SelfSync: A Dynamic Round-Trip Engineering Environment

Ellen Van Paesschen[1] - Wolfgang De Meuter[2] - Maja D'Hondt[2]

[1] Programming Technology Laboratory
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
evpaessc@vub.ac.be
[2] Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq, Cédex, Lille, France
wdmeuter@vub.ac.be - maja.d-hondt@lifl.fr

Abstract. Model-Driven Engineering (MDE) advocates the generation of software applications from models, which are views on certain aspects of the software. In this paper, we focus on a particular setup which consists of a graphical data modeling view and a view on an object-oriented implementation, which can be either textual or graphical. A challenge that arizes in the context of MDE is the notion of *Round-Trip Engineering* (RTE), where elements from both views can be manipulated and thus need to be synchronized. We systematically identify four fundamental RTE scenarios. In this paper, we employ the framework of these scenarios for explaining *SelfSync*, our approach and tool for providing dynamic support for RTE. In SelfSync, the entities of the data modeling view and the corresponding implementation objects are one and the same. Additionally, we present a comparison with related work accompanied by an extensive discussion.

1 Introduction

Model-Driven Engineering (MDE) advocates generating software applications from models, which are views on certain aspects of the software. One commonly found approach is to support one or more graphical modeling views on the one hand and an implementation view on the other, which can be either textual, i.e. the actual source code, or graphical. In this paper, we focus on a particular setup which consists of a *data modeling* view and a view on an *object-oriented* implementation.

An important issue that arizes in the context of MDE is the notion of *Round-Trip Engineering* (RTE). Several definitions exist of RTE, but all boil down to the following: when there exist at least two views on a software artefact, each view can be used to manipulate the artefact and all the other views need to be synchronized accordingly [1], [6], [12], [20]. RTE often considers a setup similar to the one we outlined above. Therefore, the challenge in this setup is that both the data modeling view and the object-oriented implementation (view) can be manipulated and thus need to be synchronized. In this paper, we identify four fundamental RTE scenarios that cover the range of possible changes to both views.

We provide a very dynamic approach to RTE, where the entities of the data modeling view and the corresponding implementation objects are one and the same [17], [16]. This contrasts with other approaches, which usually employ a synchronization strategy based on transformation [12], [20], [28]. In this paper, we first present our approach and accompanying tool, SelfSync, in Section 2. We then present the four identified RTE scenarios in Section 3. Next we show how our approach and tool address these four scenarios in Sections 4 to 7. We present and discuss related work in Section 8. Finally, we conclude in Section 9.

2 SelfSync

SelfSync supports data modeling in an *Extended Entity-Relationship* (EER) diagram [4] and object-oriented programming in the prototype-based language *Self* [21], [26]. In this section we elaborate on these two parts while introducing an example (sections 2.1 and 2.2). We then explain how SelfSync is used to prototype applications rapidly (section 2.3). Finally, we describe the three views that SelfSync synchronizes during Round-Trip Engineering (section 2.4).

2.1 EER Modeling

EER diagrams consist of the typical data modeling elements, similar to *Class Diagrams* in the *Unified Modeling Language* (UML) [9]: entities (classes in the UML), attributes and operations[1] in entities, and association and inheritance relations between entities. The associations can be 1-to-1, 1-to-many, and many-to-many. There are some variants of the typical data modeling elements, such as entities and weak entities, and simple, primary and derived attributes. The EER notation we use combines existing approaches: Chen's boxes [4], the relations of the crow's feet notation and the cardinalities of [7] [2]. We use different colours to denote the differences between entities and weak entities, and between simple, primary and derived attributes. We want to stress that our new combined notation is merely a consequence of our choice of development platform.

In Figure 1 an EER model of a moderate banking system is shown. This example is used throughout the paper. A `Customer` has a primary attribute `customerID` and simple attributes `customerName`, `customerStreet` and `customerCity`. `Customer` is in a many-to-many relation with `Loan` (role `borrows`) and with `Account` (role `accounts`), and in a many-to-one relation with `Employee` (role `banker`). `Payment` is a weak entity that is dependent of `Loan`. An `Account` can be specialized into a `SavingsAccount` or a `CheckingsAccount`.

[1] We extended the standard EER diagram with operations in addition to attributes.
[2] The order of cardinalities is reversed, as in the Object Modeling Technique

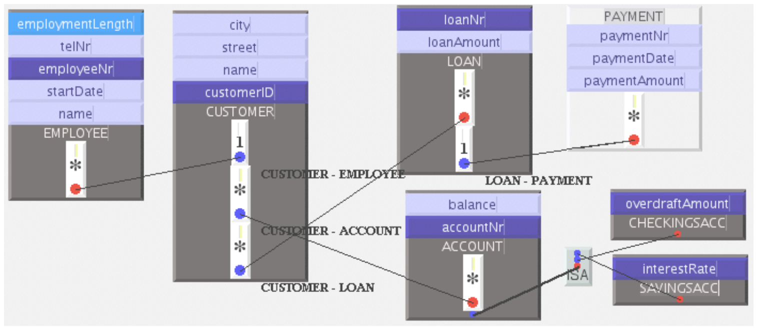

Fig. 1. An EER diagram for a moderate banking system.

2.2 Self

The object-oriented implementation language we employ is the prototype-based language Self. In general, prototype-based languages can be considered object-oriented languages without classes. As such, a *prototype* is used for sharing data between objects and new objects can be created by cloning a prototype. Self, however, introduces another programming idiom, *traits*, which share behavior among objects and let objects inherit from them, which allows for simulating classes [3]. Note that in Self everything is an object, more specifically prototypes, traits and cloned objects, which can again be prototypes.

The Self development environment provides support for visual programming using *outliners*, graphical views on objects. Objects, attributes and methods can be created and initialized using menus of the outliners. This is depicted in Figure 2 by the two boxes on the top left, labeled *Self code* and *Self outliners*.

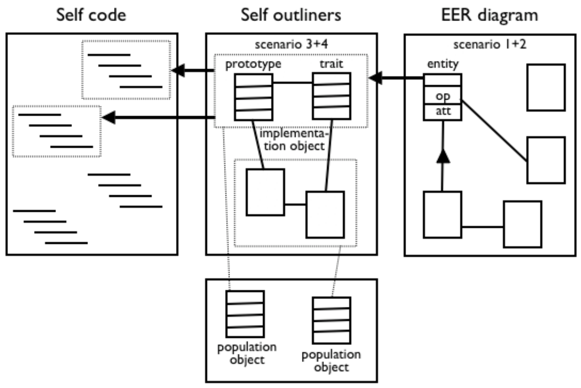

Fig. 2. The setup of our tool for supporting Round-Trip Engineering.

2.3 Two-Phased Approach

We distinguish two phases when using SelfSync, which are typically but not necessarily executed subsequently. For each phase we indicate how this setup is implemented and provide terminology that is used in the remainder of this paper.

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.

Our implementation strategy consists of generating per entity a prototype for sharing the entity's data and a traits object for sharing its behavior. This results in the following setup, again depicted in Figure 2: the top right-hand box in this figure is the *EER diagram*, whose entities are mapped to the corresponding Self outliners. Each entity corresponds to a prototype-traits pair, bounded in a dashed box. This is denoted by the arrow from the entity in the EER model to the dashed box containing prototype and traits outliners. We refer to the prototype-traits pair that implements a certain entity from the EER diagram as an *implementation object*.

The second phase of our approach is an *interactive prototyping* process[3]. 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. With the notion of a *population object* we 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. For example, when an actual `Customer` object is created and initialized, our system asks the user how many `Account` objects this `Customer` is to refer to, which can be any number or unlimited.

2.4 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:

[3] 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.

– the EER data modeling view: consists of all the information pertaining to entities (attributes and operations) as well as inheritance between entities and associations with multiplicities between entities
– a code-time implementation view: the outliners on the implementation objects, which show everything related to object-oriented programs; programmers can enrich the implementation objects with additional attribute slots, fill in the method bodies, create new implementation objects manually, etc. Note that relations in the implementation view are implicit since these occur when a certain object has one or more objects as attribute.
– a run-time implementation view: (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.

3 Round-Trip Engineering Scenarios

Round-Trip Engineering is especially crucial in the context of MDE, where multiple views of a software application can in principle be manipulated and the other views need to be synchronized accordingly [1], [6], [12], [20]. When considering a graphical model as one view and the (graphical or textual) source code as another, Round-Trip Engineering typically considers *forward* and a *backward* activities. The former consists of changing the graphical model after which the source code needs to be synchronized with the model. The latter denotes changes to the source code and subsequent synchronization steps to the graphical model.

Based on the direction of synchronization we make a distinction between the views the changes take place in: 1) the EER data modeling view and 2) the code-time object-oriented implementation view both described at the end of Section 2.4. The data modeling view represents synchronization in the forward direction, whereas the code-time implementation view represents the inverse. All elements contained in these views can *evolve*, which we use in this paper as a collective term for being created, changed or deleted.

Based on which kinds of elements evolve in a view, we make another distinction in Round-Trip Engineering: 1) changes to entities, attributes and operations in the data modeling view, and changes to implementation objects, data and method slots in the code-time implementation view and 2) changes to association and inheritance relations in the data modeling view, and changes to relations between implementation objects in the code-time implementation view.

Each of the four scenarios corresponds to a particular direction of Round-Trip Engineering and particular elements that are changed and subsequently synchronized as summarized in Table 1:

Table 1. The four scenarios that cover synchronization between a graphical data modeling view and an OO code-time implementation view.

	Entities	Relations
Data modeling view	Scenario 1	Scenario 2
OO code-time implementation view	Scenario 3	Scenario 4

scenario 1: changes to entities, attributes and operations in the data modeling view, which are synchronized in both the code-time and the run-time implementation view

scenario 2: changes to association and inheritance relations in the data modeling view, which are synchronized in both the code-time and the run-time implementation view

scenario 3: changes to implementation objects, data and method slots in the implementation view, which are synchronized in the data modeling view

scenario 4: changes to (implicit) relations in the implementation view, which are synchronized in the data modeling view

Note that in scenario 1 and 2 when operations or relations in the data model evolve, this can impact the population objects in the run-time implementation view. In Figure 2 the four scenarios are situated in the different views of the SelfSync architecture.

4 Scenario 1: Entity Evolution From Model To Code

We use the banking system EER model (see Figure 1) as an example. This model is extended to support simple insurances. We illustrate the scenario with the following steps:

1. Add two new entities `insurance` and `insurer` to the banking system model
2. Add a new attribute `policyNr` to `insurance`
3. Add a new attribute `insuredObject` to `insurance`
4. Add a new operation `checkClaim` to `insurance`

To realize the entity evolution scenario, the following actions are performed in SelfSync at code-time, by the user followed by our automated synchronization mechanism:

1. Add a new blank entity view to the EER diagram via the appropriate menu and name it `insurer`. *Synchronization steps:* First, a new blank entity view becomes graphically visual; since this is a new view on a new implementation object, a new implementation object is automatically created. This newly created implementation object contains no public data slots and an empty traits object to contain methods. The implementation object is automatically saved in the `banking` schema object. The name of the graphical entity view in the diagram is changed to `insurer`, this is propagated automatically onto the viewed implementation object.

2. Analoguously to step 1, the entity view `insurance` is added.
3. Next, we add a new attribute to the graphical entity view `insurance` via the appropriate menu, and name it `policyNr`. *Synchronization steps:* First a blank attribute (dark/light blue) becomes graphically visual inside the `insurance` entity view. Automatically, a new data slot is added to the `insurance` implementation object. The renaming is propagated to the implementation object by renaming the original data slot in the implementation object. This implies that changes to the contents of the data slot in the implementation object are not lost when the corresponding attribute in the entity view is renamed.
4. Similarly, a new operation is added to the `insurance` entity view via the appropriate menu, and is named `checkClaim`. *Synchronization steps:* First a blank operation (red) becomes graphically visual inside the `insurance` entity view. Automatically, a new method slot is added to the traits object of the `insurance` implementation object. The renaming is propagated to the traits object by renaming the original method slot. The body of the method can be viewed and edited from inside the entity view: the changes are propagated to the method body in the implementation object.

Deleting attributes and operations automatically results in deleting the corresponding data or method slot in the implementation object. Deleting an entire entity view automatically results in deleting the implementation object from the banking schema object.

By adding, removing, renaming, and changing an operation to an entity view, all run-time population objects that are created from the entity view's code-time implementation object, are affected. This is a consequence of adding corresponding method slots in the traits object of the code-time implementation objects, that are shared by the code-time implementation object as well as by all its run-time population objects.

5 Scenario 2: Relationship and Specialization Evolution From Model To Code

We use the banking system EER model (see Figure 1) as an example. This model is extended to support simple insurances. We illustrate the scenario with the following steps:

1. Specialize the entity `employee` into `insurer` in the banking system model
2. Add a new 1-to-n relation between the entities `customer` and `insurance` in the banking system model

To realize the entity evolution scenario, the following actions are performed in SelfSync at code-time, by the user followed by our automated synchronization mechanism:

1. Add a new specialization to the EER diagram from the entity view `insurer` to the entity view `employee`, via the appropriate menu. *Synchronization*

steps: Automatically, the `insurer` implementation object inherits from the `employee` implementation object. Deleting the specialization in the EER model automatically results in removing the inheritance between the two implementation objects.
2. Add a new 1-to-n relationship between the entity views `insurer` and `customer`, via the appropriate menu. *Synchronization steps:* Automatically a slot called `1_to_n_relation_insurer_customer` is added to both viewed implementation objects `insurer` and `customer`. This slot contains a reference to the other partner entity object. Deleting the relationship in the EER model automatically results in deleting the slot.

After the interactive prototyping phase a 1-to-1 or 1-to-n relationship between two entity views also results in satisfying the cardinality constraints imposed by these relations. When two entity views 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 run-time population objects in two ways. First we ensure that all run-time population objects (i.e. the clones) of the first code-time implementation object's type (i.e. the prototype) have at most one reference to run-time population objects of the second code-time implementation object's type. Secondly, we also ensure that only one run-time population object of the second code-time implementation object's type refers to run-time population objects of the first type. If two entity views are in a 1-to-1 relationship, this is enforced in the two directions. Our system checks for violation of these constraints, each time the slots of a run-time population object are updated.

Adding dependencies between two entity views results in another kind of enforcement. In this case we ensure that when a run-time population object is deleted, all run-time population objects whose corresponding entity view is dependent of the entity view of the deleted run-time population object, are deleted also.

Note that since the multiplicity and dependency information is stored in the traits objects shared by both code-time implementation and run-time population objects, changing relationships and multiplicities or dependencies in the EER diagram affects also existing run-time population objects.

6 Scenario 3: Object Evolution From Code To Model

We use the banking system EER implementation as an example. When a new code-time implementation objects is created it is installed in the schema object and its entity view becomes visual. When an entire code-time implementation object is deleted, the entity view automatically dissapears from the EER diagram. The other cases are illustrated with the following steps:

1. Add a new attribute `insurer` to the implementation object `insurance` in the banking system implementation
2. Rename the attribute `insurer` in the implementation object `insurance` to *myInsurer*

3. Add a new method `extendPolicy = ('to be implemented')` to the implementation object `insurance`
4. Change the body of `extendPolicy` in the implementation object `insurance` to `(numberOfInsuredObjects:(numberOfInsuredObjects + 1))`

To realize the entity evolution scenario, the following actions are performed in SelfSync at code-time, by the user followed by our automated synchronization mechanism:

1. Add a new data slot to the `insurance` implementation object via the main Self object menu and name it *insurer*. *Synchronization steps:* Automatically, the `insurance` entity view in the EER model is extended with a new attribute `insurer`. Note that deleting a data slot in an implementation object automatically results in deleting the corresponding attribute in the entity view.
2. Rename the `insurer` data slot in the `insurance` implementation object to *myInsurer* by double-clicking it. *Synchronization steps:* Automatically, the `insurer` attribute in the `insurance` entity view in the EER model is renamed to `myInsurer`.
3. Add a new method slot `extendPolicy = ('to be implemented')` to the implementation object `insurance`. *Synchronization steps:* Automatically, the `insurance` entity view in the EER model is extended with a new method attribute `extendPolicy`. When this method body is viewed or edited in the `insurance` entity view in the EER model via the operation menu, the text `'to be implemented'` becomes visible. Note that deleting a method slot in an implementation object automatically results in deleting the corresponding operation in the entity view.
4. Change the body of `extendPolicy` to `(numberOfInsuredObjects:(numberOfInsuredObjects + 1))` by clicking the method body symbol in the `insurance` implementation object. *Synchronization steps:* When the method body of the `extendPolicy` operation is viewed or edited in the `insurance` entity view in the EER model via the operation menu, the new body `(numberOfInsuredObjects: (numberOfInsuredObjects + 1))` becomes visible.

When these changes are applied to run-time population objects, the entity views of the corresponding code-time implementation objects are not affected.

7 Scenario 4: Reference and Inheritance Evolution From Code To Model

We use the banking system EER implementation as an example. We illustrate the scenario with the following steps:

1. Change the contents of the attribute `myInsurer` in the implementation object `insurance` to contain the `insurer` implementation object.
2. Create a new child of the implementation object `insurance`

To realize the entity evolution scenario, the following actions are performed in SelfSync at code-time, by the user followed by our automated synchronization mechanism:

1. Set the contents of the `myInsurer` data slot in the `insurance` implementation object to contain the `insurer` implementation object either via the appropriate Self menu, a user action, or at run-time. *Synchronization steps:* Automatically, a one-to-one relationship link is drawn between the `insurance` entity view and the `insurer` entity view in the EER model, given no 1-to-1 link is drawn between them currently. This synchronization is performed dynamically: when we manually remove the 1-to-1 link in the EER model, it is automatically re-drawn, each time the Self system updates the slots of the `insurance` implementation object and discovers that it (still) contains a reference to the `insurer` implementation object
2. Create a new child of the implementation object `insurance` via the Self main menu. *Synchronization steps:* Automatically, a new entity view is added to the EER model. Simultaneously, a new implementation object has been created, inheriting the slots of the `insurance` implementation object. Next a new "is-a" link is drawn between the new entity view and the `insurance` entity view in the EER model.

When these changes are applied to run-time population objects, the entity views of the corresponding code-time implementation objects are not affected.

8 Related Work and Discussion

We situate our approach in the intersection of three domains: Round-Trip Engineering, visual programming and agile development. We discuss Borland's Together (section 8.1) and the Naked Objects approach (section 8.2), respectively, as representatives for the first two domains. A concrete instance of agile modeling and other related work can be found in Section 8.3. In Section 8.4 we compare SelfSync to the related approaches.

8.1 Round-Trip Engineering

The state-of-the-art in RTE includes application such as Rational XDE [25], Borland Together [28], and FUJABA [22]. One of the leaders in this domain is Borland's Together. This set of commercial tools provides support for modeling, designing, implementing, debugging, and testing applications. The synchronization mechanism between UML class diagrams and implementation is realized by the *LiveSource* technology. More specifically, the implementation model (i.e. the source code) is parsed and rendered as two views: a UML class diagram and in a formatted textual form. LiveSource is in fact a code parsing engine. The user can manipulate either view and even the implementation model. However, all user actions are translated directly to the implementation model and then translated back to both views. We discuss the relation to SelfSync in Section 8.4.

Other related work in RTE, is mostly concerned with characterizing RTE rather than providing concrete tool support. In [1], RTE is described as a system with at least two views that can be manipulated. Applying the inverse transformation f^{-1} on a view that is transformed using f, should again yield the same view. The Automatic Roundtrip Engineering [1] approach advocates the automatic derivation of this inverse transformation function based on the original transformation function. Our approach is based on Model-View-Controller (MVC) [10]. Therefore, a change initiated in a view is not actually performed in the view, but in the underlying implementation element, which results in the relevant views being automatically updated.

In [20] RTE is connected to inconsistency handling. In SelfSync, MVC makes inconsistency handling superfluous since no inconsistencies are introduces for the same reasons explained above. The same work states that RTE is not merely a combination of forward and backward engineering since there is not always a one-to-one mapping between similar elements in different views. In contrast, we deliberately assume such a mapping in order to automate the synchronization bidirectionally.

8.2 Visual Programming

At the level of visual programming we compare SelfSync to the Naked Objects [18], [24] approach that also applies MVC, but in one direction: from code to model. Building a business system consists solely of defining the domain business objects (i.e. code-time implementation objects) in Java, which immediately are made visible to and manipulable by the user in a business object model. The Naked Objects Java framework represents classes as icons and uses Java interfaces to determine the methods of any business object and render them visible on the screen by means of a generic viewing mechanism.

With respect to run-time support, the user can visually create new business objects, specify their attributes, add associations between them, or invoke methods on them. The ready-to-use objects are visually represented and automatically created and updated in the Java program.

8.3 Other Related Work

In this section we describe other related work that is not discussed in detail but included for completeness.

Since the late eighties, it has been encouraged to combine (E)ER models and object-orientation (OO) [5], [15]. Various approaches and techniques exist for translating EER into object-orientation [8], [14], [11], [13]. Such mappings can be used in the domain of object-relational (O/R) mappers [29], [23], [27]. These tools generate an object implementation from a data model such as (E)ER, and possibly support synchronization of both models. Some of them generate code to enforce constraints on relationships and dependencies between implementation objects, based on the data model. However, these applications do not consider behavior at the level of the datamodel.

Finally, since SelfSync allows rapid prototyping, we consider a concrete example of agile modeling [2]. In this case the stress is less on synchronization and more on rapid prototyping and testing. In [2] applying eXtreme programming to modeling is realized by making UML diagrams executable. Different UML diagrams are translated into Petri-Nets and interpreted by a Petri-Net engine. This engine can be seen as a *UML Virtual Machine* and contains a Java parser. The precise evolution support in this case depends on the environment in which the UML models are created and in which the UML Virtual Machine is integrated. As is, as far as we know, no support for RTE is provided.

8.4 Discussion

The ensuing discussion compares SelfSync to the related work introduced above. We distinguish four tracks: (1) UML versus EER, (2) forward RTE support (scenarios 1 and 2), (3) backward RTE support (scenarios 3 and 4), and (4) run-time RTE support.

UML versus EER. There is an almost religious discussion between the (E)ER and the UML communities as to which approach is better. Typical claims are that (E)ER modeling is more formally funded but that the UML is more open [19], [9]. In our work, however, the use of EER does not exclude the transfer of our conceptual results to an UML-based context. In this paper, we describe Round-Trip Engineering on the data modeling level in terms of entities, attributes and operations, and association and inheritance relations. These EER modeling elements have equivalent modeling elements in Class Diagrams of the UML.

Forward RTE Support. Forward RTE support, embodied by scenarios 1 and 2, is provided by Together's LiveSource. Naked Objects only provides backward RTE support and some run-time support. We first mention the similarities with SelfSync, and then discuss the differences.

Similarities. Both LiveSource and SelfSync provide forward RTE support when evolving the following data modeling elements: entities or classes, attributes, operations, relations and specializations. In LiveSource this is supported by first propagating the changes to the implementation model and then updating the views, i.e. the class diagram and the formatted source code. In SelfSync this is supported because these data modeling elements and the corresponding implementation elements are in reality the same. Although LiveSource uses Java as implementation language, which is class-based, and SelfSync uses Self, originally a prototype-based language, this is not the fundamental difference here. Indeed, a mapping needs to be devized between data modeling and implementation elements, whether this is entities on classes or entities on prototypes and traits.

Differences. The only other support offered for cardinalities and dependencies in Together is not inherent to LiveSource but a consequence of the fact that Together supports the technology of Enterprise Java Beans (EJB), the component model for J2EE. EJB 2.0's container-managed persistence specification allows fine-grained control over entity bean relationships. When we add an association between two container-managed entity beans in a class diagram, parameters such as relation name and multiplicities need to be supplied. Automatically a new container-managed relationship is created. To the best of our knowledge, the actual enforcement of cardinalities is only limited and only due to the static typing that is provided by Java. In particular, an attribute cannot contain an object of another type than declared with the attribute. Although we employ a dynamically typed implementation language, we provide this level of enforcement, and more. For example, we also ensure that only one population object of a certain type refers to another population object if the latter is allowed to have a single reference to the first type.

To enforce dependencies using EJB a `cascade-delete` XML tag is used in the description of relationships: when the entity bean is deleted, all its dependents it is in a relationship with, are deleted as well. The difference with SelfSync is that, although we provide similar support for enforcing dependencies, we provide it in the context of an RTE tool and not solely in the implementation technology.

Backward RTE Support. Backward RTE support, embodied by scenarios 3 and 4, is provided by Together's LiveSource and Naked Objects. We first mention the similarities with SelfSync, and then discuss the differences.

Similarities. LiveSource and SelfSync provide backward RTE support when evolving the following object-oriented implementation elements: classes in the class-based approaches and prototypes or traits in our prototype-based approach, attributes, methods, references between classes or prototypes, and inheritance between classes or traits. Naked Objects support the same except for references and inheritance. In Together's LiveSource and SelfSync the backward RTE support is enabled in an analogous way to the forward RTE support. Naked Objects' support for backward RTE is similar to ours, i.e. through MVC, but an additional compilation step of the changed Java code is necessary.

Differences. LiveSource and SelfSync do not differ in the kind of backward RTE support provided, only in the internal strategy, as explained earlier. Naked Objects, however, does not provide support for synchronizing evolving references and inheritance in the Java code.

Run-time RTE Support. Run-time RTE support only makes sense in the forward direction, more specifically changes to the data model are reflected in the run-time population objects. It is nonsensical to automatically synchronize a data model when changes are made to instantiated and initialized objects. Especially since most statically typed, class-based implementation languages such as Java would restrict the possible changes that can be made based on the source code. In dynamically typed languages or prototype-based languages (or both) there are less restrictions to changing the run-time population objects, but even in these cases it is undesirable to reflect them in the data model.

Only SelfSync provides full forward run-time RTE support. This means that evolution of attributes, operations, relations, specializations, cardinalities and dependencies are reflected in the run-time population objects. The main reason SelfSync supports this is primarily due to the dynamic character of the implementation language, Self. Using another dynamically typed language, such as Smalltalk, would allow us to achieve similar results. In a context where a statically typed implementation language is used, such as Java, one would have much less flexibility in changing the data model (or even the source code directly) and synchronizing the corresponding ready-to-use population objects. Another reason why this is supported in SelfSync, is that Self separates state sharing and behavior sharing.

In the Naked Objects approach there is only support at the level of adding associations between instantiated business objects, which is reflected in the corresponding Java objects.

9 Conclusion

This paper presents three contributions with respect to Round-Trip Engineering (RTE) in a particular Model-Driven Engineering setup consisting of a data modeling view and a view on an object-oriented implementation. First of all, we identify and describe fundamental set of four Round-Trip Engineering scenarios. These scenarios distinguish between direction, from model to code or vice versa, and kind of elements that evolve, entity views and implementation objects and their elements on the one hand, or relations between them on the other. A second contribution is our tool, SelfSync, which provide very dynamic support for these four RTE scenarios, not only at code-time but in relevant run-time situations as well. This is a direct result of the entities of the data modeling view and the corresponding implementation objects being one and the same in SelfSync. Finally, we describe related work and present a comparison accompanied by an extensive discussion.

References

1. U. Assman. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82.
2. M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. pages 175–189, 2001.
3. C. Chambers, D. Ungar, B.-W. Chang, and U. Holzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):0–, 1991.
4. P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
5. P. P. Chen. Er vs. oo. In *Entity-Relationship Approach - ER'92, 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, October 7-9, 1992, Proceedings*, volume 645 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 1992.

6. S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal? In *UML'99, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 630–644. Springer, 1999.
7. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley World Student Series, 3 edition, 1994.
8. J. Fong. Mapping extended entity relationship model to object modeling technique. *SIGMOD Record*, 24(3):18–22, 1995.
9. M. Fowler and K. Scott. *UML distilled: a brief guide to the standard object modeling language.* Addison-Wesley Longman Publishing, Boston, MA, USA, 2000.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.
11. M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the er approach in an oo environment. In *Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings*, volume 823 of *Lecture Notes in Computer Science*, pages 376–389. Springer, 1993.
12. A. Henriksson and H. Larsson. A definition of round-trip engineering. Technical report, Linkopings Universitet, Sweden, 2003.
13. R. Herzig and M. Gogolla. Transforming conceptual data models into an object model. In *ER'92, Karlsruhe, Germany, October 1992, Proceedings*, volume 645 of *Lecture Notes in Computer Science*, pages 280–298. Springer, 1992.
14. C.-T. Liu, S.-K. Chang, and P. K. Chrysanthis. Database schema evolution using EVER diagrams. In *Advanced Visual Interfaces*, pages 123–132, 1994.
15. S. B. Navathe and M. K. Pillalamarri. Ooer: Toward making the e-r approach object-oriented. In *Entity-Relationship Approach: A Bridge to the User, Proceedings of the Seventh International Conference on Enity-Relationship Approach, Rome, Italy, November 16-18, 1988*, pages 185–206. North-Holland, 1988.
16. E. V. Paesschen, M. D'Hondt, and W. D. Meuter. Rapid prototyping of extended entity relationship models. In *ISIM 2005, Hradec Nad Moravici, Czech Republic, April 2005, Proceedings*, pages 194–209. MARQ, 2005.
17. 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.
18. R. Pawson and R. Matthews. Naked objects: a technique for designing more expressive systems. *ACM SIGPLAN Notices*, 36(12):61–67, Dec. 2001.
19. K.-D. Schewe. UML: A modern dinosaur? In *Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, Saariselkä (Finland), 2000.* IOS Press, Amsterdam, 2000.
20. S. Sendall and J. Kuster. Taming model round-trip engineering. In *Proceedings of the Workshop on Best Practices for Model-Driven Software Development at OOPSLA 2004, Vancouver, Canada*, 2004.
21. D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87, Orlando, Florida, USA*, pages 227–242, New York, NY, USA, 1987. ACM Press.
22. Fujaba: http://wwwcs.uni-paderborn.de/cs/fujaba/.
23. Llblgen: http://www.llblgen.com/.
24. Naked objects framework: http://www.nakedobjects.org.
25. Rational: http://www-306.ibm.com/software/awdtools/developer/rosexde/.
26. Self: http://research.sun.com/self/.
27. Simpleorm: http://www.simpleorm.org/.
28. Together: http://www.borland.com/together/.
29. Toplink: http://www.oracle.com/technology/products/ias/toplink/index.html.