SelfSync: A Dynamic Round-Trip Engineering Environment

Ellen Van Paesschen
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
Brussel, Belgium
evpaessc@vub.ac.be

Wolfgang De Meuter
Laboratoire d'Informatique
Fondamentale de Lille
Université des Sciences et
Technologies de Lille
59655 Villeneuve d'Ascq
Cédex, Lille, France

Maja D'Hondt
Laboratoire d'Informatique
Fondamentale de Lille
Université des Sciences et
Technologies de Lille
59655 Villeneuve d'Ascq
Cédex, Lille, France

Categories and Subject Descriptors: D.2.2 Software Engineering, Design Tools and Techniques [CASE, OO design methods, evolutionary prototyping], D.2.13 Software Engineering, Reusable Software, D.2.6 Software Engineering, Programming Environments.

General Terms: Design, Experimentation.

Keywords: Model-driven Engineering (MDE), Round-Trip Engineering, dynamic languages, prototype-based programming, rapid prototyping, role modeling.

Extended abstract

Model-Driven Engineering (MDE) advocates the generation of software applications from models, which are views on certain aspects of the software. We focus on a particular setup which consists of a graphical data modeling view and a view on an object-oriented implementation. In our Round-Trip Engineering (RTE) tool SelfSync [3], [2], the entities of the data modeling view and the corresponding implementation objects are one and the same resulting in a continuous, highly dynamic synchronization between the two views.

SelfSync is the first prototype-based[1] RTE environment as it successfully combines prototypes and Extended Entity-Relationship (EER) [1] diagrams. This environment is built on top of the object-oriented prototype-based language Self [6] and integrates a graphical drawing editor for EER diagrams. SelfSync realizes co-evolution between domain analysis objects in an EER diagram and Self implementation objects.

A bidirectional link between domain analysis objects and implementation objects is created by adding an extra view to the model-view-controller (MVC) architecture of the Morphic user interface in Self. The existing view of a Self *outliner* (i.e. the default graphical representation of implementation objects) is extended with an EER *entity view* in such a way that both views represent the same underlying Self object (the model). Both views are connected and synchronized onto the level of attributes and operations.

Two-Phase Approach. Development in SelfSync constitutes two phases. In the first, *active modeling* phase a user draws an EER diagram while corresponding Self objects are automatically created. In reality, these objects *are* the modeled entities: drawing a new EER entity automatically results in an *EER entity view* being created on a new object. Hence, we support incremental and continuous synchronization *per entity* and *per object*: changes to an EER entity are in fact changes to the outliner of an object and thus are automatically propagated to the object via Self's reflection mechanism. Similarly, changes to an object, made via the object's outliner, are automatically propagated to the corresponding EER entity. View-dependent information, such as relationships constraints in the EER diagram and method bodies in the Self objects, is preserved during changes and subsequent synchronization.

The second phase of our approach is an *interactive prototyping* process [2]. This phase allows a user to create and initialize ready-to-use objects from each implementation object created in the previous phase, thus *populating* the application. We use the term *population object* to distinguish the objects that result from this phase with the implementation objects that are created in the previous phase.

This phase cannot be supported in a fully automatic way, because choices need to be made that depend on the preferences of the user of the program.

One Repository - Three Views. Performing the two-phase approach described above results in a setup that consists of one common repository, the actual Self code, and three views on it:

- the EER view consists of all the information pertaining to entities as well as inheritance between entities and associations with multiplicities between entities
- the outliners on the implementation objects, which show everything related to object-oriented programs;

[1]Prototype-based languages can be considered object-oriented languages without classes.

[2]Note that a *prototype* is a special object in prototype-based languages for supporting data sharing of several objects whereas *prototyping* is the activity of instantiating and initializing a program into a ready-to-use, running system.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

programmers can enrich the implementation objects with additional attribute slots, fill in the method bodies, create new implementation objects manually, etc.

- the outliners on the population objects, which contain actual data for running the application

SelfSync synchronizes the three views, which is partly facilitated because the objects in the three views are actually different views on the same Self code. On the other hand, view-dependent information is not visible in all the views. For example, multiplicities in EER diagrams are not visible in the population objects but are nevertheless enforced by SelfSync.

The crucial difference between SelfSync and other round-trip engineering tools, such as the highly advanced TogetherJ [7], is that SelfSync explicitly considers the third view, i.e. the running application populated with objects, and includes it in the round-trip engineering process. In other approaches, only the implementation objects, which are typically class definitions, are synchronized with the modeling view. Once the classes are instantiated into objects, which we refer to as population objects, the synchronization with the modeling view is no longer supported.

The Power of SelfSync. SelfSync has a number of typical characteristics that enable advanced and dynamic round-trip engineering between the EER diagram and the object-oriented program in Self:

1. **Multiplicity constraint enforcement.** After the interactive prototyping phase we ensure that the multiplicity constraints imposed by a one-to-one or one-to-many relationship between two entities are satisfied at all times. When two entities are in a relationship in which the first one has a single reference (one or zero) to the second one, the uniqueness of this reference is enforced in the population objects in two ways. We first ensure that all population objects that have been derived from the first entity refer to at most one population object that has been derived from the second entity. Secondly, we also ensure that only one population object derived from the second entity refers to population objects derived from the first entity. If two entities are in a one-to-one relationship, this is enforced in the two directions. Our system checks for violation of these constraints, each time the slots of a population object are updated.

2. **Object dependency enforcement.** Dependencies between entities in an EER diagram result in another kind of enforcement of population objects derived from these entities. In this case we ensure that when a population object is deleted, all population objects it refers to, that have been derived from an entity that depends on the entity from which the deleted population object is derived, are deleted also. Note that for the enforcement to be actually performed, the population objects that are candidates for deletion are not allowed to be referenced by any other population object.

3. **Method synchronization.** The EER diagram used in SelfSync is extended with operations. These operations are linked to the method bodies of the corresponding methods in the implementation objects. First, the method bodies can be edited in the EER diagram, which is automatically synchronized with the actual method bodies in the implementation objects, and vice versa. Second, we also support the possibility to "inject" behaviour before or after one or more selected operations in the EER diagram. Again, this new piece of code is automatically added in the beginning or end of the method bodies of all selected operations in the EER diagram. These *code injections* maintain their identity: at any point in time the layers of different code injections of an operation can be consulted. Each of these injections can be removed locally or in all operations where this specific injection was added.

4. **Object generations** Changing a method in an implementation object has repercussions on all population objects that have been derived from it. Since we allow changing method bodies by manipulating the corresponding operations in the EER diagram, SelfSync supports behavioural evolution of entire existing generations of population objects, steered from the EER diagram.

5. **Role modeling.** We extended SelfSync's EER model with a modeling concept for roles [4]. Modeling a role object results in corresponding implementation objects being automatically created with the structure of *warped hierarchies* [4], [5]. In these implementation objects, an arbitrary number of roles can be added or removed dynamically thanks to multiple inheritance and dynamic parent modification. The technique is based on meta-programming and Self's state inheritance mechanism called *copy-down* [6].

1. REFERENCES

[1] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[2] E. V. Paesschen, M. D'Hondt, and W. D. Meuter. Rapid prototyping of eer models. In *ISIM 2005, Hradec Nad Moravici, Czech Republic, April 2005, Proceedings*, pages 194–209. MARQ, 2005.

[3] E. V. Paesschen, W. D. Meuter, and M. D'Hondt. Selfsync: a dynamic round-trip engineering environment. In *Proceedings of the ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems (MoDELS'05), October 2-7, Montego Bay, Jamaica*, 2005.

[4] E. V. Paesschen, W. D. Meuter, and M. D'Hondt. Role modeling in selfsync with warped hierarchies. In *Proceedings of the AAAI Fall Symposium on Roles, November 3 - 6, Arlington, Virginia, USA*, 2005 (to appear).

[5] E. V. Paesschen, W. D. Meuter, and T. D'Hondt. Domain modeling in self yields warped hierarchies. In *Workshop Reader ECOOP 2004, Oslo, Norway, June 2004*, volume 3344 of *Lecture Notes in Computer Science*, page 101, 2004.

[6] Self: http://research.sun.com/self/.

[7] Together: http://www.borland.com/together/.