Viewpoint Synchronization of UWE Models

Daniel Ruiz-González[1], Nora Koch[2], Christian Kroiss[2], José-Raúl Romero[3], and Antonio Vallecillo[1]

[1] GISUM/Atenea Research Group. Universidad de Málaga, Spain
[2] Web Engineering Group, Ludwig-Maximilians-Universität München, Germany
[3] Atenea Research Group. Universidad de Córdoba, Spain {daniruiz,av}@lcc.uma.es, {kochn,kroiss}@pst.ifi.lmu.de, jrromero@uco.es

Abstract. Viewpoint modeling has demonstrated to be an effective approach for specifying complex software systems by means of a set of independent views and correspondences between them. As any other software system, a Web application evolves during its lifetime, and its specifications change to meet new requirements or to adapt to business changes. As a consequence, view elements and correspondences can be added, modified or removed, which may cause synchronization and consistency problems in the system specifications. UWE is a Model-Driven Web Engineering approach that uses several viewpoints for addressing the different concerns involved in the specification and development of complex Web systems. In this paper we examine the explicit and partially automatic specification of correspondences between UWE views, and present a tool for helping synchronize them under the presence of changes to the view elements.

1 Introduction

Viewpoint modeling is currently seen as an effective technique for specifying complex software systems by means of a set of independent viewpoints and correspondences between viewpoint elements. Each viewpoint focuses on a particular aspect of the system, abstracting away from the rest of the concerns. Correspondences specify the relationships between the elements in different views, together with the constraints that guarantee the consistency among these elements.

During its life cycle, a software system evolves and its specification is subject to changes in order to meet new requirements or to adapt to business changes. Thus designers may have to add, remove and/or modify elements in the views, or in the set of correspondences. One of the consequences of adopting a multi-viewpoint approach to system design is that description of the entities that appear in different views must be consistent. Thus, we should always keep them synchronized. One possible solution is the adoption and implementation of synchronization mechanisms able to propagate the changes on the related views. In particular, a modification in a view may induce a modification in another view or, alternatively, a set of correspondences may need to be adapted.

Web Engineering is a specific discipline in which both Model-Driven Software Development and Viewpoint Modeling can be successfully applied. For example, existing Model-Driven Web Engineering (MDWE) approaches such as OO-H, UWE, OOWS,

WebML, MIDAS or beContent (see [1] for a comprehensive survey covering the majority of these proposals), already provide a set of suitable methods and tools for the design and development of most kinds of Web applications. In particular, the UML-based Web Engineering approach (UWE) is a well-known MDWE methodology that defines four basic viewpoints on a Web system for structuring its specifications: Content, Navigation, Business Process and Presentation. However, very few MDWE approaches provide notations and mechanisms for establishing correspondences between their viewpoints, or for synchronizing them when they evolve.

In this paper we examine the explicit specification of correspondences between UWE views, and present a tool for synchronizing them under the presence of changes to the view elements or to the set of correspondences. Correspondences between UWE model elements can either be derived automatically from relationships at metamodel level (intensional approach) or specified individually for model elements (extensional approach). We use classes and dependencies of the Unified Modeling Language (UML), respectively, for expressing and visualizing the correspondences between UWE views. Once the correspondences are specified, the possible changes are propagated using Beanbag [2], a very flexible and efficient notation and engine for synchronizing general data dictionaries.

The structure of this paper is as follows. After this introduction, Sections 2 and 3 present some preliminaries about viewpoint modeling and UWE. Section 4 presents our proposal for specifying correspondences between UWE views. Then, Section 5 describes how the UWE views can be synchronized using the information provided by the correspondences, and the tool we have developed to support the change propagation and synchronization process. Finally, Sections 6 and 7 discuss some related works and present some conclusions, respectively.

2 Viewpoint Modeling and Evolution

One way to cope with the inherent complexity of large distributed software systems is by dividing the design activity according to several areas of concerns, or **viewpoints**, each one focusing on a specific aspect of the system, as described in IEEE Std. 1471 [3]. This approach has been adopted by most MDWE methodologies that propose the construction of different views (i.e., models) which comprise at least a content model, a navigation and a presentation model — although naming them differently. Each viewpoint addresses one particular concern, and normally uses its own specific (viewpoint) language, which is defined in terms of a set of concepts specific to that concern, their relationships, and their well-formedness rules. A **view** is a representation of the whole system from the perspective of a viewpoint.

Although separately specified, developed and maintained to simplify reasoning about the complete system specifications, viewpoints are not completely independent: elements in each view need to be related to elements in the other views in order to ensure the consistency and completeness of the global specifications. Such relationships are described in terms of **correspondences** [4].

In a viewpoint modeling context, viewpoint languages are defined in terms of metamodels, and the views are then models that conform to these metamodels. Correspon-

dences are defined as models, too, which conform to the appropriate metamodels. Such Correspondence metamodels can be defined either ad-hoc (see, e.g., [5]) or use a model transformation language in order to define viewpoint correspondences as model transformations (see, e.g., [6]).

System specifications can evolve due to changes in the requirements or for many other reasons. Thus, the designer may need to perform changes in the views by modifying, for example, one of their elements. Since the system is described as a set of dependent views, any action on a view might cause a similar or a different action on other views. If views are not explicitly related by correspondences, the modeler will not be able to easily know which elements in the views have some relationship with the modified element. Since any action on a model element can be affected by a correspondence, inconsistencies need to be found and solved.

There are several problems that may happen when trying to maintain the synchronization and consistency between views, once a change has happened (either in an element of a view, or in a correspondence, see [7]). Sometimes the consistency can be easily restored if the correspondences provide enough information to propagate the change.

3 UML-based Web Engineering

UML-based Web Engineering (UWE) is a method for systematic and model-driven development of Web applications. It follows the principle of "separation of concerns" by modeling the content, the navigation structure, the business processes, and the presentation of a web application separately. UWE implements a model-driven development process by defining model transformations of different types to derive platform specific models from platform independent models and to generate running programs [8]. UWE relies on standards: its modeling language is defined by an extension of the UML metamodel [9] and mapped to a so-called UML profile; its transformations are defined using (de-facto) standard transformation languages like ATL [10].

In order to illustrate UWE's approach and our proposal we will use a running example, the *Simple Address Book* [11]. It models an application that manages an address book of contacts. Each contact contains a name, two phone numbers (main and alternative), two postal addresses (main and alternative), and one e-mail address. The system should offer a page with an introductory text and the list of contacts in the agenda, which is stored in a database. For each object the set of its non-empty attributes values is displayed. The address book web application supports browsing of contacts, adding new contacts, editing and removing of contacts.

The content model in UWE (represented by a UML class diagram, see Fig. 1) provides the specification of the domain-relevant information for the Web application. The content model of our example contains contacts that are organized in an address book.

Based on the content model, the navigation model of the Web application is built (Fig. 2) by a model transformation and a set of successive refinements. This model specifies the hypertext structure of the system, which is described in terms of *nodes* and *links*. Classes stereotyped ≪navigationClass≫ (like AddressBook or Contact) represent navigable nodes for information retrieval; classes stereotyped ≪processClass≫ (such

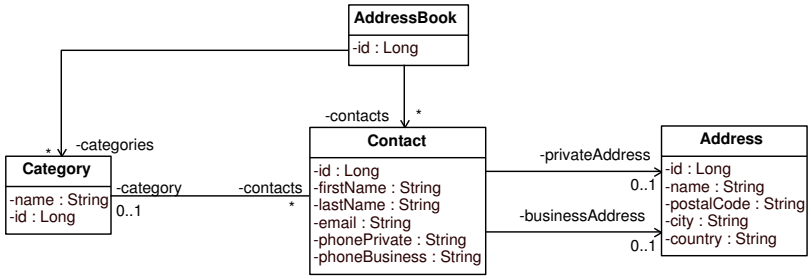

Fig. 1. Content model of the AddressBook example.

as AddContact and RemoveContact) define navigation nodes where transactions may occur. Direct links are modeled by associations; in particular, associations stereotyped ≪processLink≫ lead to or leave from process classes. Some special navigation nodes are used to organize links. For example, several instances of a navigation class are reached by ≪index≫ classes (like ContactList) and choices of links are represented by ≪menu≫ classes, like ContactMenu (see Fig. 2). In the address book application, users can navigate from the homepage using an index with the contacts, either to view the information associated to a selected contact, or along another navigation path to add or remove a contact from the database.

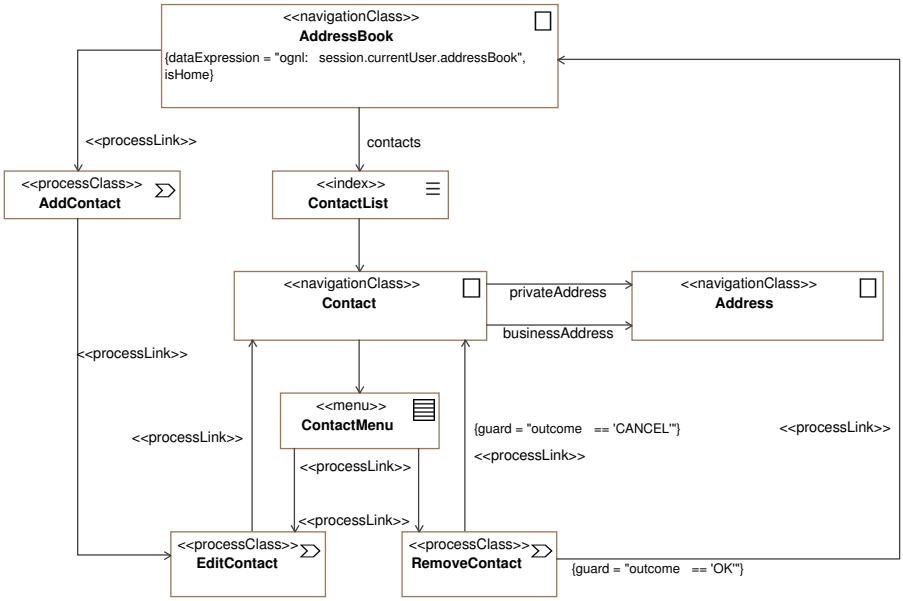

Fig. 2. UWE: Navigation model of the AddressBook example.

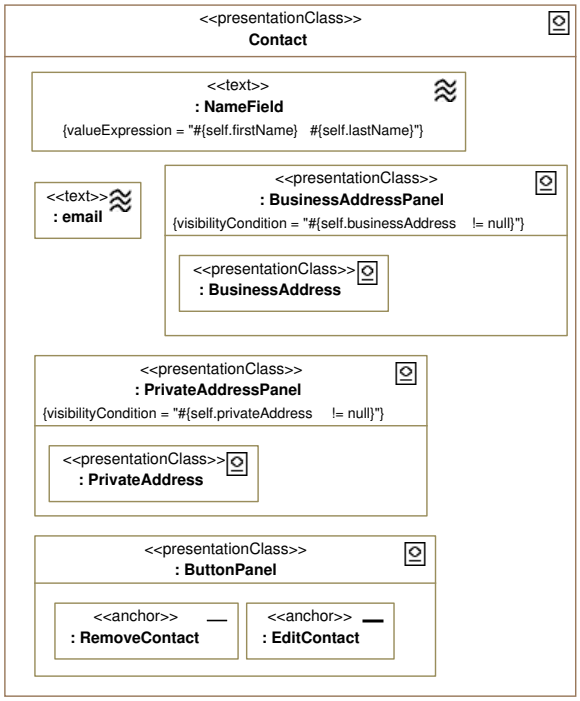

Fig. 3. UWE: Presentation model of the AddressBook example (excerpt).

Each process class in the navigation model is refined by a process structure (described in a class diagram) defining additional classes used in the process, and by a process flow (described by a UML activity diagram) modeling the data and control flow of the process. For example, user interactions with the web application are modeled by UML actions stereotyped as ≪userAction≫.

Finally, the presentation model provides an abstract view of the user interface (UI) of the application, where concrete aspects such as colors and fonts of UI elements are not considered. A structured class stereotyped as ≪presentationClass≫ models the presentation of each navigation class. UI elements (e.g., text, images, etc.) contained in presentation classes are modeled by classes stereotyped accordingly (≪text≫, ≪image≫, etc.), indicating the abstract type of the widgets to use. Presentation classes can be nested, modeling the hierarchical structural of web pages. A presentation class that is not contained in another represents a top-level page of the Web application. In addition, a presentation class is defined for each user action. The presentation model therefore has the form of a forest of presentation classes. An excerpt of the presentation model of the address book application is shown in Fig. 3.

4 Representing Correspondences

The objective is to build the correspondence model that comprises a set of correspondences between model elements of a pair of views. The specification of the correspondences is not a difficult but a tedious task. However the work is worth as it allows for synchronization of the different views of a model. It is then an unavoidable work. Therefore our focus is on the automation of the correspondence specification process. This is particularly easy in UWE due to the characteristics of its metamodel. There are then basically two approaches to model correspondences between the views of a system: *extensional* and *intensional*.

4.1 Extensional Specification of Correspondences

Extensional models describe correspondences between the particular elements of the views. In UWE, views are expressed as UML models. The UML 2 language defines *abstraction dependencies*, possibly constrained by OCL statements, as the natural mechanism to model a relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints [9]. Thus, correspondences between related elements in two views can be expressed using UML dependencies between them. We make use of the fact that dependencies are directed relations, and that the client of the dependency "depends" on the supplier. This establishes a semantic relationship between them that indicates which one should change if a modification is made to one of them.

Moreover, we mentioned above the importance of endowing correspondence specifications with enough information to propagate the changes. In some approaches this is accomplished by adding a constraint, which establishes a relationship that must hold between the elements related by the correspondence. OCL can be the natural language of choice here, enabling a very flexible and expressive way to specify such constraints [5]. Other authors [6] also suggest the use of model transformation languages (e.g., QVT [12]) for establishing the relations and for specifying (and enforce) the changes. The problem with these approaches resides precisely in their flexibility and expressive richness, since they assume the system designer is able to specify complex constraints using these notations.

A more simple, pragmatic and practical solution is based on the characterization of the kinds of correspondences that are normally used, and the "componentization" of their behavior so that they can be easily re-used. For instance, one common kind of correspondence found in most UML models establishes that the two related elements should have the *same name*. Another usual correspondence dictates that an element in a view should have a *subset* of the attributes of the related element in the other view. For instance, this is found in UWE models when relating an element in the Content model with its counterpart in the Presentation model: the attributes of the latter need to be a *subset* of the attributes of the former (no new attributes can be added in presentation view).

Table 1 shows some examples of the most common correspondences found in UWE specifications. Some of them are specific to UWE (e.g., validIndex), while some others can be used in any multi-view UML specification (e.g., sameName). The *semantics*

Supplier	Client	Correspondence type
Content::Class	Navigation::NavigationClass	sameName
Content::Class	Navigation::ProcessClass	sameName
Content::Association	Navigation::Index	validIndex
Navigation::Link	Presentation::Anchor	validAnchor
BusinessProcess::UserAction	Presentation::Button	validButton

Table 1. Examples of common correspondences in UWE.

of each kind of correspondence (i.e., the relation that must hold between the related elements) is predefined. For example, the semantics of a sameName correspondence can be defined by the OCL expression {client.name = supplier.name}. The fact that correspondences are defined using directed dependencies also indicates how the change should be propagated in case of de-synchronization (that is, from the supplier to the client).

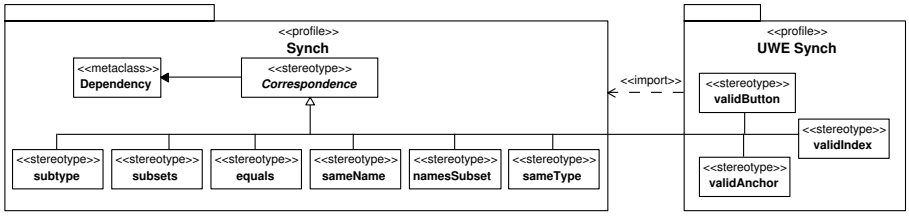

Fig. 4. The Synch and UWESynch Profiles (excerpt).

The idea is then to stereotype the UML dependencies that represent the correspondences with the identifier of the semantics they are expected to exhibit. For this reason we have defined a UML profile, which is shown in Fig. 4. It contains one stereotype for each kind of correspondence, and is currently structured in two related profiles. The first one, Synch, contains those correspondences which can be used in any UML multi-view specification. The second one, UWESynch, contains those kinds of correspondences which are specific to UWE. Fig. 5 show some examples of these correspondences.

4.2 Intensional Specification of Correspondences

It is not realistic that the system designer has to deal with definition and visualization of all the correspondences of a system. In particular, it is almost impossible to cope with their specification, management and maintenance in case of large systems, with thousands of correspondences between their elements.

Intensional approaches define correspondences as relations between *types* of model elements at metamodel level, instead of between the model element themselves — i.e., between viewpoint elements and not between view elements. However, this approach is

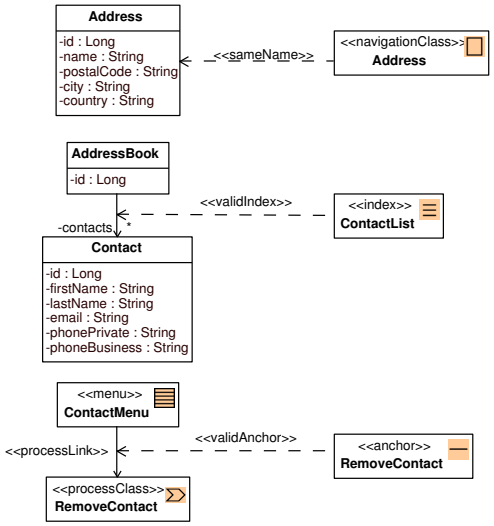

Fig. 5. Example of correspondence specifications using our profile.

not free from problems either: for typical users of the specification, correspondences are easier to use, visualize and understand when they are drawn as relationships between individual elements in the views, instead of expressed as formulae relating metamodel elements.

In UWE, some intensional correspondences are defined by its metamodel, which defines some relations between types of model elements: a navigation class is related to a class of the content model; an anchor of the presentation model is related to a link in the navigation model, etc. For example, the relation for the validAnchor correspondence is specified by the following OCL expression:

```
client.getPresentationContainer().getNavigationNode() =
   supplier.getLinkSource()
   and client.class.name = supplier.getLinkTarget().name
```

Note that for the evaluation of the expression we define a set of OCL helper functions, such as getPresentationContainer and getLinkTarget, which handle the UWE specific stereotyped elements.

In our approach, starting with a set of UWE views, our tool automatically generates the initial set of extensional correspondences from the intensional ones (see Sec. 5.2) by interpreting a collection of expressions like the one shown above. Once the whole set of correspondences is modeled, the designer can then add or modify those which are required in his/her particular case, although normally they are just a few (Fig. 8). The fully automated generation greatly facilitates the task of defining correspondences in UWE models.

4.3 Well-formed rules

Apart from specifying the correspondences between model elements, any proper multi-view specification of a system should also allow the specification of *required correspondences*, which describe the well-formed rules that the set of correspondences between elements of views should obey [13]. For example, UWE requires that every ≪anchor≫ of the presentation view should be related to a ≪navigationLink≫ or ≪processLink≫ of the navigation view by a ≪subsets≫ correspondence. This means that for a UWE specification to be correct, the set of correspondences should include one of these correspondences for every element in the presentation view.

In our proposal, this is expressed by a set of OCL constraints on the set of correspondences. They can be checked by any OCL engine, e.g., the MagicDraw validation engine that comes with some of its latest versions, or with any external OCL tool. In any case, validating these well-formed rules over the set of correspondences falls outside the scope of this paper, as it happens with the validation of the well-formed rules of the view elements (i.e., the intra-view consistency).

5 Synchronization

Synchronization of the different views of a model can be performed once inconsistencies in the views are detected. Inconsistencies can be found by checking that correspondences are valid for the existing model elements. If not, the synchronization tool proposes changes with the objective to achieve synchronized views again.

5.1 Correspondence Model

We have mentioned above that correspondences between the related elements in separate views are modeled by UML dependencies, stereotyped according to the kind of semantics we want to impose to the correspondence. UML dependencies provide a very convenient mechanism for drawing correspondences in a easy manner, and for visualizing them. However, UML dependencies are quite *volatile*, in the sense that the user may easily delete or modify them without the system being aware of the changes. For example, if the user removes one of the classes related by a dependency, the dependency is also removed, which may not be completely correct. This is why we need some kind of more persistent representation of correspondences, which in our approach is provided by classes stereotyped ≪CorrespondenceClass≫. Each UML dependency that represents a correspondence is then associated to one of these classes.

Correspondence classes are managed by the SynchronizeIt! tool in a transparent way, and therefore the user is completely unaware of them. Thus, if the user decides to delete or modify one UML dependency, the tool will be able to know about the change. More precisely, the tool will check that every UML dependency has its associated correspondence class. In case of conflicts the tool reacts as follows:

- If there is a UML dependency and it has a correspondence class with the same name and information, everything is Ok.

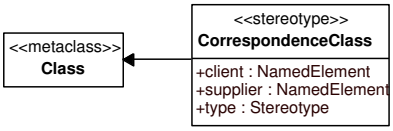

Fig. 6. The stereotype for Correspondence classes.

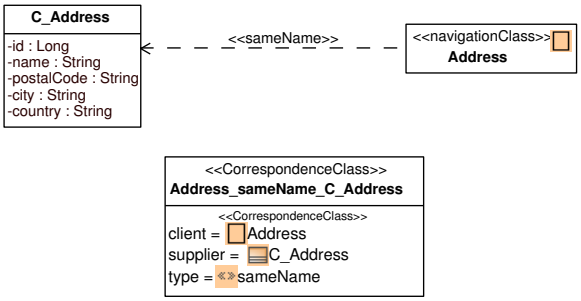

Fig. 7. Example of correspondence specifications using our profile.

- If there is a UML dependency with no associated correspondence class, it is assumed that the dependency has been added by the user, and therefore the correspondence class is created and added to the model.
- If there is a correspondence class with no associated UML dependency, it means that the user has deleted the UML dependency or any of the associated classes.
  - If the two related classes exist but the UML dependency does not, the correspondence class is removed.
  - If the supplier class exists but neither the client nor the UML dependency do, the correspondence class is also removed.
  - If the client class exists but neither the supplier nor the UML dependency do, a warning is raised because the user seems to have removed the supplier and left the clients with a "dangling" dependency to it.

Fig. 6 shows the specification of the stereotype for correspondence classes, and Fig. 7 shows an example of the correspondence class associated to the top correspondence of Fig. 5. As we can see, we have also simulated that the designer has decided to change the content model, renaming class Address to C_Address – probably without being aware that there was a class in the navigation model that was related to it (maybe even more elements in other models, not only that one). This has produced an inconsistency in the models, which could go easily unnoticed if no explicit correspondences have been specified. And even in this case, the propagation of changes to re-synchronize the model is not an easy task if if has to be manually performed. This is why we need an automated process for change propagation.

5.2 Synchronization Process

Fig. 8 shows an overview of the synchronization process. Starting from the set of UWE models with the different views (Sec. 3), the tool automatically builds the initial set of (extensional) correspondences between the elements in the separate views. These extensional correspondences are derived from the set of intensional correspondences defined in the UWE metamodel (see Sec. 4). In addition, the system designer can define more correspondences depending on the particular requirements of the application.

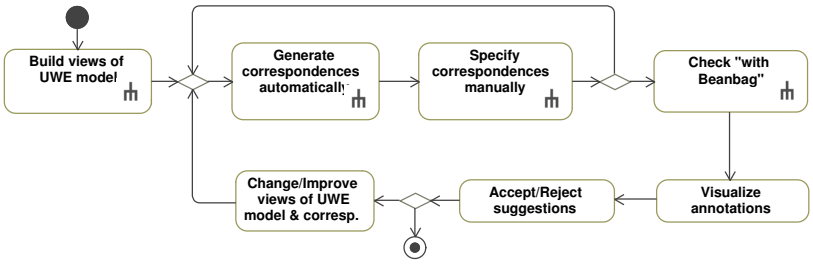

Fig. 8. Overview of the synchronization process.

The consistency of the complete UWE system specification can be checked based on the different UWE models (each one representing a UWE view) and the sets of correspondences between their elements. The checking process is depicted in Fig. 9, and further detailed in Sec. 5.3.

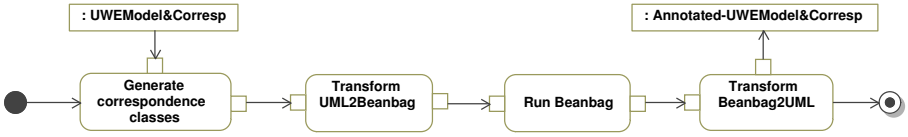

Fig. 9. Consistency checking process.

The result of this process is a set of annotations which are added to the original UWE model. These annotations are represented by means of stereotypes which decorate elements of the UWE model (including its correspondences) indicating that for these elements the synchronization imposed by a correspondence has been broken, together with the proposed changes to restore it. The user can then visualize the annotated model, being able to modify some of the proposed changes if required. Once the designer is happy with all the proposed changes, another process simply traverses all model elements and implements the proposed changes, removing the annotations. Fig. 10 shows the UML profile used for annotating the models, while Fig. 11 shows an example of its application after the synchronization process (using the example model of Fig. 7).

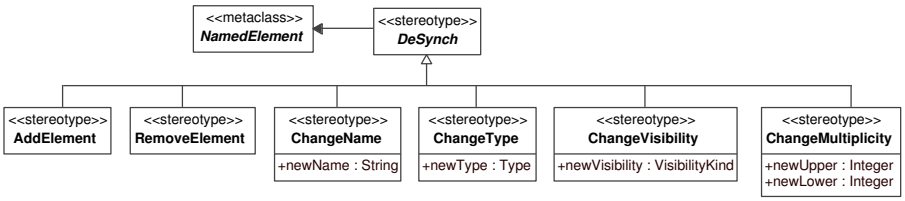

Fig. 10. Profile for annotating de-synchronized models.

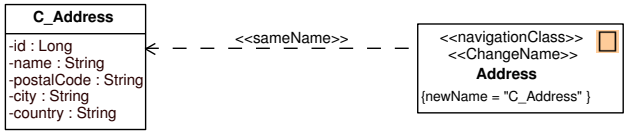

Fig. 11. Example of annotated model after synchronization checks are performed.

5.3 Tool Support for Synchronizing the Models

The tool we have developed supports the process depicted in Fig. 9 [14], which in turn uses the Beanbag tool [2] to detect and resolve the inconsistencies found. Beanbag is a very generic and powerful synchronization engine, which handles general data dictionaries. Thus, the UWE models need to be transformed into the data dictionaries that Beanbag is able to deal with. This is done by a set of ATL transformations, which extract the relevant information from the models, and express it using two dictionaries: one for the model elements involved in the correspondences, and other for the correspondences themselves. For instance, the example depicted in Fig. 7 gets transformed into the following Beanbag dictionaries:

```
{ // Dictionary of model elements
  "Content::C_Address"->{name->"C_Address"} ,
  "Navigation::Address"->{name->"Address"} ,
}
{ // Dictionary of correspondences
  1->{type->"sameName",
      idEndC_Address->"Navigation::Address", idEndAddress->"Content::C_Address"},
}
```

Then the Beanbag engine is executed using a set of synchronizing functions, one for each type of correspondence. These functions are defined in a Beanbag program and then compiled into a synchronizer. For instance, the Beanbag function we have defined for ≪sameName≫ correspondences is the following:

```
sameName (entity1, entity2) {
   entity1."name" = entity2."name";
 }
```

Please note that the direction of the change propagation (i.e., which element is the supplier and which one is the client) is deduced from the order in which the elements are listed in the corresponding entry in the dictionary of correspondences.

The result of the Beanbag execution is another Beanbag dictionary, with the changes required to synchronize both models. Such changes include additions, deletions and modifications to the data dictionaries. In our example, what we get is the following:

```
{
  "Navigation::Address"->{name->"C_Address"},
}
```

Finally, this information is used to annotate the UWE models with the required changes to restore the synchronization, producing in this case the model shown in Fig. 11.

6 Related Work

There is a large number of approaches related to our work. In the first place we have the works on synchronizing artifacts in software engineering, mostly influenced by the original works on multi-view consistency mechanisms [15,16]. These studies use a generic representation of modifications and rely on users to write code to handle each type of modification in each type of view. This idea has influenced later efforts on general model synchronization frameworks, such as [17,18,19]. Although in principle these frameworks allow the definition of correspondences between views and the propagation of changes, they all try to detect inconsistencies and solve them by changing the affected elements, but without any well-defined underlying architectural framework that allows the precise and explicit specification of viewpoints, views, and correspondences between them. In addition, it is unrealistic to suppose that all inconsistencies can be automatically solved. This is the main problem of many proposals that try to resolve views inconsistencies by automatically synchronizing the models using model transformations (e.g. [2]). In addition, these proposals also suffer the intrinsic problems of model transformations when used for inconsistency solving, as discussed in [20,21].

The approach of Cicchetti and Di Ruscio [22] is similar to our approach, as it establishes relationships between viewpoints of the models of a web application. They provide a weaving model for that. The difference to our approach lies in the semantics of their relationships, which are used to integrate the different viewpoints. The relationships of our approach, instead, have the objective to allow the propagation of changes in case of model evolution.

Ultimately, a key issue to realize and implement change management and propagation is the availability of synchronizing tools that can detect the changes and show them to the designer, so that he can decide what to do, and how to proceed. This may imply propagating the changes either forwards, backwards, or even changing the correspondences. In this respect, the most inspiring work for our proposal are the works (and tools) by Benjamin Pierce and his colleagues in the Harmony group, who have explored bidirectional transformations extensively in the context of trees [23], and more recently in the context of relational databases [24]. In particular, Unison [25] is a file-synchronization tool for Unix and Windows which allows two replicas of a collection of files and directories to be stored on different hosts (or different disks on the same host), modified separately, and then brought up to date by propagating the changes in each replica to the other. Our proposal aims at providing similar functionality, but for elements in different views related through correspondences, instead of files in different hosts related by paths and names, which is a simpler case.

7 Conclusion and Future Work

Software development is increasingly shifting its focus from coding to modeling, thus evolution management techniques tend to encompass also the evolution of model-based artifacts. In particular, in multi-view design, model elements in a view are related by many-to-many correspondences to other view elements and thus any change propagation mechanism should manage the conflicting effects that changes may cause in the whole system.

In this paper we have proposed a notation for the explicit specification of correspondences between UWE views, and have briefly presented a tool for synchronizing these related views under the presence of changes to the view elements or to the set of correspondences. The tool, called SynchronizeIt!, is currently in alpha version and can be downloaded from [14].

There are many different lines of work that we plan to address in the very short term. Firstly we want to fully integrate the tool with MagicUWE, the UWE plugin for MagicDraw [26]. Secondly, we want to extend the validation process to take into account the well-formed rules that the set of UWE correspondences should fulfil. Finally, we want to enhance the current visualization facilities of the annotated models so that the affected elements change their color and appearance when visualized with MagicDraw.

Acknowledgements. We'd like to thank Yingfei Xiong and the people at the Japan National Institute of Informatics for their help and support with Beanbag. This work has been supported by the DFG Project MAEWA II, WI 841/7-2, the EU FET-GC2 IP project SENSORIA (IST-2005-016004) and the Spanish projects TIN2008-03107, TIN2008-00889-E and P07-TIC-03184.

References

1. Rossi, G., Pastor, O., Schwabe, D., Olsina, L.: Web Engineering: Modelling and Implementing Web Applications. Springer (2008)
2. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proc. of ASE'07, New York, NY, USA, ACM (2007) 164–173
3. IEEE: Recommended Practice for Architectural Description of Software-Intensive Systems, New York, USA. (2000) IEEE Std. 1471.
4. Linington, P.: Black Cats and Coloured Birds What do Viewpoint Correspondences Do? In: Proc. of the 4th International Workshop on ODP and Enterprise Computing (WODPEC 2007), Maryland, US, IEEE Digital Library (2007)
5. ISO/IEC: Information technology – Open distributed processing – Use of UML for ODP system specifications. ISO and ITU-T, Geneva, Switzerland. (2008) ISO/IEC FDIS 19793, ITU-T X.906.
6. Romero, J.R., Moreno, N., Vallecillo, A.: Modeling ODP Correspondences using QVT. In: Proc. of MDEIS'06. (2006) 15–26
7. Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multi-viewpoint systems using ASP. In: Proc. of WODPEC 2008, Munich, Germany (2008) 19–28

8. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering: An Approach Based on Standards. In Olsina, L., Pastor, O., Rossi, G., Schwabe, D., eds.: Web Engineering: Modelling and Implementing Web Applications. Volume 12 of Human-Computer Interaction Series. Springer, Berlin (2008) 157–191
9. OMG: Unified Modeling Language 2.1.1 Superstructure Specification. OMG, Needham (MA), USA. (2007) OMG doc. formal/07-02-05.
10. The Eclipse Foundation: ATL: The ATLAS Model Transformation Language. (`http://www.eclipse.org/m2m/atl/`)
11. Busch, M., Koch, N., Kroiss, C.: The Simple Address Book Modeling Example in UWE. (`http://www.pst.ifi.lmu.de/projekte/uwe/exampleSimpleAddressBook.html`)
12. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005) OMG doc. ptc/05-11-01.
13. Romero, J.R., Vallecillo, A.: Well-formed rules for viewpoint correspondences specification. In: Proc. of WODPEC 2008, Munich, Germany (2008) 29–36 `http://www.lcc.uma.es/~av/Publicaciones/08/wodpec2008-correspondences.pdf`.
14. Ruiz-González, D., et al.: The SynchronizeIt! tool. (`http://atenea.lcc.uma.es/index.php/Portada/Resources/SynchronizeIt!`)
15. Finkelstein, A., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multi-perspective specifications. In: Proc. of ESEC'93, London, UK, Springer-Verlag (1993) 84–99
16. Grundy, J., Hosking, J., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE Trans. Softw. Eng. **24**(11) (1998) 960–981
17. Ivkovic, I., Kontogiannis, K.: Tracing evolution changes of software artifacts through model synchronization. In: Proc. of ICSM'04. (2004) 252–261
18. Johann, S., Egyed, A.: Instant and incremental transformation of models. In: Proc. of ASE'04, IEEE Computer Society (2004) 362–365
19. Bottoni, P., Parisi-Presicce, F., Pulcini, S., Taentzer, G.: Maintaining coherence between models with distributed rules: from theory to Eclipse. Electron. Notes Theor. Comput. Sci. **211** (2008) 87–98
20. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Proc. of MoDELS'07. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 1–15
21. Garcia, M.: Bidirectional synchronization of multiple views of software models. In: Proc. of DSML'08. Volume 324 of CEUR Workshop Proceedings. (2008) 7–19
22. Cicchetti, A., Ruscio, D.D.: Decoupling Web Application Concerns through Weaving Operations. Science of Computer Programming, Elsevier **70**(1) (2008) 62–86
23. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007)
24. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Proc. of PODS'2006, ACM (2006) 338–347
25. The Harmony group: The Unison file synchronizer. (`http://www.cis.upenn.edu/~bcpierce/unison`)
26. LMU – Institute for Informatics, Programming and Software Engineering: UWE - MagicUWE. (`http://www.pst.ifi.lmu.de/projekte/uwe/toolMagicUWE.html`)