have been rediscovered under the umbrella of Model Driven Engineering (MDE) to tackle problems such as UI composition or UI plasticity (adaptation of the UI to the context of use while maintaining ergonomic properties [19]).

The Cameleon Reference Framework (CRF) [3] defines four levels of abstraction of UIs: (i) Task and concept, both defining the user and system tasks and the concepts of a specific domain, (ii) Abstract User Interface (AUI) describing the structure of the user interface without any specific widget, (iii) Concrete User Interface (CUI) describing widgets of the User Interface and specifying also elements from the AUI, and finally (iv) Final User Interface (FUI) implementing the CUI in a specific language. The two first levels are PIMs. The two last levels are PSMs (for a language point of view or from a operating system point of view). The framework also proposes transformations to shift from one level to the next one. The metamodels described in this article are at the AUI level: AliasExchange is a subset of AUI, AliasCompose reifies additional information about composition/interaction links.

Many approaches adopt an MDE approach based on this four level model abstraction of UIs. For example, [20] or [21] use AUI or CUI to compose user interfaces. At the end, they use model transformation to generate a FUI usable for an end-user. In such work, composition is addressed but mostly from an ergonomic and usability point of view. the composition process is based on the structural aspects of user interfaces located at the Abstract/Concrete User Interface (AUI/CUI) or task level and does not exploit information related to the functional part as does Alias.

Servface [22], a European project, decorates service descriptions with user interface annotations. These annotations allow for the generation of a high quality UI to interact with the annotated services. The generation process of a FUI is based on refinement in the different models presented in [3]. Servface and Alias share the same goal: building a user interface for a composition of services. However, Servface composition is implemented by using a task tree description (service operations are bound to system tasks) and annotations whereas Alias composition extracts the tasks sequences in UI interactions from the service workflow instead of duplicating such information.

Work on planning [23] proposes an approach to compose interactive services (*e.g.* functional core and UI) from user needs. For this, the framework asks users about what they want in a natural language. Then, an incomplete task model is built and transformed into a planning problem. After that, the results of the planning problem are translated back into task models which are refined successively into a AUI, CUI and then FUI.

Lastly but not least, most of these approaches work at one CRF level and then define successive transformations to obtain a FUI. Hence, they follow a top-down approach and use mostly model-to-code transformation. In contrast,

the Alias composition process adopts a top down approach as well as a bottom up one as existing artifacts are extracted from code. Alias makes greater use of transformation techniques as mentioned in section 4. However, work on planning and Alias have both demonstrated the power of model transformation for bridging domains together: the first one between UI composition and planning and the second one between UI composition and predicate logic.

6 Conclusion

Alias is a logical approach for composing user interfaces at the Abstract User Interface (AUI) level. Its originality comes from the fact that the composition is deduced from the way services are composed. This paper has shown the pertinence of adopting a MDE [4] approach with regards to our needs:

- expressing the composition algorithm in a formalism that best fits,
- reusing the composition algorithm for different concrete platforms (service-based platforms such as OSGi, web services, SCA or component-based platforms such as Fractal, OpenCCM, Sofa and so on),
- handling heterogeneity of UI description languages.

We have shown how we used MDE to operationalize the Alias composition process. We definitively think that MDE brings a lot of benefits in the UI composition research area. This approach enables the isolation of the composition algorithm. Changing of formalism will not impact all the framework but only the transformations directly related to this aspect. In the same way, using reverse engineering prevents the designer from focusing on platform diversity and avoids combination issues that would arise if we translate the concrete services and UI directly into the composition algorithm in PROLOG.

In addition, the use of MDE gives a lot of opportunities to the Alias framework. Considering that work around plasticity in the Human-Computer Interaction (HCI) community is based on models at different levels, our ultimate goal is to transform Alias models into plasticity models to obtain final user interfaces with ergonomic properties.

Acknowledgements

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

References

1. Natis, Y.V.: Service-oriented architecture scenario. Gartner, Inc. (2003)
2. Heineman, G., Councill, W. (eds.): Component-Based Software Engineering, Putting the Pieces Together. Addison-Wesley, Reading (2001)
3. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interacting With Computers 15/3, 289–308 (2003)
4. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer 39, 25–32 (2006)
5. OMG: Model Driven Architecture. OMG Document ormsc/2001-07-01 (2001)

6. Pinna-Déry, A.M., Joffroy, C., Renevier, P., Riveill, M., Vergoni, C.: ALIAS: A Set of Abstract Languages for User Interface Assembly. In: SEA 2008, Orlando, Florida, USA, IASTED, pp. 77–82. ACTA Press (2008)
7. The Object Managemant Group: Unified Modeling Language Specification 2. OMG Document formal/2009-02-02 (2009)
8. Marino, J., Rowley, M.: Understanding SCA (Service Component Architecture). Addison-Wesley Professional (2009)
9. Mori, G., Paternò, F., Santoro, C.: Design and development of multidevice user interfaces through multiple logical descriptions. IEEE Transactions on Software Engineering 30, 507–520 (2004)
10. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
11. Mens, T., Gorp, P.V.: Applying a model transformation taxonomy to graph transformation technology. Electronic Notes in Theoretical Computer Science 152, 143–159 (2006)
12. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, p. 128. Springer, Heidelberg (2006)
13. Kniesel, G., Koch, H.: Program-independent composition of conditional transformations. Technical Report IAI-TR-03-1, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany (2003) (updated February 2004)
14. Bézivin, J., Chevrel, R., Brunelière, H., Jossic, A., Jouault, F., Piers, W.: Modelextractor: an automatic parametric model extractor. In: The international workshop on Object-Oriented Reengineering (WOOR) at the ECOOP 2006 Conference, Nantes, France (2006)
15. The Object Managemant Group: CORBA Component Model Specification, 4.0 edition. OMG Document formal/2006-04-01 (2006)
16. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper. 36, 1257–1284 (2006)
17. Peltz, C.: Web services orchestration and choreography. Computer 36, 46–52 (2003)
18. Khalaf, R., Mukhi, N., Weerawarana, S.: Service-oriented composition in bpel4ws. In: WWW (Alternate Paper Tracks) (2003)
19. Scapin, D., Bastien, J.: Ergonomic criteria for evaluating the ergonomic quality of interactive systems. Behaviour & Information Technology 16, 220–231 (1997)
20. Lepreux, S., Hariri, A., Rouillard, J., Tabary, D., Tarby, J., Kolski, C.: Towards Multimodal User Interfaces Composition Based on UsiXML and MBD Principles. In: Jacko, J.A. (ed.) HCI 2007. LNCS, vol. 4552, p. 134. Springer, Heidelberg (2007)
21. Pinna-Déry, A.M., Fierstone, J.: Component model and programming: a first step to manage Human Computer Interaction Adaptation. In: Chittaro, L. (ed.) Mobile HCI 2003. LNCS, vol. 2795, pp. 456–460. Springer, Heidelberg (2003)
22. Servface Project: Service annotation for user interface composition (7th Framework European Programme Project) (2008), `http://www.servface.org`
23. Gabillon, Y., Calvary, G., Fiorino, H.: Composing interactive systems by planning. In: UbiMob 2008, Saint Malo, France, pp. 37–40 (2008)