Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

Advanced Round-Trip Engineering

An Agile Analysis-driven Approach for Dynamic Languages

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Ellen Van Paesschen

Academiejaar 2005 - 2006

Promotoren: Prof. Dr. Theo D'Hondt en Dr. Maja D'Hondt

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

Abstract

Concepts and processes that are typical for human problem solving, such as approaching a problem from different viewpoints or different levels of abstraction, and iterative and incremental trial and error thought processes underlie the state-of-the-art in software engineering methods. Model-Driven Development (MDD) considers multiple models at different levels of abstraction and transforms them in a semi-automated way, while Agile Development advocates a lightweight, change-oriented development cycle. Agile MDD aims at combining MDD and its voluminous transformations between complex models, with Agile Development that considers a highly iterative and incremental development process.

The use of multiple, manipulable views on a system requires that these views are kept consistent. Keeping different evolving artifacts in software engineering consistent is a key factor in Round-Trip Engineering (RTE) as it combines forward and reverse engineering activities. Current RTE tools however implement insufficient agility to support Agile MDD practices. Therefore, we propose a new approach to RTE, that can be supported by the next generation of RTE tools that meet the specific needs of Agile MDD. *Advanced Round-Trip Engineering* (ARTE) extends traditional RTE by considering a high-level view, by applying an Agile Development cycle, and by integrating run-time objects in the RTE process.

In this dissertation, we elaborate on an instance of ARTE, *myARTE*. First, myARTE includes an analysis view, to which role modelling is added, that is kept consistent with an implementation view. Second, myARTE approaches RTE from an agile, lightweight perspective by continuously realising high-level, incremental transformations between the analysis and the implementation view that represent a common model, in combination with a rapid prototyping phase. Finally, myARTE considers the run-time object generations that are created during rapid prototyping. These object generations are rescued between different iterations of the RTE process and are constrained by multiplicities and dependency relationships in the analysis view.

Using the high-level prototype-based programming language Self for the implementation view in myARTE not only allows us to implement the myARTE approach in a concise way, but moreover enables to map the three criteria of myARTE to twelve finer-grained language mechanisms in object-oriented programming languages, that are required for implementing myARTE tools. From a detailed analysis of the presence of building blocks for these language mechanism in object-oriented programming languages, we observe that dynamically-typed prototype-based programming languages with a powerful meta-programming mechanism that combine the high flexibility of "everything is an object" with the efficiency of an explicit mechanism for sharing behaviour, provide basic language constructs for building the required language mechanisms.

myARTE employs an EER model in its analysis view, that applies existing EER notations in combination with a role modelling concept that allows to express mutually exclusive roles. A warped hierarchies implementation pattern presents a general solution to implement roles that are modelled in the analysis view. We have implemented a graphical drawing editor that represents this extended model, and that is integrated in the Self programming environment.

SelfSync is a proof-of-concept tool that implements our myARTE instance. The SelfSync tool is validated in terms of scalability with the extensive and mature TELEBIB domain model that defines the standard for the Belgian insurance business. Finally, the myARTE approach is extended with Advanced Method Synchronisation (AMS) that allows methods and their bodies to be visualised in the data modelling view and to be synchronised in the implementation and population views. The presence of language constructs that are required to implement AMS is investigated in various object-oriented programming languages. The SelfSync tool is extended with AMS: the EER view includes operations that correspond to method slots in the implementation view. We show that AMS can one day lead to an approach for creating Executable Models and to support for Visual AOP.

Abstract

Concepten en processen die typisch zijn voor het menselijk probleem-oplossen zoals een probleem benaderen vanuit verschillende abstractieniveaus, en iteratieve en incrementele "trial-and-error" denkprocessen liggen mee aan de basis van de huidige software engineering methoden. Model-Driven Development (MDD) beschouwt verschillende modellen op verschillende abstractieniveaus, die semi-automatisch getransformeerd worden in uitvoerbare code, terwijl Agile Software Development een meer light-weight change-georiënteerd ontwikkelingsproces ondersteunt. Agile MDD is een eerste poging om de volumineuze model transformaties tussen complexe models in MDD te combineren met het uiterst iteratief en incrementeel ontwikkelingsproces van Agile Development.

Het gebruik van verschillende aanpasbare *views* op een systeem vereist dat deze views consistent zijn. Verschillende evoluerende software artifacten consistent houden speelt een cruciale rol in Round-Trip Engineering (RTE) aangezien dit proces Forward Engineering en Reverse Engineering combineert. De huidige RTE tools bevatten te weinig agiliteit om Agile MDD methoden te ondersteunen. In deze thesis introduceren we een nieuwe methode voor Agile MDD die kan ondersteund worden door een nieuwe generatie van RTE tools. *Advanced Round-Trip Engineering* (ARTE) is een evolutie van de huidige RTE aanpak, die een hoog-niveau view beschouwt, een agile ontwikkelingsproces ondersteunt en run-time objecten in het RTE proces betrekt.

In deze thesis werken we een bepaalde instantie van ARTE uit die we myARTE noemen. Ten eerste beschouwt myARTE een analyseview dat het modelleren van rollen ondersteunt, en dat gesynchroniseerd wordt met een implementatieview. Ten tweede benadert myARTE RTE van een agile lightweight perspectief door hoog-niveau incrementele transformaties te realiseren tussen het analyseview en het implementatieview, in combinatie met een interactief rapid prototyping proces. Ten derde groepeert myARTE gelijkaardige run-time objecten in generaties die mee gesynchroniseerd worden bij veranderingen in het implementatieview en die beperkt worden door de multipliciteiten en afhankelijkheidsrelaties in het analyseview.

De hoog-niveau prototypegebaseerde taal Self gebruiken voor het implementatieview laat niet enkel toe om de myARTE aanpak beknopt en comfortabel te implementeren, maar zorgt er ook voor dat de drie myARTE criteria geprojecteerd kunnen worden op twaalf taalonafhankelijke taalmechanismsen die typisch zijn voor een myARTE implementatie.

Een gedetailleerde analyse over hoe deze taalmechanismen gebouwd kunnen worden in objectgeoriën- teerde programmeertalen leert ons dat dynamisch getypeerde, prototypegebaseerde talen met een krachtig reflectiemechanisme, die de flexibiliteit van "alles is een object" combineren met een expliciet mechanisme voor gemeenschappelijk gedrag, kant-en-klare taalconstructies ondersteunen die gebruikt kunnen worden als bouwstenen voor de myARTE taalmechanismen.

myARTE gebruikt een Extended Entity-Relationship model in het analyseview dat bestaande EER notaties uitbreidt met een modeleerconcept voor rollen dat disjunctie kan uitdrukken. Een nieuwe implementatietechniek voor Warped Inheritance Hierarchies kan gebruikt worden als een algemene oplossing voor de rollen in het analyseview. Dit uitgebreid EER model is geïmplementeerd in een grafische tekeneditor in de Self omgeving.

SelfSync is een proof-of-concept tool die de myARTE aanpak ondersteunt. SelfSync is gevalideerd met het uitgebreide TELEBIB domeinmodel dat de standaard definieert voor de Belgische verzekeringsindustrie.

De myARTE aanpak en SelfSync zijn uitgebreid met Advanced Method Synchronisatie (AMS). Deze uitbreiding laat toe dat methoden en hun bodies zichtbaar worden via operaties in het analyseview, en gesynchroniseerd worden in het implementatieview en het run-time view. AMS kan aanleiding geven tot Executable Models en Visueel Aspect-Oriented Programming (AOP).

Acknowledgments

First of all I would like to thank Theo D'Hondt for giving me the opportunity to become a researcher at his lab and for allowing me to choose my own research subject.

I owe an enormeous "thank you" to Maja D'Hondt, who guided me through all the steps of my PhD in such a way that nobody can compare to her. The same goes for Wolfgang De Meuter whose original ideas often inspired my research. They are both more to me than "just" colleagues.

I thank the other members of my PhD committee: Jean-Marc Jézéquel, David Ungar, Olga De Troyer and Viviane Jonckers.

I also thank the "Instituut voor Wetenschap en Technologie" (IWT) for providing me with a doctoral grant in the first four years of this research.

There are a number of people who took over my responsibilities so I could concentrate on writing this dissertation and still got paid: thank you, Jessie De Decker, Brecht Desmet, Johan Fabry, Stijn Mostinckx and Tom Van Cutsem.

Jorge Vallejos did a great job in managing the video conference during my private defense. Thanks, Jorge! I also thank Roel Wuyts and Pascal Costanza who have been proof-readers for different chapters of this dissertation.

Thanks to all the other people that make the Prog lab both a challenging and an amicable place to work: Andy Kellens, Coen De Roover, Dirk Deridder, Dirk van Deun, Elisa Gonzalez Boix, Isabel Michiels, Johan Brichau, Kris Gybels, Linda Dasseville, Peter Ebraert, Sofie Goderis and Thomas Cleenewerck.

My friend David Verhaeghe made some original illustrations for this dissertation and I would like to thank him for this.

I owe a big thank you to my parents Liliane Couvreur and Guido Van Paesschen for raising me with an overdose of love and attention, and for supporting me to study until I was almost 30.

Last but not least I would like to say thank you to my life partner Mario Vandeput, for being his enjoyable self and for enduring me with a lot of patience in the last months of writing this dissertation. I dedicate this work to him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Context and Statement

Human thinking and problem solving can be represented in terms of representational structures in the mind and computational procedures that operate on those structures [108]. Most work in cognitive science assumes that the mind includes mental representations analogous to computer data structures, such as concepts, rules and analogies, in combination with computational procedures similar to computational algorithms.

Currently software engineering tools lack cognitive support [121]. From a cognitive perspective, software development tools should aid developers by participating in their thinking and work [121], as opposed to the traditional goal of software engineering tools where the focus is on "enabling" instead of helping to develop a new system.

We believe that a new approach for building sophisticated software tools, that realizes support for specific cognitive processes by considering multiple views on a system under development, together with a highly iterative and incremental development cycle, can play an important role in solving the current lacking tool support for the state-of-the-art in software development approaches.

1.1.1 Multiple Views on Software Systems

Minsky states that the process of building the problem space during human problem analysis can be represented with frames [78]. The perception dynamics, i.e. the dynamics of frame development during the process of understanding a situation, is based on framesets:

"The different frames of a system describe the scene from different viewpoints, and the trans-

> *formations between one frame and another represent the effects of moving from place to place. Different frames correspond to different views, and the names of pointers between frames correspond to the motions or actions that change the viewpoint."*

The problem-solving space can be subject to transformations as the primary purpose in problem solving should be to increase the understanding of the problem space and to find representations within which the problems are easier to solve. The focus is more on getting information for this reformulation, not on finding solutions [78]. When people try to understand a description of a situation, the representations they build during the process of understanding are arranged in layers. Each representation is laid on the previous one and modifies it [96]. When the description of a situation is considered, the details of the situation's representation are abstracted. Only the core of the representations remain. Additional detail or new knowledge can be "developed" later, in ways that differ according to different situations.

In contrast to the popular acceptance of the frame structure in object-oriented programming languages, Minsky's analysis of perception dynamics was generally ignored or simply misunderstood [96]. Traditional software engineering methods such as the Waterfall model are used to manually refine an abstract analysis model into a more technically detailed design model and finally into a working implementation.

It is however possible to establish multiple views on a software system by specifying it from different *viewpoints* [85]. A viewpoint on a system represents a perspective that focuses on specific concerns. This allows us to suppress details to provide a simplified model having only those elements related to the concerns of the viewpoint. A *view* (or *viewpoint model*) of a system is a representation of the system from the perspective of a viewpoint. The different views can represent different levels of abstraction where each level of abstraction will be a different viewpoint of this system, as well as the same level of abstraction, for example when the viewpoints of different end-users on the system are considered. As a consequence the model of a complete system can be divided into different sub-models which will specify different parts of the system under development.

Recent tendencies in software engineering take such multiple views into account: *Model-Driven Development* (MDD) [120], [61] involves the use of domain-specific languages for creating models that express application structure or behaviour for the problem domain at different levels of abstraction. The models are subsequently transformed into executable code by a sequence of *model transformations* [64]. A part of MDD is concerned with the representation of the models that are created during software development. Approaches that follow the MDA standard, for example, consider three views with three different levels of abstraction: a computation-independent level, a platform-independent level, and a platform-specific level. Each view can contain multiple models that are constructed with the help of different languages.

The use of multiple views in software development activities requires that each model is kept consistent with the others it is related to. The ability to maintain the consistency of multiple, changing software artifacts in software development environments and tools is a key factor in what is referred to as *Round-trip Engineering* (RTE). RTE involves forward and reverse engineering steps such that software artifacts, such as, code and graphical models, become synchronized at certain points in time [95]. Where forward engineering deals with generating one or more software artifacts from an input artifact, where the generated artifact is closer to the final deployable system, reverse engineering involves generating one or more software artifacts that abstract certain details of the input artifacts, thereby trying to recover any information lost in the forward step [95]. Both forward and reverse engineering steps involve transforming one or more artifacts into one or more other artifacts. This is often realised by a *"single, isolated transformation, where any information in the target artifact is not considered, and typically a new artifact is created, possibly replacing the previous version, if one exists"* [95].

Some characterize RTE as an application of mathematical domain transformation. In that case RTE systems are defined as systems where the result of forward engineering a design that is reverse engineered from a product, is identical to the original product [55]:

> *Let D be the design of the product P, let r be a reverse-engineering procedure such that r(P) = D*
> *and let g be a product generation procedure such that g(D) = P.*
> *Iff g(r(P)) ≡ P (and thus r(g(D)) ≡ D) then g,r is a round-trip engineering system.*

The state-of-the-art in RTE research includes Automatic Roundtrip Engineering (ARE) [6] where the inverse of the aforementioned domain transformations are derived automatically. A special case of ARE is Automatic Model View Controller Engineering (MVARE) [68] where the well-known Model-View-Controller (MVC) [47] design pattern is applied to software construction. The "model" is the system under development while the views are the different viewpoints on the system. The controller is responsible for maintaining relations between views and model, and for ensuring that every change in every view is propagated to the other views and to the model.

Some sophisticated RTE tools apply model-to-model transformation since they are considered beneficial for considering different views on a system and for synchronising them [29]. MDD approaches, such as ATOM [126] and GreAT [1] realise model-to-model transformations that translate beteen source and target models that can be instances of different or the same meta-model [29]. Some of these approaches generate intermediate models in order to bridge large abstraction gaps between different kinds of models.

Tool support is provided by both open-source and commercial tools, such as OptimalJ [143] and XDE [144]. Other RTE tools, such as Together [150] apply a higher-level MVC-based approach for keeping views consistent. More precisely, changes to a visual diagram or to the source code are first translated and realised in a common underlying model and then translated in the different views.

1.1.2 An Iterative and Incremental Development Cycle

The mental process created by modellers and developers during design activities can be characterised by a number of steps [43]:

1. A a mental model of a proposed solution to the problem is constructed.
2. The mental model is mentally executed: a simulation on the model is run to see if it solves the problem.
3. If this is not the case (usually because the model was too simple), the inadequate model is "played" back against those parts of the problem to identify where it failed. Next the mental model is enhanced in the appropriate areas.
4. The first three steps are repeated until there exists a mental model that appears to solve the problem.

The resulting mental process is a very rapid, iterative process of fast trial and error actions. The mind forms a solution to the problem, knowing that it will be inadequate because the mind is not yet able to fully grasp all the facets of the problem. That problem solution, the mind knows, must be in the form of a model, because it is going to be necessary to try sample input against the model, run a quick simulation (inside the mind) on the model using the sample input, and get sample outputs (still inside the mind) from that simulation [43].

As a result of the highly iterative and incremental mental process of modellers and developers, rapid modelling and simulation are crucial in software development. Another key factor is the ability to propose solutions and allow them to fail. Software development is seen as opportunistic, concrete, and necessarily iterative. Modellers and developers typically work toward partial solutions for subsets of requirements, using prototypes to evoke further requirements and to reformulate the goals and constraints of the problem [21]. By providing techniques to quickly construct, evaluate, and change partial solutions, prototyping becomes a fulcrum for system development.

Agile Software Development [27] stresses a highly iterative and incremental development cycle and rapid prototyping in order to anticipate "change" during the different phases of software engineering. There are several reasons why requirements change frequently during software development, such as new required functionality, misunderstandings by developers or end-users, or a lack of understanding of the problem domain. Several software development methods try to anticipate rapidly changing requirements by employing a more lightweight approach to software engineering. This is achieved by prioritizing issues such as flexibililty, end-user interaction, productivity, and individuality. The commonalities of these methods' solutions are gathered in Agile Software Development [27], denoting the ability to "change quickly and easily". This approach to software engineering focuses on various short iterations of development to anticipate changing software requirements. Although Agile Software Development is new and has its roots in industry, its importance rises in academic research. Currently the validity of agile methods as the new "best practice" is being researched. There already exist a number of frequently applied agile methods such as *Rapid Prototyping*, *Extreme Programming* (XP) and *Adaptive Software Development*.

Agile MDD combines an agile iterative and incremental development cycle with the multiple views on a system that are typical for MDD. The kind of models to use is not specified and code is written progressively in step with the models [4, 5]. Agile MDD promotes an evolutionary approach, in which implementation occurs iteratively and incrementally. Development starts with a small part of modelling, followed by a significant amount of coding, followed by a large number of short iterations of model changes and code changes. The *Agile Models* that are initially created are just "barely good enough", i.e. sufficient for the task at hand. Agile Models merely capture independent aspects of the system being developed. Design efforts are divided between modelling and coding, with the majority of the design being performed as part of the implementation efforts. *Executable Models* are artifacts that can be employed in each phase of the Agile MDD life cycle. Since they can be modified, run and tested in short incremental, iterative cycles, Executable Models can function as a working prototype of the system being developed, that evaluates the validity for the represented abstract diagram.

There already exist a number of practices for Agile MDD. *Agile MDA* [74] seconds the idea that code and Executable Models are operationally the same. This approach depends on *Executable UML* [75] that defines execution semantics for a subset of UML with the help of the Object Constraint Language (OCL) [142]. From an Agile Modelling perspective, there exist three practices to Agile MDD [5]. The most frequently used is *Agile Manual Modelling* (AMM), where inclusive models are created using simple tools and the implementation is created "manually". A second approach is *Agile Computer-Aided Software Engineering* (ACASE) that uses CASE tools to automatically derive a working implementation from a manually refined model. Thirdly, there is an approach to the aforementioned Agile Model-Driven Architecture (AMDA) [74], that employs modelling tools to transform different kinds of models, and to automatically generate a working implementation. These last two approaches are significantly less frequently used than the first one.

We believe that the low success of Agile MDD practices that involve an automatic transformation, as opposed to the popularity of Agile Manual Modelling, is a consequence of the fact that current tool support is not suited for Agile MDD. We believe that Agile MDD practices require a new generation of RTE tools that are adapted to the specific needs of Agile MDD.

The shortcomings and requirements for this new kind of RTE tools, are introduced in the following section.

1.1.3 Shortcomings of RTE Tool Support for Agile MDD

We state that the current RTE tools underperform for Agile MDD in the following three ways:

- **High-level Models are Not Synchronised**: The views that are considered in the state-of-the-art in RTE tools and that are kept consistent with the aforementioned model-to-model transformation often exhibit – even in a very preliminary stage – a certain amount of technical detail. This makes it hard for end-users to totally comprehend the models that are created in these views and decreases end-user interaction during the Agile development process. This also implies that modelling constructs such as Roles [114] that are typically encountered early in the software development cycle, such as during the requirement analysis phase, are not available as first class modelling concepts in MDD.

- **No Agile Life Cycle Support**: During Agile MDD it is quite common to model storm[1] for several minutes, during which (a part of the) new requirements are integrated. Model storming

[1] The activity of some people together creating models with simple tools such as a whiteboard.

is then followed by coding, based on practices such as testing or refactoring, for several hours or days. The state-of-the-art in RTE tools are intended for generative MDD that creates sophisticated models that can automatically be transformed with sophisticated model-to-model transformations into different models in order to reflect the realities of various deployment platforms [111]. As Agile MDD focuses on the thought process that rises prior to the actual implementation [111] these tools are often too complex for the task at hand, such as short model storming or a small amount of coding that are typical for one iteration in the agile development cycle.

Moreover, an agile development cycle requires an advanced synchronisation strategy that anticipates changes to a high-level model for which there already exist pieces of a corresponding implementation, that should survive subsequent model transformations. Most RTE tools do not provide such synchronisation support, as traditionally code is only derived after extensive modelling. Additionally, few of these RTE tools support rapid prototyping and test simulation.

- **The Run-time View is Neglected**: For testing purposes during the implementation phase, Agile MDD advocates rapid prototyping any model that was developed so far, even at a very preliminary stage. Current RTE tools however, do not link directly to a semi-automatic rapid prototyping or testing process and especially lack this kind of support for preliminary, incomplete high-level models. Moreover, quickly changing requirements sometimes results in the necessity for a system to be adapted at run-time. Current RTE tools do not provide support to influence run-time objects, steered from a high-level model.

1.2 Goal: Advanced Round-Trip Engineering (ARTE)

The goal of this dissertation is to provide a new approach to RTE to be supported by the next generation of RTE tools that are suitable to be employed in Agile MDD. We define *Advanced Round-Trip Engineering*, or ARTE, as a substantial evolution of existing RTE. ARTE considers multiple views on one model and is characterised by the following three criteria:

- **ARTE includes a High-Level View**: ARTE has to support multiple views. In order to increase the similarity with human problem solving and stimulate end-user interaction we require one of the views to be as close as possible to how humans experience a problem domain in a preliminary stage: *high-level* and sufficiently abstract to be comprehensible for end-users.

- **ARTE is Agile**: ARTE has to support a highly iterative and incremental development cycle. This can be achieved by supporting two optimisations of the traditional transformations used in RTE. First, ARTE has to support *incremental transformations* [95]: only the changed elements are transformed, rather than all artifacts. Additionally, ARTE has to support *change-preserving transformations*, as required in [95]: *"information in the target artifact has to be preserved and not lost on the return trip"*. We refer to transformations that support both optimisations as *optimal transformations*.

- **ARTE is Dynamic**: ARTE has to consider a *run-time view* on the system under development, that is integrated in the Agile synchronisation. In order to keep the run-time view consistent with the other views in ARTE, it has to be possible that changes in the high-level view affect the elements of the run-time view.

In this dissertation we elaborate on a specific instance of ARTE, also referred to as “myARTE” that supports the above three criteria as explained in section 1.2.1.

As opposed to the traditional notion of RTE where multiple models are synchronised, we approach RTE in an agile way by synchronising multiple views on one model. Since we consider the transformations for realising consistency between the different views from a higher level, we selected a *high-level* programming language, more precisely Self, that implements a maximum level of abstraction and expressiveness, for constructing the mechanisms that implement the three myARTE criteria. Using the language Self in myARTE allows us to implement the complex requirements for myARTE in an intuitive and appealing way due to the simplicity and the high expressiveness of the language.

Using Self also brings us novel insights into the myARTE requirements. More precisely we are able to map the three criteria of myARTE to more fine-grained language mechanisms in the implementation language of an ARTE approach (see section 1.2.2). We provide a detailed analysis of these required characteristics in different programming languages. This analysis represents an indication of the feasibility of implementing ARTE in these programming languages and can be seen as the basis for guidelines for implementing myARTE in different languages.

As a proof-of-concept of myARTE, we build an ARTE tool in Self, called SelfSync, that is validated with a mature, extended domain model (see section 1.2.3).

Finally, our instance of ARTE is extended with Advanced Method Synchronisation (see section 1.2.4). This has two repercussions. First, SelfSync models can be considered as Executable

Models *in embryo*. Second, extending SelfSync with method synchronisation can allow for a first step in the direction of Visual Aspect-Oriented Programming.

1.2.1 myARTE: An Instance of ARTE

In our approach to ARTE we use multiple views on a system under development that represent one and the same underlying model that is viewed with different levels of abstraction. The highest-level view myARTE considers is the *analysis view* that uses the Extended Entity-Relationship (EER) model [24]. The second view we consider is an implementation view that is realised in the high-level language Self. Each of the views can be manipulated directly. The fact that we use multiple views on one and the same underlying model allows us to meet the requirement for consistency between the different views in a high-level manner, compared to using voluminous model-to-model transformations between different kinds of models. Instead of these complex model transformations we apply light-weight, incremental transformations between different views. In this way the analysis view and the implementation view are kept consistent, without information loss. Moreover, in our ARTE approach we include a rapid prototyping process that can be applied at any time to test the system that was developed so far, even at a very preliminary stage. As a result of the rapid prototyping phase we are able to consider a third, run-time view on systems that is also realised in Self. In our approach to ARTE we realize specific constraint enforcement on run-time objects based on multiplicities and dependencies in the analysis view.

In the following three subsections we describe how myARTE supports these three criteria of ARTE defined above.

Criterium 1: myARTE Includes an Analysis View

In object-oriented software engineering, object-oriented analysis is integrated as a phase in the software development cycle [67]. Numerous object-oriented analysis methods have been developed such as Shlaer-Mellor [97], Jacobson [58], Coad-Yourdon [26], and Rumbaugh [91].

In theory, analysis is claimed to provide statements that describes the problem domain without saying anything about a solution [39]. Such statements should not construct anything artificial about the problem domain and should be independent of any technical platform. Some state that object-oriented analysis is concerned with developing software engineering requirements and specifications that are expressed as a system's object model (which is composed of a population of interacting objects) [37], as opposed to the traditional data or functional views of systems. Others claim that object-oriented analysis is not new and in several cases, such as Rumbaugh's approach [91], coincides with conceptual, static data modelling of systems that has been around since the seventies [52], [53].

This diversity in perspectives is related to the fact that the artifacts of the analysis, that can be referred to as *analysis models*, come in at least two flavours [52]. First they can be seen as artifacts of the requirements analysis process for modelling a part of the world in a platonic sense, i.e. a *domain model* [40] that describes the significant concepts and their relationships in a business. Such models are pure representations of the knowledge about a business, independent to technology platforms. These domain models can exhibit static as well as dynamic information.

A second use of analysis models is to represent a high-level specification of a data base or a physical system. These models are intended to be refined into a more detailed design model. In that case the representation reflects the technology involved and object modeling techniques have additional characteristics to describe physical design issues.

When developing static domain models, it is expected that there is no difference in content between an object model of a business and an Entity-Relationship (ER) model [24] of the same business [52]. The diversity in the targets of the analysis models is often considered more crucial than the notation and the terminology of the data modelling language [52] [39].

In this dissertation we use the term *analysis* for that part of the object-oriented analysis that coincides with static, conceptual data modelling. As a consequence the *analysis models* we consider in this dissertation are *static* domain models that represent the data and the relationships that are significant in the problem domain. An *analysis view* on a system under development presents analysis models of this system. As a consequence, in the remainder of this dissertation we can use the terms "analysis view" and "data modelling view" synonymously.

Modelling Language Our instance of ARTE employs the Extended Entity-Relationship (EER) model [24] to represent elements in the analysis view. EER is sufficiently comprehensible to facilitate the agile principle of close collaboration with end-users. Nevertheless, EER holds the expressive power to be extended with non-trivial modelling concepts, such as Roles.

Although we propose to use EER, and not the de facto modelling language UML, this is not considered as a crucial requirement for our ARTE approach. In fact, for static data modelling, the UML is considered as functionally equivalent to the ER model [51]. However, the UML is sometimes seen as more appropriate for designers, while the EER model is considered as more suitable when end-users are involved, such as during (requirements) analysis [51].

Role Modelling While performing analysis, a difficulty encountered frequently, is modelling roles [80]. The data modelling view of our instance of ARTE explicitly includes a Role Modelling concept [114]. It is possible to model that one entity can perform one or more roles. Moreover, the data modelling language in our ARTE instance enables us to express that some roles are mutually exclusive.

In myARTE, this Role Modelling concept is synchronised into a suitable implementation. One of the perspectives on roles involves a paradox between conceptual roles that can be considered as dynamic subtypes and implemented roles that can be seen as implementation supertypes of the objects that perform them [100], [114]. This paradox can result in a complex mapping between conceptual roles in a data model and a corresponding class-based implementation elements [38]. Due to this problematic mapping, including roles in ARTE is an all but trivial experiment.

Criterium 2: myARTE is Agile

In myARTE the synchronisation between the data modelling view and the implementation view happens in a light-weight and incremental way. We aim at providing ARTE tool support for an Agile Development cycle that consists of two phases, as illustrated in figure 1.1. In the first phase, an

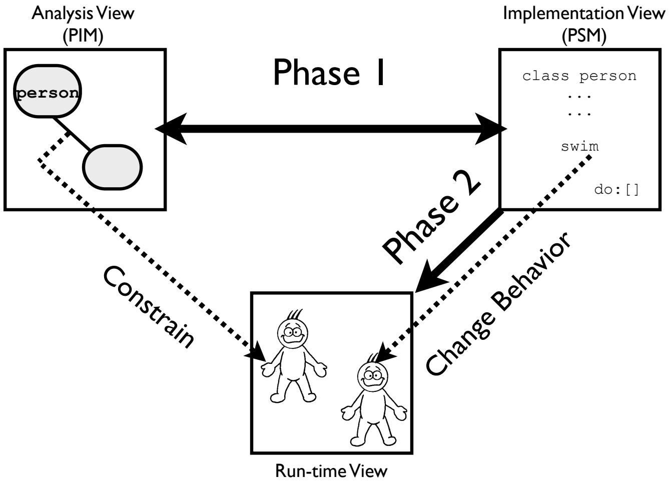


Figure 1.1: A New Agile MDD Practice for ARTE.

EER data modelling view and an object-oriented implementation view are continuously synchronised. Model storming in the analysis view automatically results in a corresponding object-oriented implementation view.

A certain set of changes to this resulting implementation, are immediately and automatically reflected in the data modelling view. Other changes that do not have a conceptual equivalent in the data modelling view, are not synchronised. Change cycles that consist out of changes in the data modelling view and in the implementation view, and automatic synchronisations, can be combined in an arbitrary way, until the system developed so far is run. For this purpose, the second phase, Interactive Prototyping, is an instantiating process that rapidly prototypes the current analysis view. In this way a third, run-time view on the application under development is created.

The first phase can be fully automated, while the second one is semi-automated, since certain information such as actual references between run-time objects, are determined interactively. The required tool support for this two-phased Agile Development cycle is reported on in [118].

The difficulty in building ARTE tools that support the above development process is twofold. First, since the change cycles can be combined in an arbitrary way, and the synchronisations have to preserve anything that was added manually in the code, we cannot simply transform an entire model into new code or vice versa. Instead, we need Optimal Transformations that are incremental, i.e. to the level of attributes and methods, and that preserve the already existing “contents” of these attributes and methods in the implementation. Second, in order to enable Interactive Prototyping, certain relationships and constraints defined in the data model, need to be taken into account in the implementation at the time that new run-time objects are actually created.

Criterium 3: myARTE is Dynamic

The run-time view is also part of the myARTE synchronisation process. A Run-time Generation is a group of similar run-time objects that are created from the same implementation object. In class-based object-oriented programming languages for example, such a generation includes instances created from the same class, while in prototype-based languages, a generation gathers run-time objects that are created from the same prototype.

In the first place, the behaviour of a Run-time Generation can automatically be changed, in the case of certain changes to the analysis view or the implementation. Second, these run-time objects are continuously and automatically constrained by certain relationships that are defined in the analysis view.

myARTE supports run-time evolution in an automated way. In [50], three kinds of unanticipated run-time software evolution are classified for which a solution in Java is provided in [49].

Code change describes how the program code is changed, *state change* describes how the state of the running system must be transferred and *timing restrictions* describes when it is appropriate to introduce the change.

First, myARTE focuses on code changes that are possibly initiated by a change in the analysis view. Examples include adding an entity, removing or renaming an attribute and changing a parent entity. Some of these actions have repercussions on a Run-time Generation. myARTE automates the propagation of code change that is performed at run-time, to the implied run-time objects. In order to do so, we require sharing between a group of similar run-time objects. The shared behaviour has to survive multiple synchronisation steps in myARTE. Moreover, we have to be able to extend the shared behaviour at run-time. In many object-oriented programming languages, these are non-trivial requirements.

Second, myARTE supports Constraint Enforcement on Run-time Generations. More specifically, myARTE synchronises the multiplicities of relationships and other constraints that are present in the data modelling view in the implementation, and enforces these constraints in the involved run-time objects. Consider for example two entities `a` and `b`, that are in a one-to-one relationship in the data modelling view. It is automatically enforced that each corresponding run-time object can have at most one reference to the other one. Such constraint enforcement requires, also in this case, that certain relationships and constraints defined in the data model, are visible in the implementation at the time that run-time objects are manipulated. Moreover, we need to be able to dynamically inspect the internal state of these run-time objects.

1.2.2 Choosing an Implementation Language for myARTE

We deliberately selected a non-conventional programming language such as Self, in order to explore its advantages for implementing the sophisticated, unanticipated requirements of myARTE, compared to more popular languages such as Java and C++ that are often used for implementing the state-of-the-art in RTE tools. Applying a high-level language such as Self in the implementation and run-time view of myARTE has a first, direct, quantitative advantage. The high expressiveness of the language allows us to implement the myARTE requirements in a significantly more compact way compared to less expressive languages such as Java and C++. Additionally, using Self brings a less obvious, qualitative advantage: it brings the novel insight of seeing the basic language constructs in Self as “building blocks” of twelve language mechanisms that realise the aforementioned three myARTE requirements. We analyse how these language mechanisms can be constructed in

C++, Java, Smalltalk, Ruby and Self. The presence of the required language characteristics can be considered as proportional to the extent in which programming languages support dynamic typing, meta-programming and prototypes. More precisely, dynamically-typed prototype-based programming languages with a powerful meta-programming mechanism are the most suitable for implementing myARTE, if they combine the high flexibility of “everything is an object” with the efficiency of an explicit mechanism for sharing behaviour. The conclusion of this analysis confirms the intuitivity and comfort when using the language Self for implementing a myARTE tool.

1.2.3 The SelfSync myARTE Tool

SelfSync is a proof-of-concept tool that supports myARTE between a data modelling view in EER and an implementation view in Self. SelfSync is built on top of Self and integrates a graphical drawing editor for EER diagrams. It supports the Agile Development cycle that combines Active Modelling with Interactive Prototyping. To let SelfSync support Role Modelling, we create an implementation pattern for *Warped Hierarchies*, in which the inheritance hierarchy for state and the inheritance hierarchy for behaviour exhibit the opposite order. In the implementation, the state of a role is dynamically added to the state of the object that starts performing the role. Our approach to Role Modelling in Self is reported on in [114] and [116].

Moreover, SelfSync implements Optimal Transformations, Run-time Generations and Constraint Enforcement. The SelfSync tool was demonstrated at the OOPSLA 2005 conference [113]. How SelfSync is built and applies the Agile Development cycle is explained in [115] and [117].

The SelfSync tool is validated with the TELEBIB domain model. TELEBIB is the standard for the Belgian insurance business. In its current state, the model covers the major business needs in the areas of policy administration, claims handling and accounting, and thus can be considered as a mature and realistic example of a domain model. This EER diagram that contains 343 entities, 1048 attributes, and 233 relationships and generalisations, was modelled in SelfSync, resulting automatically in a corresponding implementation.

In the TELEBIB model 23 roles are identified, that can be restructured due to the expressive Role Modelling construct in SelfSync. We discuss when and where different kinds of application functionality can be added in the synchronised implementation view. Interference problems between generated implementation elements, are mainly caused by multiple inheritance. Moreover, TELEBIB in SelfSync is considered as a highly generic domain model that is applicable in both small insurance applications that require only a reduced part of the TELEBIB domain model, and in large applications that employ the entire domain. Finally, TELEBIB in SelfSync benefits from the Constraint Enforcement at run-time for multiplicities, dependencies and role combinations.

1.2.4 Advanced Method Synchronisation

Due to the incremental character of the transformations in SelfSync and Self's uniformity between state and behaviour, myARTE is extensible with *Advanced Method Synchronisation* (AMS) in a straightforward way. AMS automatically synchronises between operations in the data modelling view and methods in the implementation view. In order to do so, we add operations to the EER model in the analysis view. We can edit the body of these operation in the analysis view. The provided Self code is automatically saved as the method body of the corresponding method in the implementation. Alternatively, consulting an operation after a change to the corresponding method, will result in the new body.

AMS has two important repercussions. First, it increases the feasibility to extend SelfSync with behaviour in the data model, that can be "run". More precisely, operations in the EER model can be evaluated in the context of the EER model itself using an Entity Checker. This approach allows us to consider SelfSync EER models as Executable Models in embryo.

Second, AMS makes it possible to "inject" pieces of code before or after existing operations in the data modelling view. We foresee that these `before/after` constructs can be consolidated in SelfSync, in such a way that this extension can be considered as a first step to a simplified version of Visual Aspect-Oriented Programming. Multiple operations in different entities in the EER diagram function as static join points that are selected graphically in a diagram instead of being described by a pointcut.

1.3 Dissertation Outline

As our approach to ARTE considers a data modelling view and an implementation view, chapter 2 focusses, among others, on the suitability of data modelling languages for the data modelling view, while chapter 3 is concerned with an analysis of object-oriented implementation languages to implement myARTE. In these chapters, we motivate why myARTE employs EER in the data modelling view, and the prototype-based language Self in the implementation view.

Next, chapter 4 introduces an ARTE approach for the selected EER modelling and the Self programming language. We deliberately decided to present a concrete approach based on EER and Self, as opposed to a more generic but abstract description. Nevertheless, the analysis of the data modelling languages in chapter 2, and the analysis of language characteristics of programming languages in chapter 3, make it possible to specialise the myARTE approach for other modelling and programming languages.

The SelfSync tool that implements our instance of ARTE, is introduced in chapter 5. The extension with operations to the data modelling view in SelfSync is discussed in chapter 6. A validation for the myARTE approach and the SelfSync tool can be found chapter 7.

Related work for our research is included in chapter 8 while a conclusion can be found in chapter 9.

A more detailed summary of this dissertation's chapters can be found below:

- **Chapter 2: Advanced Round-Trip Engineering**. This chapter provides a general insight in Round-Trip Engineering (RTE). We elaborate on the motivation for an abstract data modelling view, a code-time implementation view and a run-time view. For the abstract data modelling view, the Unified Modelling Language and the Extended Entity-Relationship Model are introduced and compared. Six evolution scenarios that represent evolution of one of the three views and the subsequent synchronisation in the other views are introduced and exemplified. Next, the functionalities of myARTE tools are elaborated.

- **Chapter 3: Language Requirements for myARTE Tools**. In this chapter we present language requirements (i.e. the constructs and the characteristics) that are crucial when implementing our instance of ARTE. Next, we identify the three programming language characteristics - dynamic typing, reflection and prototypes - as indicators for the presence of the above language requirements. We continue with an in-depth discussion on how myARTE could be implemented in C++, Java, Smalltalk, Ruby and Self.

- **Chapter 4: A New Practice For Agile Model-Driven Development**. In this chapter we introduce a practice for ARTE, that is compatible with Agile MDD. We define an extension of the Extended Entity-Relationship (EER) notation for the data modelling view. The prototype-based programming language Self is used as a platform for the implementation and run-time views. A set of default mappings or correspondences are defined between the EER view and the Self views. We provide a number of myARTE scenarios for illustrating this new practice.

- **Chapter 5: SelfSync: An ARTE Tool**. As a proof-of-concept, we provide an ARTE implementation in Self, by developing an ARTE tool called SelfSync. SelfSync is written entirely in Self.

It includes a graphical drawing editor for EER diagrams, that is integrated in the Self environment, and that was developed with the help of the Morphic framework. SelfSync implements an active bidirectional link between entities in the EER view and objects in the code-time implementation view, based on a Model-View-Controller architecture. We illustrate the use of SelfSync with concrete usage scenarios.

- **Chapter 6: Advanced Method Synchronisation: Towards Executable Models and Visual AOP**. In this chapter we describe how myARTE is extended with Advanced Method Synhronisation (AMS): operations in the data modelling view are syncronised with methods in the implementation view.

  We illustrate how AMS allows to "evaluate" operations in the context of the data model, and explain how myARTE models one day might fit in the domain of Executable Models.

  Next, we describe how AMS makes it possible to implement before/after constructs in the data modelling view. We elaborate on how these *Code Injections* can be considered as a step in the direction of Visual AOP.

- **Chapter 7: Validation**. In this chapter we provide a validation of our ARTE approach and the associated SelfSync tool with the help of a case study. TELEBIB defines the standard for Belgian Insurances and includes a mature and extended domain model. We model this domain model entirely in SelfSync and highlight the advantages of our approach.

- **Chapter 8: Related Work**. In this chapter we provide an overview of related research in the domain of data modelling and ARTE, and we discuss an example of an Executable Model.

- **Chapter 9: Conclusion**. The conclusion of this dissertation contains a list of our contributions and the future research directions.

Chapter 2

Advanced Round-Trip Engineering

In this chapter we introduce Agile Model-Driven Development (MDD) that combines multiple views on a system under development with a highly iterative and incremental change-oriented development cycle (see section 2.1). Round-Trip Engineering (RTE) plays an important role in Agile MDD where the multiple, rapidly changing views on a system are to be kept consistent, as introduced in section 2.2. The Evolution Scenarios in section 2.3 result from the systematic construction of possible usage scenarios for RTE tools and illustrate which changes are relevant and how they are synchronised in the different views. We believe that traditional Round-Trip Engineering tools are incapable of supporting Agile MDD. Therefore, we introduce Advanced RTE (ARTE) in section 2.4 as a substantial evolution to RTE, that fits Agile MDD and can be supported by the next generation of RTE tools. We elaborate on myARTE, that is one instance of ARTE in section 2.5. Since myARTE considers a static data modelling view we study two possible data modelling languages. Section 2.6 describes the Unified Modelling language (UML) while the Extended Entity-Relationship (EER) model is introduced in section 2.7. Why myARTE benefits from the EER model is described in section 2.8. Finally a conclusion is provided in section 2.9.

2.1 Agile Model-Driven Development

There are several reasons why requirements change frequently during software development such as a demand for new functionality, misunderstandings by developers or end-users, or a lack of understanding of the problem domain. In the late 1990's several software development methods tried to anticipate this so-called *requirements churn* by employing a more lightweight approach to software engineering. This was achieved by prioritizing issues such as flexibililty, end-user interaction, productivity, and individuality. In 2001 the commonalities of these methods' solutions were gathered in Agile Software Development [27] denoting the ability to "change quickly and easily". This approach to software engineering focuses on various short iterations of development in order to respond quickly and easily to changing software requirements. The common development values of Agile Software Development are stated in a manifesto [123] in terms of priorities:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Although Agile Software Development is new and has its roots in industry it rises in academic research. Currently the validity of agile methods as the new "best practice" is being researched. There already exist a number of frequently applied agile methods.

2.1.1 Agile Software Development Practices

During *Rapid Prototyping* usable prototypes are produced early in the development cycle to permit end-user feedback and analysis in each iteration [11]. Rapid Prototyping is also one of the core elements of *rapid application development* (RAD) [7]. This software development method extends the iterative development and Rapid Prototyping of the spiral model with the use of Computer-Aided Software Engineering (CASE) tools. RAD significantly increases the speed of development due to the use of CASE tools and increases the degree to which software meets the requirements.

Extreme Programming (XP) is the best-known agile method in both industry and the academic world [8]. Originally formulated by Kent Beck and influenced by Smalltalk, XP contains practices such as the use of *releases*, i.e. intervals of approximately six months. A release consists of iterations of two weeks each during which developers implement user stories. User stories resemble use-cases or usage scenarios and replace requirement documents. They are written compactly in collaboration with the end-user in natural language. XP encourages to keep the design as *simple* as possible. All code to be included in a release is *pair programmed*: i.e. created by two people working together at a single computer. *Test-Driven Development* (TDD) has to increase reliability. TDD is an evolutionary approach to development which combines test-first development where a test is written before write just enough production code is implemented to fulfill the test, with refactoring. For more details on XP we refer to [8].

Adaptive Software Development is an agile method [57] that grew out of Rapid Application Development. In ASD, the static *plan-design-build* life cycle is replaced by a dynamic *speculate-collaborate-learn* life cycle where continuous change is the norm. This change-oriented life cycle is dedicated to continuous learning, re-evaluation, peering into an uncertain future, and intense collaboration among developers, management, and end-uses. For more details we refer to [57]. Such an ASD life cycle has six basic characteristics:

- mission-focused: the emphasise is on results not tasks
- feature-based: based on the end-user functionality to be implemented in one iteration
- iterative
- time-boxed [1]
- risk-driven
- change-tolerant

For a more complete survey of agile methods we refer to [2].

2.1.2 Agile Model-Driven Development

Agile Model-Driven Development (MDD) uses agile practices for Model-Driven Development (MDD). MDD assumes that all crucial knowledge about a system can be present in an abstract model from which the real software is derived automatically with sophisticated model transformations. Such a *model* is defined by Mellor et al. [76] as

> *A coherent set of formal elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis, such as*
>
> - *Communication of ideas between people and machines*
> - *Completeness checking*
> - *Race condition analysis*
> - *Test case generation*
> - *Viability in terms of indicators such as cost and estimation*
> - *Standards*

[1] In time-boxed development projects are divided into smaller parts with their own fixed deadlines and deliverables.

- *Transformation into an implementation*

A vast research area in MDD deals with model transformations. For a taxonomy of the current model transformation approaches and tools we refer to [29]. Model transformations can be *model-to-code* transformations where for example a model is transformed into text that is compiled afterwards. Such an approach can be found in the object-oriented framework Jamda [134] that applies a visitor mechanism for generating code. Various tools such as ArcStyler [124], OptimalJ [143] and XDE [144] support model-to-code transformations. MDD approaches, such as ATOM [126] and GreAT [1] realise *model-to-mode*l transformations that translate beteen source and target models that can be instances of different or the same meta-model [29]. Some of these approaches generate intermediate models in order to bridge large abstraction gaps between different kinds of models. Tool support is provided by both open-source and commercial tools, such as OptimalJ [143] and XDE [144].

Other MDD research deals with the representation of the different models. Many approaches follow the *Model-Driven architecture* (MDA) [64, 141] standard defined by the Object Management Group (OMG). The basic idea of the MDA is that a system's functionality is defined as a platform-independent model (PIM) in an appropriate specification (modelling) language and is then translated to one or more platform-specific models (PSMs) for the actual implementation. Although MDE or MDD do not require the MDA or the UML, often the MDA standards are applied and the Unified Modelling Language (UML) [41] has become the de facto specification language for PIMs. The MDA model architecture includes multiple standards such as the Unified Modelling Language (UML) [41] (see also section 2.6) and the XML Metadata interchange (XMI) [152]. For each standard the MDA includes guidelines for expressing the specifications of a system into PIMs or PSMs. The UML is discussed into more detail in section 2.6. For more details on the other standards we refer to [141]. Most existing practices for MDD have a generative nature and employ sophisticated, voluminous model transformations to to semi-automatically transform PIMs into PSMs.

Agile MDD does not specify the kind of model to use and code is written progressively in step with the models [4, 5]. Agile MDD promotes an evolutionary approach, in which implementation occurs iteratively and incrementally. Development starts with a small part of modelling followed by a significant amount of coding, iterating back when necessary. The *Agile Models* that are initially created are just "barely good enough", i.e. sufficient for the task at hand. There are no analysis or design models as different Agile Models merely capture independent aspects of the system being developed. Design efforts are divided between modelling and coding, with the majority of the design being performed as part of the implementation efforts.

An iterative Agile MDD life cycle [4] consists of six phases as illustrated in figure 2.1: the *initial modelling* activity includes two main sub-activities, *initial requirements modelling* and *initial architecture modelling*. These activities are performed in the first iteration. The other activities *model storming*, *reviews*, and *implementation* occur during any iteration, including the first. Often sophisticated modelling tool such as Borland's Together [150] are employed for "visual programming" during the implementation phase.

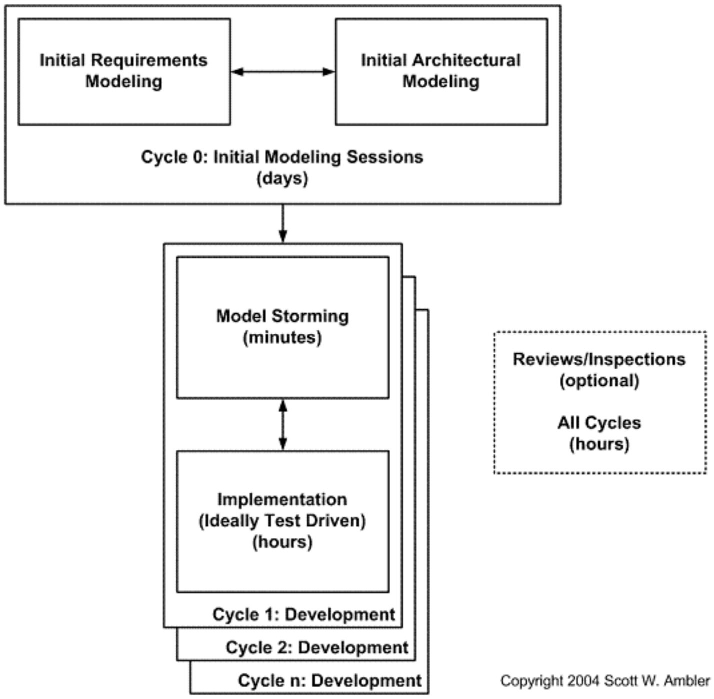

Initial Requirements Modeling
Initial Architectural Modeling
Cycle 0: Initial Modeling Sessions (days)
Model Storming (minutes)
Implementation (Ideally Test Driven) (hours)
Cycle 1: Development
Cycle 2: Development
Cycle n: Development
Reviews/Inspections (optional)
All Cycles (hours)
Copyright 2004 Scott W. Ambler

Figure 2.1: Life cycle phases in Agile MDD [5].

2.1.3 Agile Model-Driven Development Practices

There exists already a number of practices for Agile MDD. The notion of Executable Models allows the OMG to state that their MDA standard is not restricted to Generative MDD but is also suitable for Agile MDD. Their *Agile MDA* [74] seconds the idea that code and Executable Models are operationally the same. In Agile MDA each executable model is a PIM. As in the MDA UML is the de facto modelling language for PIMs, Executable Models in this case totally depend on *Executable*

UML [75]. The latter is a profile of UML that defines execution semantics for a selective subset of UML with the help of the Object Constraint Language (OCL) [142]. The subset is computationally complete and modelling rules are enforced to allow execution. The executable UML models, such as class diagrams and state diagrams are merged and translated by a model compiler [74] into code that is considered as the PSM.

There is no need to manipulate the PSM or to visualize it as a model: the code is a weaving together of the elements of the PIM and of the required platforms, and it is executable. There exist a number of academic applications of Executable UML, such as Extreme Models (XM) [14]. One of the few industrial examples can be found in the Nucleus UML suite [140] that includes a code generator for embedded systems, that supports Executable and Translatable UML (xtUML).

In [5] three practices to Agile MDD are introduced from an Agile Modelling perspective. We summarize them by describing the fundamental steps of one iteration. The most frequently used is *Agile Manual Modelling* (AMM) where inclusive models (i.e. that cover the entire functionality) are created using simple tools (paper, white-board, drawing editors) and simple techniques (use cases, CRC cards, etc.). “The code is the design” philosophy is followed: agile modelers convert the model manually to the code. Agile developers then alter the code and deploy the working software. This approach is followed by the Extreme Programming (XP) community. Figure 2.2 illustrates this approach that currently is applied during the majority of software development cases [5].

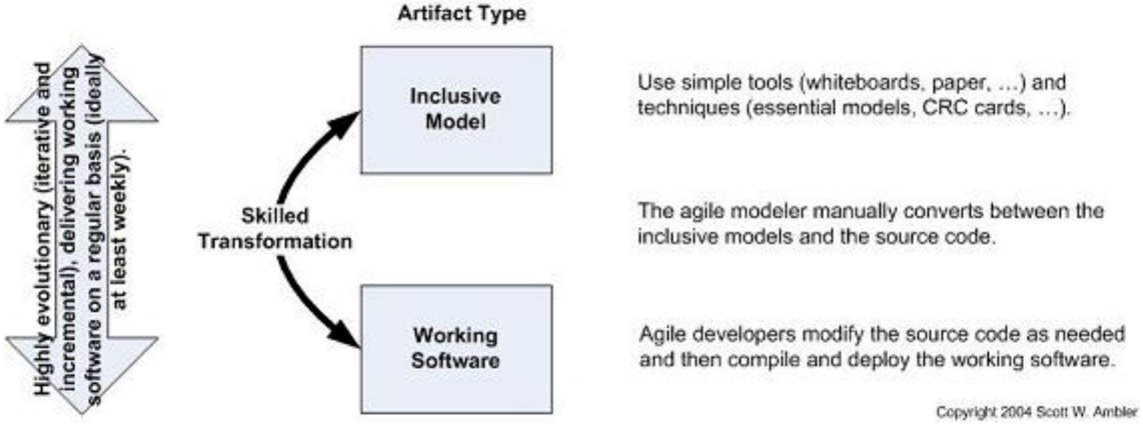
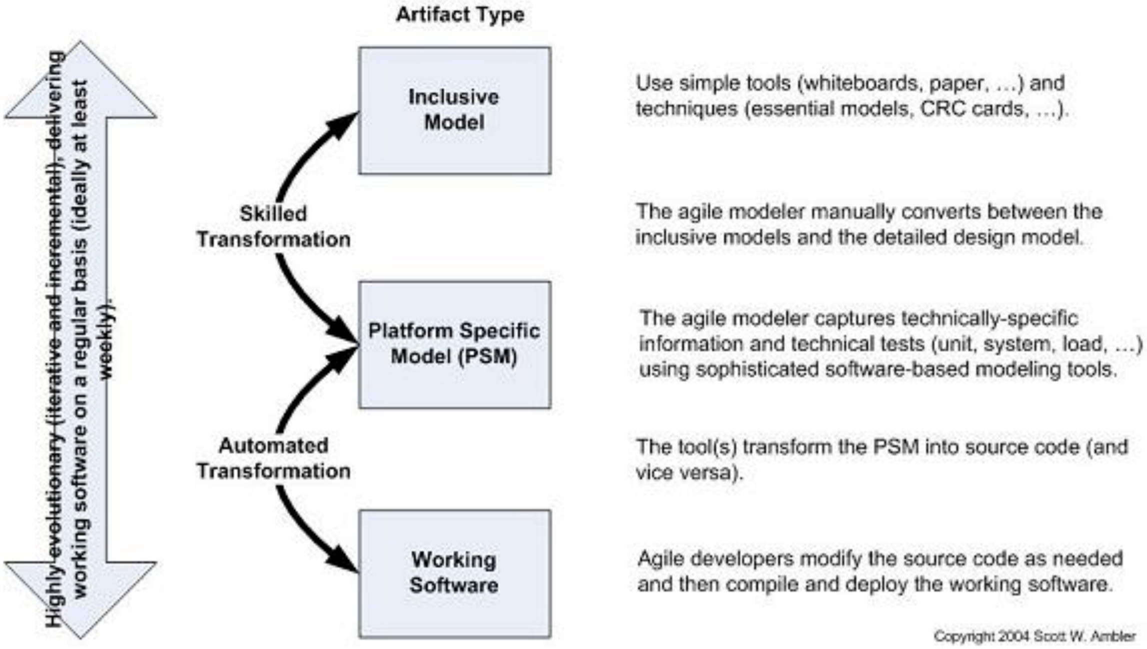

Artifact Type
Highly evolutionary (iterative and incremental), delivering working software on a regular basis (ideally at least weekly).
Inclusive Model
Skilled Transformation
Working Software
Use simple tools (whiteboards, paper, ...) and techniques (essential models, CRC cards, ...).
The agile modeler manually converts between the inclusive models and the source code.
Agile developers modify the source code as needed and then compile and deploy the working software.
Copyright 2004 Scott W. Ambler

Figure 2.2: Agile Manual Modelling [5].

A second approach is *Agile Computer-Aided Software Engineering* (ACASE) (see figure 2.3) where inclusive models are used to analyze and explore requirements with end-users. Agile modelers convert the inclusive model manually into a detailed design model and capture technically-specific information and tests using sophisticated modelling tools. These tools transform the PSM into code (and vice versa).

These modelling tools are expected to support the automatic synchronisation between the models and the code during Round-Trip Engineering (see section 2.2). Agile developers should be able to model or write code, and have the corresponding code/models updated automatically. This approach can be taken by any type of agile team, including XP teams.

It is estimated that less than a quarter of development teams applies this practice [5].


Artifact Type
Highly evolutionary (iterative and incremental), delivering working software on a regular basis (ideally at least weekly).
Inclusive Model
Skilled Transformation
Platform Specific Model (PSM)
Automated Transformation
Working Software

Use simple tools (whiteboards, paper, ...) and techniques (essential models, CRC cards, ...).

The agile modeler manually converts between the inclusive models and the detailed design model.

The agile modeler captures technically-specific information and technical tests (unit, system, load, ...) using sophisticated software-based modeling tools.

The tool(s) transform the PSM into source code (and vice versa).

Agile developers modify the source code as needed and then compile and deploy the working software.

Copyright 2004 Scott W. Ambler

Figure 2.3: Agile Computer-Aided Software Engineering (ACASE) [5].

Thirdly, figure 2.4 illustrates what is referred to as a “realistic” approach to the aforementioned Agile Model-driven Architecture (AMDA) [74]. This approach differs from ACASE in the fact that agile modelers convert the inclusive model manually into a detailed PIM and capture domain information and acceptance tests using sophisticated modelling tools. These tools then transform the PIM into one or more PSMs (and vice versa). In [5] it is estimated that this advanced practice for MDD is applicable for 5% of all development teams, most of them being embedded or real-time system developers.

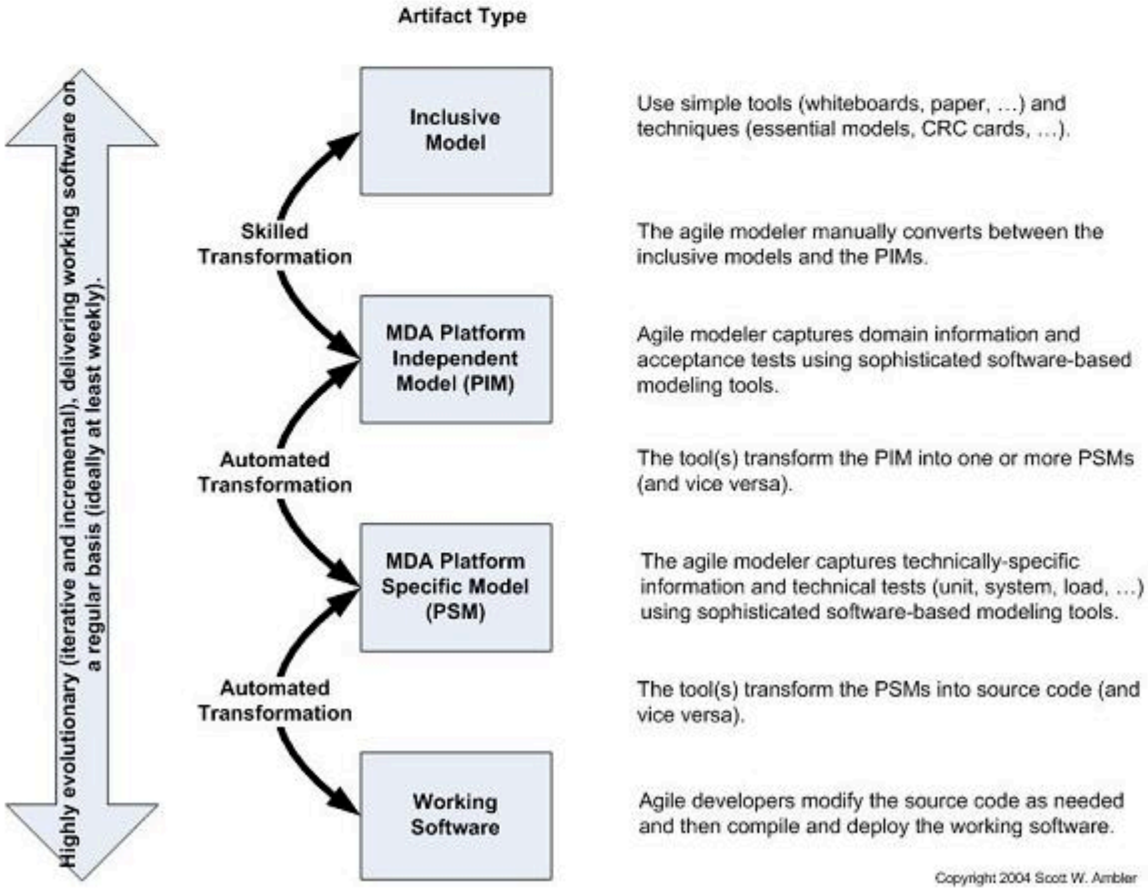

Artifact Type
Highly evolutionary (iterative and incremental), delivering working software on a regular basis (ideally at least weekly).
Inclusive Model
Skilled Transformation
MDA Platform Independent Model (PIM)
Automated Transformation
MDA Platform Specific Model (PSM)
Automated Transformation
Working Software
Use simple tools (whiteboards, paper, ...) and techniques (essential models, CRC cards, ...).
The agile modeler manually converts between the inclusive models and the PIMs.
Agile modeler captures domain information and acceptance tests using sophisticated software-based modeling tools.
The tool(s) transform the PIM into one or more PSMs (and vice versa).
The agile modeler captures technically-specific information and technical tests (unit, system, load, ...) using sophisticated software-based modeling tools.
The tool(s) transform the PSMs into source code (and vice versa).
Agile developers modify the source code as needed and then compile and deploy the working software.
Copyright 2004 Scott W. Ambler

Figure 2.4: Agile Model-Driven Architecture (AMDA) [5].

The sophisticated modelling tools that are responsible for the automated transformations are often Round-Trip Engineering tools (see section 2.2).

2.1.4 Summary of The Discussed Approaches

In table 2.1, the discussed approaches of this section are summarised.

2.2 Traditional Round-Trip Engineering

Using multiple changing views on a system under development, as MDD advocates, requires at least that interrelated views are kept consistent [95]. The ability to maintain the consistency of multiple, changing software artifacts in software development environments and tools is a key factor in what is referred to as *Round-trip Engineering* (RTE). RTE involves forward and reverse engineering steps such that software artifacts, such as, code and graphical models, become synchronized at

Agile Software Development
Rapid Application Development (RAD)
Extreme Programming (XP)
Test-Driven Development (TDD)
Adaptive Software Development (ASD)
Agile Model-Driven Development
Agile MDA
Executable (and Translatable) UML
Agile Manual Modelling (AMM)
Agile CASE (ACASE)
Agil MDA (AMDA)

Table 2.1: Summary of the discussed approaches in section 2.1.

certain points in time [95]. Where forward engineering deals with generating one or more software artifacts from an input artifact, where the generated artifact is closer to the final deployable system, reverse engineering involves generating one or more software artifacts that abstract certain details of the input artifacts, thereby trying to recover any information lost in the forward step [95]. Both forward and reverse engineering steps involve transforming one or more artifacts into one or more other artifacts. This is often realised by a *"single, isolated transformation, where any information in the target artifact is not considered, and typically a new artifact is created, possibly replacing the previous version, if one exists"* [95].

Sometimes RTE is defined as an application of mathematical domain transformation. In that case RTE systems are defined as systems where the result of forward engineering a design that is reverse engineered from a product, is identical to the original product [55]:

> *Let D be the design of the product P, let r be a reverse-engineering procedure such that r(P) = D*
> *and let g be a product generation procedure such that g(D) = P.*
> *Iff g(r(P))* $\equiv$ *P (and thus r(g(D))* $\equiv$ *D) then g,r is a round-trip engineering system.*

The state-of-the-art in RTE research includes Automatic Roundtrip Engineering (ARE) [6] where the inverse of the aforementioned domain transformations are derived automatically. A special case of ARE is Automatic Model View Controller Engineering (MVARE) [68] where the well-known Model-View-Controller (MVC) [47] design pattern is applied to software construction. The "model" is the system under development while the views are the different viewpoints on the system. The controller is responsible for maintaining relations between views and model, and for ensuring that every change in every view is propagated to the other views and to the model.

In this dissertation we focus on a specific case of RTE where:

- The system under development is represented by multiple views that view one underlying model.

- One of the views is a *graphical high-level view* containing static, structural data diagrams such as UML [41] class diagrams and Entity-Relationship [24] diagrams. We also refer to this view as the *data modelling view*. This view consists of all the information pertaining to *nodes* (entities, classes, attributes and operations) as well as *links* (inheritance, associations with multiplicities and aggregations).

- One of the views is an *implementation view*: a *code-time* implementation view containing textual and/or graphical views on the implementation objects, which show everything related to object-oriented programs. Developers can alter these implementation objects.

Agile MDD advocates the use of a third, run-time view in three ways.

- Rapidly changing requirements demand automated Unit Testing. The Extreme Programming Unit Testing approach has been automatized in the xUnit framework [9] that resulted in multiple standards suchs jUnit for Java, sUnit for Smalltalk and rUnit for Ruby. Borlands ALM Solution includes the jUnit standard for automated Unit Testing.

- Moreover quickly changing requirements sometimes result in the necessity for a system to be adapted at run-time. Current RTE tools do not provide support to influence them steered from a high-level model.

- Dynamic Aspect-Oriented Programming (AOP) [35] stresses the importance of dynamicity and is characterised by:

  - *Per Instance Interception*: The ability to do more fine grained bindings at the instance level. So that two instances of the same class can have two separate interceptor/advice chains.

  - *Hot Deployment*: The ability to add and remove advice bindings at run-time to any prepared class.

  Dynamic AOP is supported by the state-of-the-art, such as Spring [147] and JBoss [136]. The latter resolves pointcut and advice bindings at run-time, but never has been integrated in existing RTE environments.

Therefore we introduce a third run-time view to RTE. The *population view* is a *run-time* implementation view containing textual and/or graphical views on run-time objects (instances), which contain actual data for running the application. We also refer to these objects as *population* objects, as they are used to "populate" an application. Population objects are derived from implementation objects.

Few RTE tools, such as the highly advanced Together [150] (see section 8.2.1), include the third population view, but none of them integrates the population objects in the RTE process. Once the classes are instantiated into run-time objects, the synchronization with the data modelling view is no longer supported.

2.3 Evolution Scenarios

In this section we illustrate six examples of Round-Trip Engineering for Agile MDD, with the help of *Evolution scenarios*. These scenarios result from the systematic construction of possible usage scenarios for RTE tools, considering the three views and the two types of elements in each of these views. Nodes are the basic elements in the different views while the links are the connections between them.

We define six scenarios that RTE tools are to support during Agile MDD. Each of them corresponds to a particular direction of RTE and to particular elements that are changed and subsequently synchronized as summarized in Table 2.2: Forward engineering is an activity embodied by scenario 1 and 2. In this case the changes applied to the data modelling view are synchronised in the implementation view, and if appropriate in the run-time population view.

	Nodes	**Links**
Forward Engineering		
Data modelling view	Scenario 1	Scenario 2
Backward Engineering		
Code-time implementation view	Scenario 3	Scenario 4
Run-time Engineering		
Run-time population view	Scenario 5	Scenario 6

Table 2.2: The six evolution scenarios.

Backward engineering involves scenario 3 and 4. The changes applied to the implementation view are synchronised in the data modelling view, and if appropriate in the run-time population view.

Scenario 5 and 6 are typical for run-time engineering. We state that, by default, the changes to run-time objects are not propagated onto the other views. E.g. if in one instance of a class a reference is added to another instance, this should not become visible in the implementation view or the data modelling view.

Such change propagation can however be appropriate when, for example, a large number of population objects deviates in the same way from the implementation object they were created from. In that case the differences can be synchronised in the implementation view and thus the data modelling view.

The scenarios can be summarised as follows:

Scenario 1: *Changes to nodes in the data modelling view, which are synchronized in both the code-time implementation and the run-time population view.* Typical evolution steps are creating and removing nodes, and adding a placeholder for state or behaviour to these nodes. For example adding an attribute to a class in a UML class diagram.

Scenario 2: *Changes to links in the data modelling view, which are synchronized in both the code-time implementation and the run-time population view.* Typical evolution steps in this scenario are adding and removing an association, or establishing specialisation between two nodes. For example adding a one-to-many association between two entities in an Extended Entity-Relationship diagram (see section 2.7). Another example is to change the multiplicity of an association between two classes in a UML class diagram.

Scenario 3: *Changes to nodes in the implementation view, which are synchronized in the data modelling view and the population view.* Typical evolution steps are creating new programming objects and adding attributes or methods to them. Examples include writing code for a new class in Java and adding an instance variable to it, which is synchronised in the data modelling view by graphically adding a new attribute to the corresponding class.

Scenario 4: *Changes to links in the implementation view, which are synchronized in the data modelling view and the population view.* Typical evolution steps are to add or remove a pointer reference, and to establish a new inheritance relationship between nodes. Examples include subclassing an existing C++ class, or changing the initial value of an instance variable in a Java class definition to contain another object.

Scenario 5: *Changes to nodes in the population view.* By default these changes are not propagated onto other views. Such change propagation can be appropriate when a large number of population objects deviates in the same way from the implementation object they were created from.

In that case the differences can be synchronised in the implementation view and thus the data modelling view. Typical evolution steps are creating or deleting new run-time objects and, if possible, extend them with new state or behaviour. Examples include instantiating a Smalltalk class.

Scenario 6: *Changes to links in the population view*. We stated that, by default, these changes are not propagated onto other views, as explained in the previous scenario. Typical evolution steps are to add or remove pointer reference in a run-time object or to establish a new inheritance relationship for a run-time object. Examples include changing the value of an instance variable in a Java instance.

2.4 Advanced Round-Trip Engineering

We argue that the low applicability of Agile MDD practices that involve an automatic transformation, as opposed to the popularity of Agile Manual Modelling, is a consequence of the fact that current traditional RTE tool support is not suited for the needs of Agile MDD.

2.4.1 Shortcomings of Traditional RTE Tools

We argue that the following shortcomings make traditional RTE tools unfit for Agile MDD:

- **High-level Models are Not Synchronised**: The views that are considered in the state-of-the-art in RTE tools and that are kept consistent with the aforementioned model-to-model transformation often exhibit – even in a very preliminary stage – a certain amount of technical detail. This makes it hard for end-users to totally comprehend the models that are created in these views and decreases end-user interaction during the Agile development process. This also implies that modelling constructs such as Roles [114] that are typically encountered early in the software development cycle, such as during the requirement analysis phase, are not available as first class modelling concepts in MDD.

- **No Agile Life Cycle Support**: During Agile MDD it is quite common to model storm[2] for several minutes, during which (a part of the) new requirements are integrated. Model storming is then followed by coding, based on practices such as testing or refactoring, for several hours or days. The state-of-the-art in RTE tools are intended for generative MDD that creates sophisticated models that can automatically be transformed with sophisticated model-to-model

[2] The activity of some people together creating models with simple tools such as a whiteboard.

transformations into different models in order to reflect the realities of various deployment platforms [111]. As Agile MDD focuses on the thought process that rises prior to the actual implementation [111] these tools are often too complex for the task at hand, such as short model storming or a small amount of coding that are typical for one iteration in the agile development cycle.

Moreover, an agile development cycle requires an advanced synchronisation strategy that anticipates changes to a high-level model for which there already exist pieces of a corresponding implementation, that should survive subsequent model transformations. Most RTE tools do not provide such synchronisation support, as traditionally code is only derived after extensive modelling. Additionally, few of these RTE tools support rapid prototyping and test simulation.

- **The Run-time View is Neglected**: For testing purposes during the implementation phase, Agile MDD advocates rapid prototyping any model that was developed so far, even at a very preliminary stage. Current RTE tools however, do not link directly to a semi-automatic rapid prototyping or testing process and especially lack this kind of support for preliminary, incomplete high-level models. Moreover, quickly changing requirements sometimes results in the necessity for a system to be adapted at run-time. Current RTE tools do not provide support to influence run-time objects, steered from a high-level model.

2.4.2 Requirements for Advanced Round-Trip Engineering

We propose a new RTE practice called *Advanced Round-Trip Engineering* (ARTE) to support Agile MDD. To solve the aforementioned shortcomings we define the following three criteria for ARTE:

- **ARTE includes a High-Level View**: ARTE has to support multiple views. In order to increase the similarity with human problem solving and stimulate end-user interaction we require one of the views to be as close as possible to how humans experience a problem domain in a preliminary stage: *high-level* and sufficiently abstract to be comprehensible for end-users.

- **ARTE is Agile**: ARTE has to support a highly iterative and incremental development cycle. This can be achieved by supporting two optimisations of the traditional transformations used in RTE. First, ARTE has to support *incremental transformations* [95]: only the changed elements are transformed, rather than all artifacts. Additionally, ARTE has to support *change-preserving transformations*, as required in [95]: *"information in the target artifact has to be preserved and not* *lost on the return trip"*. We refer to transformations that support both optimisations as *optimal transformations*.

- **ARTE is Dynamic**: ARTE has to consider a *run-time view* on the system under development, that is integrated in the Agile synchronisation. In order to keep the run-time view consistent with the other views in ARTE, it has to be possible that changes in the high-level view affect the elements of the run-time view.

In this dissertation we elaborate on a specific instance of ARTE, also referred to as "myARTE" that supports the above three criteria as follows:

1. myARTE has to consider an *analysis view*: In myARTE this is a static data modelling view in the Extended Entity-Relationship format [24] that is kept consistent with an implementation view. myARTE integrates a *Role Modelling* concept in the analysis view, that automatically is synchronised into an implementation and that is made operational in the run-time view.

2. myARTE has to employ *Optimal Transformations* for keeping the different views consistent.

3. myARTE has to let *Run-time Object Generations* actively participate in the RTE process. MyARTE illustrates this support by constraining run-time objects with multiplicities and dependencies in the analysis view.

2.5 Required Characteristics of ARTE Tools

In this section we elaborate on each of the three functionalities that are to be implemented by myARTE.

2.5.1 Role Modelling

myARTE considers an analysis view. As a consequence modelling constructs that are frequently encountered in analysis models, become available in the RTE process. In myARTE we want to illustrate this by including a first-class role modelling concept in the analysis view. There are at least three viewpoints to approach roles [100]:

- *Roles are named places*. The simplest notion of a role refers to labels of the relationships in modelling languages. The Enitiy-Relationship [24] and its extension add role names to the entities that participate in a relationship. Also in UML it becomes more and more accepted to replace relationship names by role names [100]. In the Object Role Model (ORM) [146] objects and roles are the sole primitives from which object (types) and association (types) are derived. ORM's extension Nijssen's Information Method (NIAM) [122] stress the linguistic role of roles. NIAM's fact types involve roles for associating properties with objects and expressing the relationships between objects.

- *Roles are types*. Roles can be seen as types that are related to the *natural types* that can play them. Intuitively the role types can be considered subtypes of the natural types. This approach requires dynamic and multiple classification as a natural type can play different role types simultaneously. However, the viewpoint on roles as subtypes is sometimes considered a misconception based on the combination of the dynamic nature of the role concept with the static properties of type hierarchies [100]. More specifically roles as types suffer from,

  > *"the paradoxical situation that, from the extensional point of view, roles are supertypes statically, while dynamically they are subtypes"* [100]

  One of the frequently used solutions is to separate the role hierarchies from the type hierarchies.

- *Roles as adjunct instances*. Roles are sometimes considered independent types whose instances include role-specific state and behaviour, but not identity. Roles are connected via relationships to the objects that play them. These objects can pick up roles dynamically and form an inseparable aggregate with these roles.

In this dissertation we agree the most on the last viewpoint on roles, i.e. roles are instances that are adjoined to the objects that play the roles. More precisely, as illustrated in figure 2.5, we consider *natural entities* and the *role entities* they can play. In the analysis view of myARTE, roles are to be integrated as a new, first class modelling concept for role relationships that connect natural entities with the role entities they can play. Each role entity in the myARTE analysis view corresponds to an implementation object in the implementation view. In the run-time view myARTE supports corresponding *natural run-time objects* and the corresponding *run-time roles*. These run-time roles are adjoined to the natural run-time objects that start play these roles dynamically.

We further elaborate on our interpretation of roles in myARTE with the help of the fifteen criteria that were defined in [100] for evaluating a vast domain of existing role approaches.

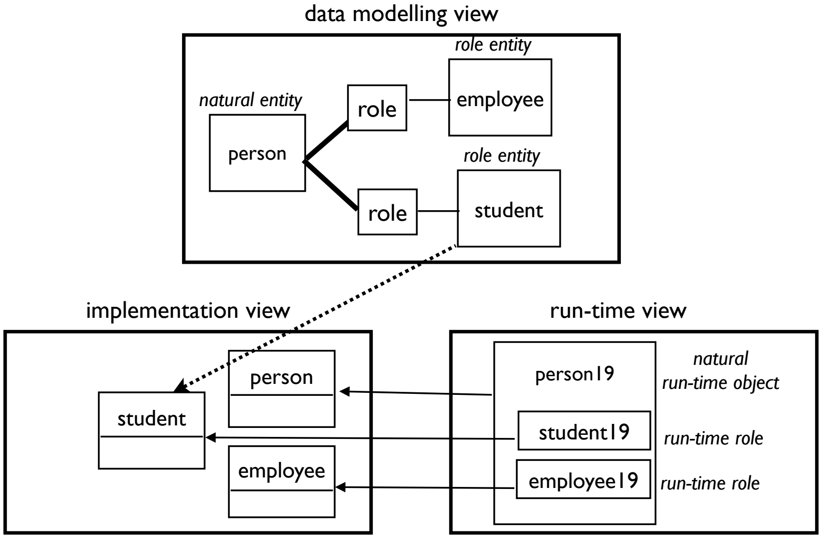


Figure 2.5: Role relationship between a natural entity and a role entity. The bold link denotes the natural entity.

1. *A role comes with its own properties and behaviour.* Yes, each role entity in the myARTE analysis view corresponds to an independent implementation object in the implementation view. In the run-time view new run-time roles are created from these implementation objects.

2. *Roles depend on relationships.* Yes, in the EER model a role entity is related to the natural entity that can play the role. In the run-time view of myARTE the corresponding run-time role objects are adjoined to the natural run-time objects that perform them.

3. *An object may play different roles simultaneously.* Yes, this is represented by multiple role relations that originate from the same natural entity in the analysis view, as illustrated in figure 2.6. Each of the corresponding run-time roles is added to the natural run-time object when it starts playing the role.

4. *An object may play the same role several times, simultaneously.* Yes, a new run-time role is added to a natural run-time object, each time it starts playing the role.

5. *An object may acquire and abandon roles dynamically.* Yes, a new run-time role is dynamically added to a natural run-time object, each time it starts playing the role. A run-time role is dynamically removed from a natural run-time object, each time it stops playing the role.

6. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* Yes, in

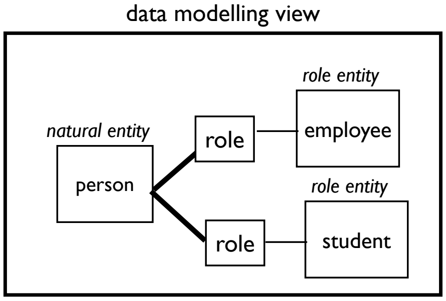


Figure 2.6: Multiple role relationships between a natural entity and two role entities. The bold links denote the natural entity.

the myARTE analysis view it can be expressed that two or more roles are *mutually exclusive* by modelling two or more role types in the same role relationship. In figure 2.7 it is expressed that engineer and salesman are mutually exclusive roles of the person natural entity. We also refer to mutually exclusive roles as *disjoint* roles. In the myARTE run-time view it then is

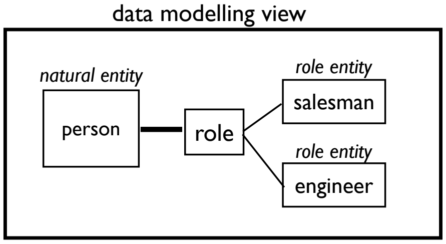


Figure 2.7: Multiple mutually exclusive role entities in a relationships between a natural entity and two role entities. The bold links denote the natural entity.

enforced that a natural-run-time object cannot start playing any run-time role that is mutually exclusive with a run-time role that the natural run-time object is already playing.

7. *Objects of unrelated types can play the same role.* Yes, since for each new role that a natural run-time object starts playing we create a new run-time role that is added to it, it is not required that these run-time objects are related. Figure 2.8 illustrates an example of such a setup: the natural entities person and company are unrelated but can play the the same role entity insured.

8. *Roles can play roles.* Yes, an entity in the myARTE analysis view can be role entity and natural entity at the same time. In figure 2.9 the employee entity is a role entity for the person natural entity as well as a natural entity for the manager role entity. At run-time this means that the

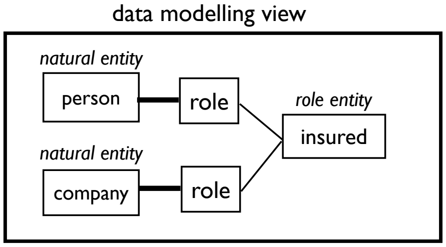


Figure 2.8: Unrelated natural entities can play the same role.

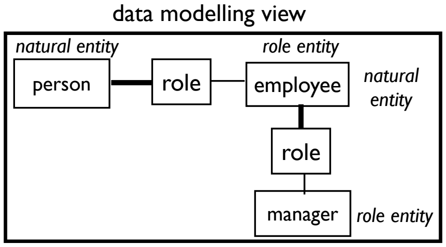


Figure 2.9: An entity can be a natural entity and a role entity at the same time.

most specialised run-time role is added to the second most specialised run-time role and so on, until all run-time roles are added to the root natural run-time object.

9. *A role can be transferred from one object to another.* Yes, in the myARTE run-time view it is possible to remove a run-time role with its specific state and behaviour from one natural run-time object, and add it to another natural run-time object.

10. *The state of an object can be role-specific.* Yes, in myARTE it is possible to address a natural run-time object from the perspective of one of its run-time roles.

11. *Features of an object can be role-specific: attributes and behaviour of an object may be overloaded on a by-role basis.* Yes, different role entities that can be played by the same natural entity are allowed to contain homonymous attributes and operations. It is possible to address the behaviour of a natural run-time object from the perspective of one of its run-time roles.

12. *Roles restrict access.* Yes, in myARTE it is possible to restrict access to the behaviour of a specific run-time role.

13. *Different roles may share structure and behaviour.* Yes, first since run-time roles are created from one implementation object they share the structure and the behaviour of this object. Second,

since roles can play roles (see feature 8), also run-time roles that correspond to different role types can share structure and behaviour, such as for example the salesman and the engineer roles in figure 2.9.

14. *An object and its roles share identity : "a role is a mask that an object can wear".* Yes, when a run-time role is added to a natural run-time object, both objects share the identity of the natural run-time object.

15. *An object and its roles have different identities, related to the counting problem.* No, based on the previous feature.

2.5.2 Optimal Transformations

To synchronise between different views, RTE tools employ specific (model) transformations. In ARTE tools the transformations are to be optimal, i.e. incremental and change preserving. Incremental transformations happen continuously after each change action and per changed node of a view onto the level of of the smallest element a node is built of. Change preservation is realised onto the level of the contents of these smallest elements. For example, an attribute's value that is set manually in the implementation view, is to be preserved during multiple synchronisations between the other views and the implementation view.

In myARTE we provide optimal transformations between an analysis view, an implementation view and a run-time view. The transformations are based on a set of *mappings* between elements of one view to elements of another view. The transformations between the analysis view and the implementation view are *bi-directional*. This means that for each mapping between an element of the analysis view and an element of the implementation view, there exists an inverse mapping between the element of the analysis view and the element of the implementation view. In the example of figure 2.10 the "is-a" relationship between car and $vehicle$ in the data modelling view maps to the inheritance relation between car and $vehicle$ in the implementation view and vice versa.

The mappings between the implementation view and the run-time view are uni-directional. More precisely, we provide mappings from the implementation view to the run-time view but not from the run-time view to the implementation view. The motivation is that the elements of the run-time view are *derived* from the elements of the implementation view. If an element of the implementation view changes, it makes sense to propagate this change to al elements of the run-time view that were derived from the changed element in the implementation view. However, if

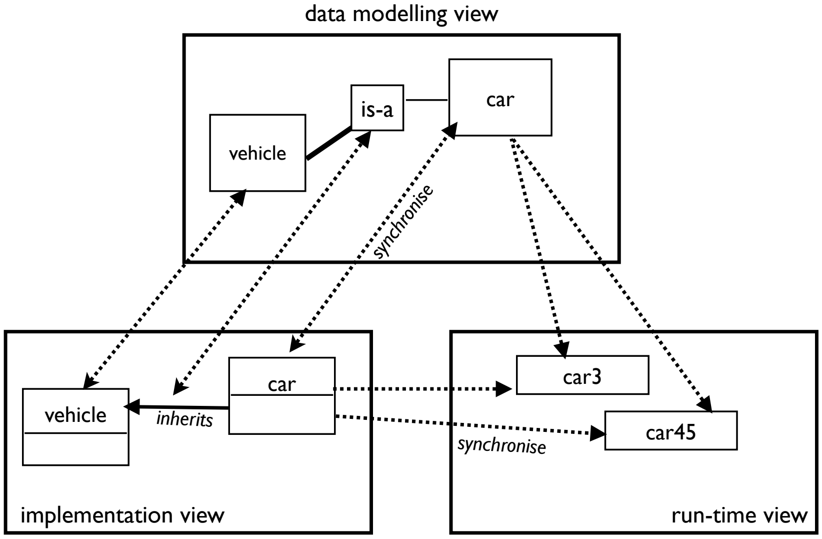


Figure 2.10: Transformations between the different views: a bi-directional mapping between the data modelling view and the implementation view, a uni-directional mapping between the implementation view and the run-time view, and a uni-directional mapping between the data modelling view and the run-time view.

one element of the run-time view changes, it is not valid to propagate this change to the element of the implementation view from which the changed element in the run-time view was derived, since then the element of the implementation view is no longer representative for the other elements in the the run-time view that were derived from it. For example, consider the implementation and the run-time views in figure 2.10. If $car3$ changes, and this change is propagated to car in the implementation view, then car is no longer representative for $car45$.

Similarly we realise a uni-directional transformation between the analysis view and the run-time view. As explained in the next section we map certain actions in the analysis view to constraints in the run-time view. For the same reasons we mentioned for the unid-directional mapping between the implementation view and the run-time view, changes in the run-time view do not affect the analysis view. Consider the data modelling and the run-time views in figure 2.10. If $car3$ changes, and this change is propagated to car in the data modelling view, then car in the data modelling view is no longer representative for $car45$. Moreover when car in the data modelling view is changed this means that car in the implementation view is also changed, making it no longer representative for $car45$.

2.5.3 Run-time RTE Programming

ARTE tools not only have to consider run-time objects in the run-time view but also need to actively synchronize and even constrain them based on the other views in the RTE process.

Run-time Object Generations

A *Run-time Object Generation* is a group of run-time objects that were created from the same implementation object. We introduce such generations in order to consider run-time objects in general and not only instances derived from classes. myARTE is required to support behavioural evolution of entire existing generations steered from the analysis view. myARTE illustrates this requirement with constraint enforcement on run-time objects based on multiplicities and object dependency.

Constraint Enforcement

In general enforcing constraints in a running system is an all but trivial requirement that has been researched extensively. In Eiffel [77] constraints are originally called *assertions*, i.e. boolean statements that never should be false. Assertions are categorised into post- and pre-conditions to be specified on a class method that have to be true before or after the method is called, and into class invariants that have to be satisfied for all instances of that class, and were first introduced via Eiffel's design-by-contract metaphor [77]. This principle was recently added in languages and tools such as iContract [65] and Handshake [32]. iContract is a Java pre-processor that supports Java expressions with assertions and is compatible with a subset of the OCL. Whereas iContract focuses more on the practical use of the design-by-contract principle, the Handshake tool [32] allows library-based run-time instrumentation of assertions in Java.

Also in the database community constraint enforcement received a significant amount of attention. [22] reports on early research on enforcing constraints expressed in a data model for a relational database. In this approach constraints are expressions in an SQL-like language that specify an inconsistent state for elements in a database table. Also in object-oriented databases the constraint enforcement is studied. In the object-oriented database O_2 [73] for example, constraints are first-class elements that are independent of any application. However, in the database domain the different ways to change the state of a database are restricted to insert, delete and update operations, as opposed to object-oriented software applications where the number of actions to change the run-time system is significantly larger.

Tool support for constraint languages such as the OCL is continuously under development.

In [119] a framework for transforming constraints from the modelling level to the implementation is defined and integrated in the Argo/UML CASE tool. Each constraint is represented as an unambiguous ten-tuple that is mapped to a separate constraint class. Four concepts are provided for characterising a constraint:

- The *scope* of the constraint that describes when the constraint has to be checked.
- The *translated form* of the constraint that corresponds to how the constraint has to be checked.
- The *rescue* of a constraint that indicates what to do when a constraint is violated.
- The *mechanism* that can be used for triggering the constraint.

The KeY-tool [3] is an extension of the Together [150] application that has extensive OCL support and that aims at detecting insufficiencies of the specification by comparing the UML design and the implementation. USE [153] is a tool for specifying information systems in a subset of UML that can be extended with additional integrity constraints in OCL. For more related approaches we refer to [119].

Multiplicity Constraint Enforcement

myARTE makes the multiplicity constraints imposed by relationships between two nodes in the graphical data modelling view, operational in the run-time view. Our ARTE instance enforces that run-time objects satisfy at all times the multiplicity constraints imposed by relationships between two nodes in the graphical data modelling view. The enforcement is based on *references* between run-time objects. We consider such a reference as a small object containing information which refers to data elsewhere, as opposed to containing the data itself.

In our myARTE instance we enforce *upper bounds*: the exact multiplicities in relationships represent the maximum number of references. More precisely, a multiplicity n on the right side of a relationship between a node `x` and a node `y` in the data modelling view (see figure 2.11) denotes that:

direction 1: the actual number of direct and indirect[3] references in each run-time object of kind `x` to other run-time objects of kind `y` is bound by the interval $[0, n]$

direction 2: the total number of direct or indirect references in run-time objects of kind `y` to each run-time object of kind `x` is bound by the interval $[0, n]$.

[3] Indirect references are references that are for example included in a collection object that is part of the constrained run-time object.

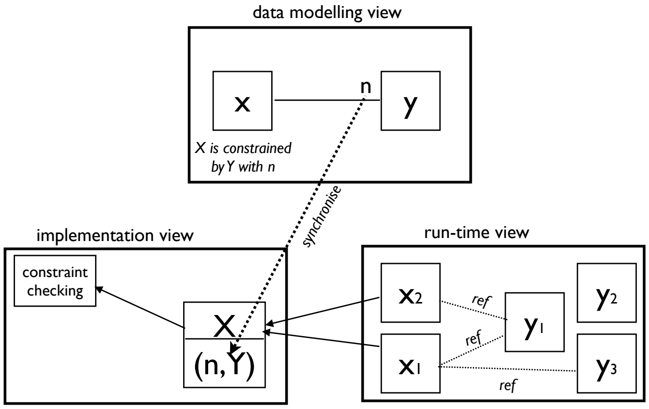


Figure 2.11: A multiplicity constraint in the three views.

We say that run-time objects of kind x are constrained by run-time objects of kind y with multiplicity n.

According to the approach in [119], the multiplicity constraints for upper bounds we focus on can be specified as follows:

- **Scope**: There exist three main cases when an upper bound multiplicity constraint has to be checked:
  - First, when a multiplicity is added or changed in the data modelling view, the entire existing constrained Run-time Object Generation has to be checked to satisfy the constraint in both direction 1 and direction 2.
  - Second, each time the state of a constrained run-time object is changed in such a way that a reference in this run-time object to another run-time object it is constrained by is added, it has to be checked wether this setup still satifies the existing multiplicity constraint in direction 1.
  - Third, each time the state of a constraining run-time object is changed such that a reference in this run-time object to another run-time object it is constraining is added, it has to be checked wether this setup still satifies the existing multiplicity constraint in direction 2. Adding a new reference in a run-time object to another run-time object can be realised by an assignment, meta-programming or via the user interface.
- **Translated form**: Our instance myARTE synchronises the multiplicities that are defined in the data model into annotations that represent an upper bound for the number of references

between the appropriate run-time objects. These annotated multiplicities are shared by the appropriate constrained Run-time Object Generations via a class variable or a variable in a traits object, cfr. figure 2.11. The checking mechanism that tests whether a constrained run-time object satisfies an upper bound multiplicity constraint is defined in a centralised object, such as a class or a traits object, that is shared by all run-time objects that are part of the ARTE process. This checking mechanism uses the annotated multiplicity that is shared by a constrained Run-time Object Generation for checking whether a specific run-time object satisfies the defined upper bound constraint.

- **Rescue**: Based on the scope of the constraint, the violation can be detected 1) after a change of a multiplicity in the data model or 2) after a state change in a run-time object that deals with adding a new reference to a constrained or a constraining run-time object.

  In the first case we generate a warning for the fact that some members of the existing population of run-time objects are inconsistent with the recently changed data model.

  In the second case the violation is detected after a specific state change in one run-time object that extends it with a new reference to a constraining or a constrained run-time object. Next to a warning it is possible to perform a compensating action by removing the added reference dynamically.

- **Triggering mechanism**: Since the checking mechanism for multiplicity constraints is stored in a shared object that is accessible to all run-time objects that are part of the ARTE process, it can be called with a method call in the run-time objects. This method is called:

  (1) for each run-time object in the appropriate constrained Run-time Object Generation, each time a multiplicity is added or changed in the data model

  (2) for one constrained run-time object after a state change in this run-time object is detected that deals with a new reference to a constraining run-time object

  (3) for one constrained run-time object after a state change in a constraining run-time object is detected that deals with a new reference to the constrained run-time object

  It has to be possible to switch the constraint checking temporarily off. For example a method that changes the state of a constrained run-time object can contain first an assignment that violates the multiplicity constraint, followed by some actions that remove a reference to a constraining run-time object. In that case we want to deactivate the enforcement before the method is called and switch it on afterwards.

Example Consider for example a `Husband` node and a `Wife` node that are in a one-to-one relationship in the data modelling view. In this case both nodes are constraining and constrained by the other one. We then first ensure that all run-time `Husband` objects refer to at most one run-time `Wife` object. Secondly, we also ensure that maximum one run-time `Wife` object refers to each run-time `Husband` object. Vice versa, all run-time `Wife` objects refer to at most one run-time `Husband` object and at most one run-time `Husband` object refers to each run-time `Wife` object.

Alternatively, consider a one-to-many relationship between a `Mother` and a `Child` node in the data modelling view. The "one side" of this relationship then constrains the other side, the `Child` node, in two ways: first a run-time `Child` object can refer to at most one run-time `Mother` object. Secondly at most one run-time `Mother` object can refer to the `Child` object at the same time. Since the "many side" lacks an explicit multiplicity there are no constraints on the number of run-time `Child` objects the run-time `Mother` objects refers to or on the number of run-time `Child` objects that refer to the run-time `Mother` objects.

Note that run-time objects that satisfy the multiplicity constraints before a multiplicity change may no longer be valid afterwards. The dynamic character of myARTE lies in the fact that whenever a multiplicity is edited in the data modelling view all previously created corresponding run-time objects should be checked to satisfy the constraint. Suppose the unspecified "many" multiplicity at the side of the `Mother` node in the data modelling view is set to 3. Then immediately all run-time `Mother` population objects that were previously not constrained by the "many side" are now allowed to reference at most 3 run-time `Child` objects and vice versa, at most 3 run-time `Child` objects are allowed to refer to each run-time `Mother` object.

Similarly, a many-to-many relationship constrains the nodes on both sides, if explicit multiplicities are provided.

Object Dependency

myARTE makes the object dependency constraints between two nodes in the graphical data modelling view, operational in the run-time view. In our instance myARTE, dependencies between a strong node x and a weak, dependent node y in a graphical data modelling view affect run-time objects[4] of kind x and y in two ways:

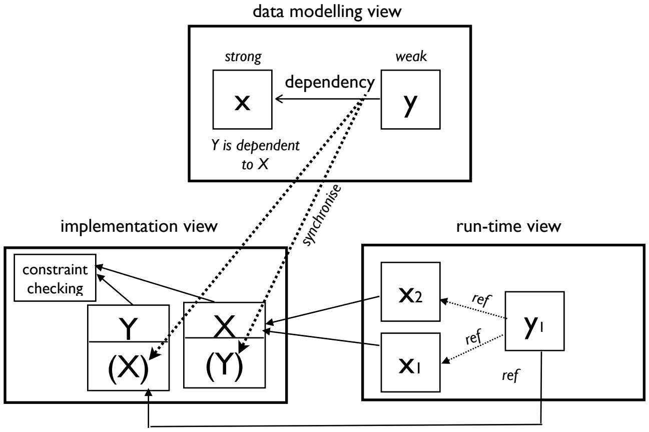


Figure 2.12: An object dependency constraint in the three views.

direction 1: Our myARTE instance enforces that a dependent run-time object always refers to a strong run-time object it can depend on.

direction 2: Our myARTE instance enforces that when a strong run-time object is deleted, all weak run-time objects that refer to it and are dependent to it are also deleted. Note that for the enforcement to be actually performed, the run-time objects that are candidates for deletion are not allowed to be referenced by any other run-time objects.

In other words a dependent run-time object can never exist as a stand-alone object. Remark that currently we do not synchronize newly added dependencies in the data model, in the run-time view: a new dependency will only affect future run-time objects. Neither do we synchronize removing dependencies in the data model, in the run-time view.

The object dependency constraints can be specified as follows:

- **Scope**: This kind of constraint has to be checked in three occasions. The first two cases deal with object dependency in direction 1 while the third handles the enforcement in direction 2:

[4] We say that the weak, dependent, run-time objects of kind y are dependent to the strong, run-time objects of kind x.

  - First, immediately after a new dependent run-time object is created and initialised it has to be checked that it contains at least one reference to a strong run-time object it can depend on.
  - Second whenever in a dependent run-time object a reference to a strong run-time object it depends on is removed, it has to be checked whether the dependent run-time object still contains at least one reference to a strong run-time object it depends on.
  - Third, when a strong run-time object with references to weak run-time objects dependent to it, is deleted, it has to be checked that these weak run-time objects are deleted also.

- **Translated form**: myARTE synchronises the object dependencies that are defined in the data model into annotations that represent 1) the kind of dependent objects in the strong objects and 2) the kind of strong objects in the dependent objects. These annotated information is shared by the appropriate strong and dependent Run-time Object Generations via a class variable or a variable in a traits object, cfr. figure 2.12.

  The checking mechanism that tests whether object dependency is enforced in the run-time objects is defined in a centralised object, such as a class or a traits object, that is shared by all run-time objects that are part of the ARTE process. This checking mechanism uses the annotated dependency information that is shared by a strong and weak Run-time Object Generations.

- **Rescue**: Based on the scope the object dependency can be violated in three cases. In the first and the second case the violation of the object dependency can be avoided by "locking"[5] this object until the user adds a reference to a strong run-time object or by the (automatic) compensating action of adding again the removed reference.

  In the third case of a violation the weak run-time objects have to be deleted or the deletion of the strong run-time object can be reversed.

- **Triggering mechanism**: Since the checking mechanism for object dependency is stored in a shared object that is accessible to all run-time objects that are part of the ARTE process, it can be called with a method call in the run-time objects. This method is called:

  (1) for each dependent run-time object after its initialisation

  (2) for one dependent run-time object after a state change in this run-time object is detected that deals with removing a reference to a strong run-time object it depends on

[5] Each message send to the object results in a warning that the object dependency is violated.

(3) for each dependent run-time object after the deletion of a strong run-time object is detected that contains a reference to the dependent run-time object

It has to be possible to switch the object dependency checking temporarily off. For example during the creation of a new dependent run-time object, the enforcement has to deactivated until the run-time object is completely initialised with the necessary references. Similarly during the deletion of a strong run-time object this enforcement has to be switched off until the delete operation is completed.

Example Consider for example a `Department` node and a dependent `Instructor` node that are in a certain relationship in the data modelling view. We then first ensure that all newly created run-time `Instructor` objects refer to a run-time `Department` object. If none of the latter are available its creation is enforced. Secondly when a run-time `Department` object is deleted all run-time `Instructor` objects have to be deleted also.

2.5.4 Relevance of myARTE Tools Characteristics in Evolution Scenarios

We can categorize the characteristics of myARTE introduced in section 2.4 by their presence in the evolution scenarios described above as summarised in table 2.3.

	S 1	S 2	S 3	S 4	S 5	S 6
Optimal Transformations	√	√	√	√	(√)	(√)
Run-time Object Generations	√	√	√	√		
Multiplicity Constraint Enforcement		√				√
Object Dependency		√			√	
Role Modelling		√			√	√

Table 2.3: Relation between the ARTE characteristics and the six evolution scenarios.

Since Optimal Transformations (cfr. section 2.5.2) describe transformations between the data modelling view and the implementation view they are relevant in scenario 1, 2, 3 and 4 where changes take place at one of these two views. When it is desired to propagate changes from the population view to the other views Optimal Transformations also make sense in scenario 5 and 6. However we have stated by default that the latter kind of change propagation is not required hence the parentheses notation in table 2.3.

Run-time Object Generations (cfr. section 2.5.3) are relevant in scenario 2 since adding a new method to a node or adding a new specialisation in the data modelling view can affect existing run-time objects in the population view. Similarly in scenario 4 adding a method or inheritance in a node in the implementation view might affect existing run-time objects in the population view.

Next Multiplicity Constraint Enforcement (cfr. section 2.5.3) occurs in scenario 2 and 6. In scenario 2 changing the multiplicity in a relationship of the data modelling view will force existing run-time objects in the population view to adhere to this multiplicity. In scenario 6 changing the relationships between population objects in the population view will trigger Multiplicity Constraint Enforcement checking.

Similarly Object Dependency (cfr. section 2.5.3) occurs in scenario 2 and 6. In scenario 2 adding a dependency relationship of the data modelling view will result in changes at the implementation view to ensure that when a new population object is created in the population view it is assured that this object can depend on another population object. Furthermore when such a dependable population object is deleted its dependent population objects should be deleted along. The aforementioned creation and deletion actions are typical changes for scenario 6.

Finally Role Modelling (cfr. section 2.5.1) occurs in scenario 2 by establishing a new role relationship between two nodes in the data modelling view. Also in scenario 5 and 6 Role Modelling has percusions as changing the role(s) of a new population object can trigger constrain enforcement of certain allowed role combinations.

2.5.5 Data Modelling Languages

In chapter 3 we elaborate on the impact that the programming language that is used in the implementation view and the population view, has on myARTE. In the next section we discuss languages that can be used in the analysis view of myARTE tools.

In general modelling implies representing the logical entities in a system and the logical dependencies between them, in a modelling language during the analysis and/or the design phase.

One of the first successful data modelling languages was based on the Entity-Relationship (ER) model [24]. This *data model* is typically used in database design and in the requirements and analysis phases of (information) system development.

Object-oriented design methods provide a specific notation to develop data models. Most of these notations or object modelling languages were integrated in the Unified Modelling Language (UML) [41].

We elaborate on the UML (section 2.6) and the ER model (section 2.7). Next we compare both languages in section 2.8.

2.6 The Unified Modelling Language

The Unified Modelling Language (UML) is an object modelling language that was developed in the early nineties by Rumbaugh, Jacobson and Booch. The UML was the result of the direct combination of the Object-Modelling Technique (OMT) [92] and the Booch method [15]. It was again the OMG that defined the UML 1.1 as a standard modelling language in 1997. The UML itself is not an analysis or a design method as it lacks a process to perform these tasks. The Rational Unified Process (RUP) [67] is a process that complement the UML to define a software engineering (design) method.

Currently the UML is the standard for object-oriented software development and for MDA.

Notation In its notation, the UML is dominated by the OMT [92] style (e.g. boxes for classes and objects). The Booch capability to specify lower-level design detail was included together with the concept and notation of *use cases* [58]. Elements from various other software development methods (e.g., *Class-Relation* [30], *CRC Cards* [10], and *OORam* [90]) were included in order to support all these methods with a single modelling language.

2.6.1 Views, Diagrams, and Model Elements

Views are abstractions that describe different aspects of a system being modeled. Views consist of *diagrams* and realize the link from the modelling language to a development method or process.

The UML provides a number of diagrams to describe both the static and the dynamic structure of a system. These diagrams are divided into three categories:

1. **Structure Diagrams:** the *Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram*, and *Deployment Diagram*.

2. **Behaviour Diagrams:** the *Use Case Diagram*, *Activity Diagram*, and *State Machine Diagram*.

3. **Interaction Diagrams:** the *Sequence Diagram, Communication Diagram, Timing Diagram*, and *Interaction Overview Diagram*.

Structure diagrams are also referred to as static diagrams while behaviour and interaction diagrams are referred to as dynamic diagrams.

The concepts used in the above diagrams are *model elements* that represent general constructs of the object-oriented paradigm such as classes, objects, messages, relationships and dependencies.

We will briefly discuss class and object diagrams.

Class Diagrams Class diagrams describe the structure of a system by showing the classes and the relationships among them. Every class is depicted as a box with the class' name. A class can display its attributes in a compartment under the name, possibly with type, initial value, and other properties. The class can also show its methods (operations) in another compartment with at least its name and optionally parameters and return type. Attibutes and operations may have their visibility marked. Classes can be linked to each other with generalization, association and aggregation relationships.

Object Diagrams Object diagrams show how specific instances of a class are linked to each other at runtime. They consist of the same elements as a class diagram. Object diagrams are used to test the accuracy of class diagrams.

2.6.2 Meta-models and the OCL

The UML's graphical notation, its syntax and its semantics are typically defined in a *specification* using meta-modelling. A *meta-model* defines what are valid models in a modelling language. In the current UML 2.0 specification the abstract syntax consists of UML class diagrams. The concrete syntax is informally specified in the UML notation. Wellformedness rules constraining the abstract syntax, specify when an instance of a particular language construct is meaningful. These rules are described in the Object Constraint Language (OCL) [142] and in natural language. The informally defined semantics of UML define a model's meaning and is also described in natural language and the OCL.

2.7 The Entity-Relationship (ER) Model

In 1976 Chen introduced the Entity-Relationship (ER) model [24] based on the mathematical foundations of set theory, mathematical relations, modern algebra, logic, and lattice theory. In that time, the three major data models were the relational, the network, and the entity-set model. Adhering mostly to the entity-set model the ER model was introduced as an abstraction layer on top of the network and the relational database model. In 1979 a new modelling technique, with *ER diagrams* to graphically represent the elements of the ER model was proposed in [25].

Around 1987, ANSI adopted the ER model as the data model for Information Resource Directory Systems (IRDS) standards. The ER model has been a driving force for computer-aided software engineering (CASE) tools. The UML also has its roots in the ER model.

Today the ER modelling approach is traditionally used as system development method and as a design method for databases

Extensions The original entity-relationship model represents exclusively static behaviour. Many extensions have been proposed to extend the original model [107]. In the context of object-oriented conceptual modelling, adding the generalization concept [83] was the most significant extension. This model is often referred to as the Extended Entity-Relationship model (EER) [34]. EER modelling concepts that are not present in the original model include: dependency, specialization, generalization, classification, and aggregation. In the remainder of this dissertation it is this extended EER model that will be considered.

Notation Currently there is no standard graphical notation for EER diagrams. Most of the differences between notations concern how relationships are specified and the way attributes are shown. In almost all variations, entities are depicted as rectangles with either pointed or rounded corners. The entity name appears inside. Relationships can be displayed as diamonds or can be simply line segments between two entities. For relationships, a name, a degree, a cardinality, and an optionality (minimal cardinality) might be represented.

Chens notation [24] consists of rectangles representing entities and diamonds representing relationships, with lines linking the rectangles and diamonds. An other well-known notation called "crow's feet" consists also of rectangles representing entities, while lines between entities represent relationships. In this case a crow's foot at the end of a line represents a one-to-many relationship. Other notations appear in database textbooks [33, 66] or existing ER modelling environments, e.g. Oracle Designer/2000 and Visible Analyst.

2.7.1 Diagrams and Model Elements

In this section we illustrate the modelling constructs of which EER diagrams are composed. The EER notation we use combines existing approaches: Chen's boxes [24], the relations of the crow's feet notation and the cardinalities of [33]. EER diagrams consist of:

- **Entities** by Chen defined as *"a thing which can be distinctly identified"* [24]. As illustrated in Figure 2.13 there exists a distinction between
  - *Strong* entities that are the default entities and that demand a primary attribute (see the next bullet) for identification.

- *Weak* entities that depend on a strong entity. This dependency implies that a weak entity can never exist without a strong entity it depends on. Intuitively when the strong entity is deleted the weak entities depending on it are to be deleted as well. The dependency relation is explicitly drawn between the two entities as illustrated in Figure 2.13. Note that the originator entity is always denoted with a blue dot while the destination entity is marked by a red one, similar to all links of our EER notation.

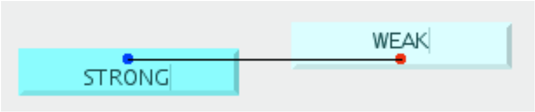


Figure 2.13: A dependency between a strong and a weak entity.

Different colours are used to denote the differences between strong and weak entities.

- **Attributes** (see Figure 2.14) are categorised into
  - *Primary* attributes, included exclusively in strong entities for identification similar to a primary key in the relational model.
  - *Simple* attributes.
  - *Derived* attributes, calculated from other attributes.

Different colours are used to denote the differences between simple, primary and derived attributes.

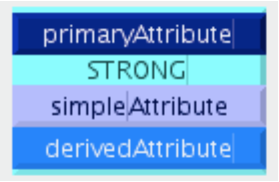


Figure 2.14: Attributes in a strong entity.

- **Relationships** (see Figure 2.15): depending on the multiplicities (also called cardinalities or arities) there is a distinction between
  - *One-to-one* relationships represent the originator entity being associated with *at most* one destination entity and vice versa.

- *One-to-many* relationships represent that the originator entity is associated with an arbitrary number of the destination entity or, if an explicit multiplicity n is provided, with *at most* n destination entities. The destination entity is associated with *at most* one occurrence of the originator entity.
- *Many-to-many* relationships represent that the originator entity is associated with an arbitrary number of the destination entity or, if an explicit multiplicity n is provided, with *at most* n destination entities and vice versa.

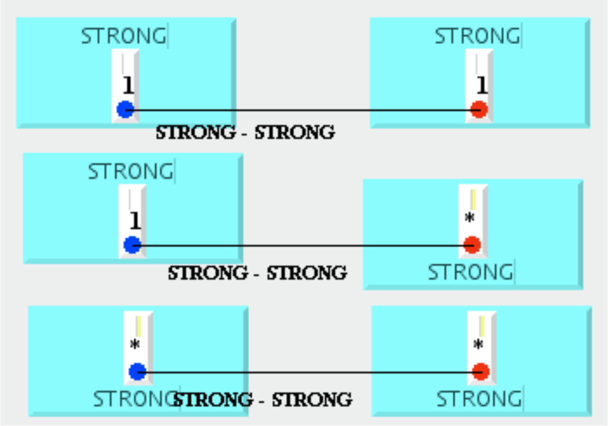


Figure 2.15: Relationships between entities.

Note that in the notation in figure 2.15 the order of the original EER multiplicities is reversed, as it is done in the Object Modelling Technique [92]. The labels of the relationships (above the cardinalities) can be filled in and function as a kind of role similar to those of Object Role Modelling (ORM) [146] or NIAM [122].

- **Specialisations** (see Figure 2.16) that represent an "is-a" relationship between entities. Note that multiple specialisation branches (links) are allowed and that the bold link always denotes the more general entity. Specialised entities in the same branch are treated as mutually exclusive (also referred to as dynamic classifications in the UML).
- **Categorisations** (see Figure 2.17) among which an object will select one to inherit from at creation time. Note that multiple categorisation branches are allowed and that the bold link always denotes the more specialised entity. Categorisation entities in the same branch are mutually exclusive.

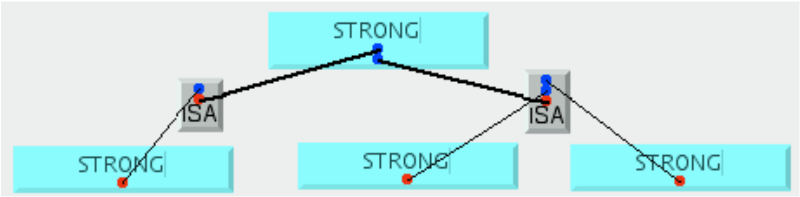


Figure 2.16: Specialisation between entities.

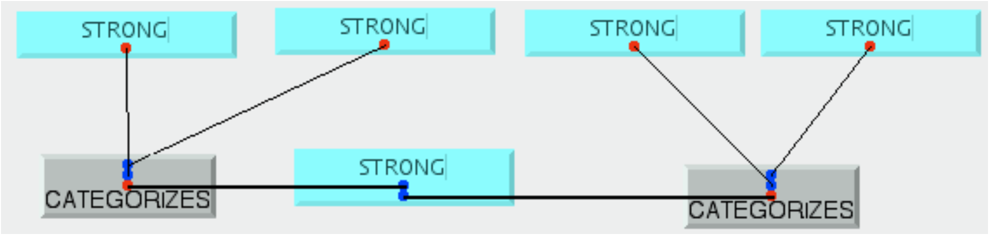


Figure 2.17: Categorisation between entities.

- **Aggregations** (see Figure 2.18) denote a whole-part relationship between an aggregate entity and the aggregated entities it is composed of.

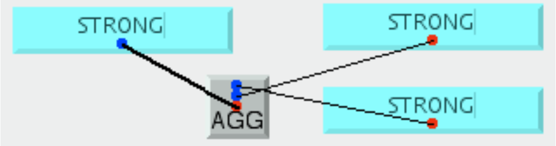


Figure 2.18: Aggregation between entities.

2.8 myARTE Tools Benefit From the EER Model

There is an almost religious discussion between the EER and the UML communities as to which approach is better. Typical claims are that EER modelling is more formally founded but that the UML is more open [93], [41].

UML class diagrams can easily be mapped to EER diagrams. David Hay [53, 54] states that for data modelling the UML is functionally equivalent to the ER model [51]:

> *"As a system of notation for representing the structure of data, the UML static diagram is functionally the exact equivalent to any other data modelling, entity-relationship (ER) modelling, or object modelling technique. Its "classes" are really "entities", and its "associations" are "relationships". It has specialized symbols for some things that are represented by the main symbols*

> *in other notations, and it lacks some symbols used in ER diagram. [...] Yes, UML does add the ability to describe the behaviour of each object class/entity, but the data structure part of the technique is fundamentally no different from any other data modelling technique in what it can represent. "*

The same author claims that UML is more appropriate for designers [51] while the (E)ER model is more suitable when end-users are involved. As a consequence, the latter can be seen as more fit for the agile principle of end-user collaboration.

Therefore, we selected the EER model to represent elements of the data modelling view in myARTE.

2.9 Conclusion

Agile MDD combines the complex models of MDD with an Agile Development cycle. There already exist various popular Agile practices, such as Extreme Programming, Rapid Prototyping and Adaptive System Development that can be applied in practices for Agile MDD, such as Agile Manual Modelling, Agile CASE and Agile MDA.

Using multiple views requires that the views that are inter-related are kept consistent. This consistency is a key factor in RTE that combines forward and reverse engineering. We focus on the kind of RTE that considers as graphical modelling view and a code-time implementation view that both view one and the same underlying model. Since Agile MDD advocates integrating changing requirements on run-time systems, RTE that is suitable for Agile MDD considers a third run-time population view.

We believe that the low success of Agile MDD practices that involve automated transformations is caused by the insufficient support of traditional RTE tools. Therefore we propose a new approach for Agile MDD called Advanced RTE (ARTE) that can be supported by the next generation of RTE tools. First, ARTE has to consider a high-level view in order to increase end-user interaction. Second, ARTE has to support optimal transformations, i.e. incremental and change-preserving transformations between the mulitple views. Finally, ARTE has to consider a run-time view that can be affected by the other views.

In this dissertation we focus on one instance of ARTE, called myARTE that supports the three ARTE criteria as follows. First, myARTE includes an analysis view for static analysis model in the Extended Entity-Relationship (EER) format. This analysis view is kept consistent with an implementation view. We illustrate this by integrating role modelling as a first class modelling concept in the analysis view. We approach roles as instances that have to be adjoined to the objects that play them. Second myARTE supports optimal transformations between an analysis view, an implementation view and a run-time view. The transformation between the analysis view and the implementation view is bi-directional. The transformations between the implementation view and the run-time view, and between the analysis view and the run-time view are uni-directional. Finally, myARTE integrates a run-time view in the RTE process. More precisely the run-time view is affected by changes in the implementation view and the analysis view. MyARTE illustrates this by grouping run-time objects in object generations and enforces constraints on these generations based on multiplicities and dependencies in the analysis view.

Chapter 3

Language Requirements for Advanced Round-Trip Engineering Implementations

In this chapter we discuss the language requirements (i.e. the characteristics) that are required by Advanced Round-Trip Engineering (ARTE), and evaluate the suitability of five programming languages as implementation candidates. In section 3.1 we identify the set of language characteristics. Next, in section 3.2 we describe the three programming language features - dynamic typing, reflection and prototypes - as indicators for the presence of the above language characteristics. We continue with an in-depth discussion on how ARTE could be implemented in C++, Java, Smalltalk, Ruby and Self as representative languages. This chapter's conclusions are summarized in section 3.6.

3.1 Programming Language Requirements

In this section we describe the programming language characteristics that are required for implementing tools that support ARTE.

3.1.1 Required Language Characteristics for Optimal Transformations

ARTE implements Optimal Transformations (see section 2.5.2) between a data modelling view and an implementation view, that are minimal and change preserving. Minimal transformations happen continuously after each change action and per changed elements onto the level of the smallest part an element consists of.

To allow such transformations we introduce the following language characteristics:

Language Mechanism 3.1 ***A Bidirectional Connection*** *implements a correspondence between the nodes in the data modelling view and the nodes in the implementation view up to the level of their smallest elements.*

This connection allows for synchronising, via a bidirectional link, between the graphical nodes of the data modelling language that is used in the data modelling view, and the corresponding nodes of the programming language used in the implementation. Conceptually spoken, it is optimal when the implementation nodes and the data modelling nodes are identical but viewed differently in different views. The smaller the "distance" between the two views, the more straightforward it becomes to realize a synchronisation.

Via the bidirectional connection, the node's elements (including their names and contents) are synchronised and preserved in both directions. To access the nodes and their elements, metaprogramming facilities to grab these different kinds of nodes are required.

To actually initiate a synchronisation, when an element of the data modelling view or the implementation view is changed, we require the following mechanism:

Language Mechanism 3.2 ***A Synchronisation Trigger Mechanism*** *initiates, each time the* state *of the data modelling or the implementation view changes, a synchronisation mechanism, that is specifc for this change, onto the other view.*

Upon a change action the above mechanism will trigger the actual synchronisation of the change in the other view, even for the smallest element of a node.

In the case that an element in one view does not have an equivalent in the other view, the information pertaining to this element has to be stored in another way, for example as an *annotation*. With this term we refer to some extra information that is stored invisibly in an object and can be consulted at any point in time.

Language Mechanism 3.3 ***A Shared Synchronisation Annotation*** *is a variable shared by a group of similar objects, that holds an annotation that is the result of a synchronisation from the EER view to the implementation view. Its value is readily available for a checking mechanism.*

In the next section, we discuss language characteristics that are required to implement Run-time RTE Programming in ARTE tools.

3.1.2 Required Language Characteristics for Run-time RTE Programming

In this section we propose a number of language characteristics that can be used to implement Run-time Object Generations, Multiplicity Constraint Enforcement and Object Dependency.

Required Language Characteristics for Run-time Generations

ARTE supports behavioural evolution of entire existing generations of run-time objects, steered from the data modelling view (see section 2.5.3). Certain changes in the data modelling view affect all existing run-time objects that correspond to the node containing the change in the data modelling view.

To support grouping run-time objects in generations, we introduce the following programming language characteristics:

Language Mechanism 3.4 ***Reified Shared Parts*** *are the shared parts of a run-time object, that are reified as a container whose identity survives after running the system.*

These parts that are shared by all the run-time objects in one generation, are reused when the system is run again after evolutionary steps. Examples of shared parts are a class in Java or a traits object in Self. Such shared parts can be used for storing Shared Synchronisation Annotations.

In order to let changes in the data modelling view or the implementation view affect an entire generation of run-time objects, their shared parts are to be modifiable:

Language Mechanism 3.5 ***Modifiable Shared Parts*** *are reified shared parts that provide operations to modify them destructively.*

In this way, changes to the shared parts, made by programmers, can be made visible to all run-time objects that already exist in the generation, and will be visible in all future ones. E.g. in some languages it is possible to add a method to an existing class, which is immediately made visible in the instances that were created from this class.

Required Language Characteristics for Multiplicity Constraint Enforcement

ARTE tools should enforce that run-time objects satisfy at all times the multiplicity constraints imposed by relationships between two nodes in the graphical data modelling view (see section 2.5.3).

Multplicity Constraint Enforcement first requires a mechanism to detect that new references have been added in a run-time object. Therefore, the following mechanism is required:

Language Mechanism 3.6 ***An Observation Mechanism for State Changes*** *monitors each change to the* state *of a run-time object.*

When a new reference is detected, a checking mechanism has to be triggered, that checks whether the run-time object respects the multiplicity constraint. In order to do so, first we iterate over all the elements of a run-time object with the following mechanism:

Language Mechanism 3.7 ***An Iteration Mechanism for References*** *iterates over the basic elements of a node in the implementation view or the population view.*

This mechanism allows for example to iterate over all the attributes of an instance of the class `Customer`.

To actually consult a specific reference in a run-time object, the following mechanism can be applied:

Language Mechanism 3.8 ***A Reference Control Mechanism*** *checks which run-time objects refer to which other run-time objects at the meta-level.*

Such a mechanism allows for inspecting for example the value of the attribute `myBanker` in an instance of the class `Customer`. The result could be an instance of the class `Employee`.

Each reference in a run-time object is matched against multiplicity constraint information. This kind of information is stored as a Shared Synchronisation Annotation. For example, adding a 1-to-many relationship between a `mother` entity and a `child` entity in the EER view, could be synchronised by annotating the `child` implementation object with `'mother'` in a variable that gathers the implementation objects that constrain `child` with multiplicity 1.

Based upon the above language characteristics it is possible to detect that the number of references to run-time objects of a constrained kind is larger than the allowed multiplicity that is contained in the shared annotation.

Required Language Characteristics for Object Dependency

ARTE enforces that a dependent run-time object always refers to a run-time object it can depend on. Moreover, ARTE enforces that when a run-time object is deleted, all dependent run-time objects are deleted as well (see section 2.5.3).

First, in order to ensure that there exists at least one reference between weak run-time objects and strong run-time objects they can depend on, we enforce the creation of such a reference at the creation time of new weak run-time objects. Therefore, we need a mechanism that detects that a new run-time object is created:

Language Mechanism 3.9 ***An Object Creation Observation Mechanism*** *observes the creation of any run-time object.*

Such a mechanism can be constructed for example by overriding a `new` method that is used to instantiate classes, with a super call to the original `new` method in combination with behaviour that reports that a new instance was created.

If a new weak run-time object is dependent, the above mechanism triggers a checking mechanism that will check all the references of the new run-time object for a run-time object it can depend on. In order to do so, the Iteration Mechanism for References that is introduced in definition 3.7 is required to iterate over all the run-time object's elements. The references themselves have to be checked with the Reference Control Mechanism (see definition 3.8), as explained before.

The dependency information that describes on which kind of object a weak run-time object can depend, is stored in Shared Synchronisation Annotations (see definition 3.3), as a Run-time Object Generation of weak run-time objects share the same dependency. In this way, it is possible to detect that there is at least one reference to a run-time object of the kind that the new run-time object can depend on.

For the second part of the Object Dependency enforcement, that deals with deletion, we require a deletion operation for run-time objects, that can propagate itself to other run-time objects:

Language Mechanism 3.10 ***A Deletion Propagation Mechanism*** *executes a number of cascading delete operations.*

If such a mechanism would for example delete a `Department` instance, then automatically all the dependent `Instructor` instances that are associated to this `Department`, are deleted as well.

When this deletion mechanism is called in a run-time object, all the references of this object are checked, with the Reference Control Mechanism (see definition 3.8) and the Iteration Mechanism for References (definition 3.7), as explained previously. Each of the referenced run-time objects is then matched against the aforementioned dependency information that is stored in Shared Synchronisation Annotations (see definition 3.3). Each of the referenced run-time objects that is dependent on the run-time object that is being deleted, is then deleted.

3.1.3 Language Characteristics for Role Modelling

myARTE tools that integrate a Role Modelling concept in the analysis view, have to automatically synchronise this concept with a suitable implementation. Roles are considered both instances that are to be adjoined to the objects that play them.

A possible implementation relies on a separation between state and behaviour, since it allows to reverse one of the specialisations by defining the reverse of state inheritance. This can help to implement relations that are either more or less specific at the level of state, and less or more specific at the level of behaviour. Hence the fact that state and behaviour are *explicitly* separated and mutually interchangeable is crucial for implementing role modelling:

Language Mechanism 3.11 ***State and Behaviour Separation*** *implies gathering object-specific knowledge (i.e. attributes) and the associated reusable knowledge (i.e. methods and class variables) in different objects.*

The reusable knowledge can be contained in the Shared Parts (see definition 3.4).

Upon starting to perform a role, the object-specific knowledge of a run-time object can be extended with the object-specific knowledge of that role. In order to do so, we require the following language characteristic:

Language Mechanism 3.12 ***State and Behaviour Addition and Removal Primitives*** *add or remove state and behaviour in run-time and implementation objects.*

These primitives enable for example to add or remove an instance variable to an instance. Another example deals with adding a method to just one instance, i.e. a *singleton* method.

To enforce that certain roles are mutually exclusive, we can use a mechanism that detects that a run-time object starts performing a new role. Such a mechanism is similar to the Observation Mechanism for State Changes that is introduced in definition 3.6. This mechanism can then trigger a checking mechanism that matches the roles of the run-time object to the mutually exclusive roles information. Such information can be stored globally, while the roles that a run-time object performs can be annotated in the object-specific part of that run-time object.

3.1.4 Conclusion

We give an overview of the language characteristics that are required to implement ARTE:

- A Bidirectional Connection (see definition 3.1)
- A Synchronisation Trigger Mechanism (see definition 3.2)
- A Shared Synchronisation Annotation (see definition 3.3)
- Reified Shared Parts (see definition 3.4)

- Modifiable Shared Parts (see definition 3.5)
- An Observation Mechanism for State Changes (see definition 3.6)
- An Iteration Mechanism for References (see definition 3.7)
- A Reference Control Mechanism (see definition 3.8)
- An Object Creation Observation Mechanism (see definition 3.9)
- Deletion Propagation (see definition 3.10)
- State and Behaviour Separation (see definition 3.11)
- State and Behaviour Addition and Removal Primitives (see definition 3.12)

3.2 Characteristics of Candidate Programming Languages

We argue that the presence of the language requirements defined in section 3.1, in a programming language, is proportional to the extent to which the language has the following three characteristics:

- **Dynamic typing** (versus static typing), as explained in section 3.2.1.
- **Meta-programming facilities** that are described in section 3.2.2.
- **Prototypes** (versus classes), as introduced in section 3.2.3

The languages that recur in the following discussions are C++, Java, Smalltalk [47], Ruby [109] and Self [145]. The reason for selecting these languages, is that they all prominently exhibit the presence or absence of the above three characteristics. C++ and Java are two mainstream object-oriented programming languages that are used in industry. They are both class-based and statically typed. Smalltalk is the oldest object-oriented class-based programming language that implements “pure” object-orientation and is dynamically typed. A rather new dynamically typed object-oriented language is Ruby that combines a pure class-based style with the flexibility of cloning. Finally, Self is the textbook example of an object-oriented prototype-based programming language and is dynamically typed.

In table 3.1 we illustrate how they fill in the above language characteristics. The reflective abilities are evaluated with a score from 0 to 2 based on criteria explained in section 3.2.2.

	Typing	Reflection	Grouping
C++	static	0	classes
Java	static	1	classes
Smalltalk	dynamic	2	classes
Ruby	dynamic	2	(singleton) classes
Self	dynamic	2	prototypes

Table 3.1: Programming characteristics of C++ , Java, Smalltalk, Ruby and Self.

3.2.1 Dynamic Typing

Nowadays, the most popular and widely-spread used programming languages are *statically typed*. In statically typed programming languages the type of variables needs to be declared explicitly before the variables are employed. Examples include Java, C++, C, ML, and Haskell. At compile-time, during static *type checking*, the variable declarations are verified and the constraints of types are enforced on them.

Recently however, a continuously growing tendency towards *dynamically typed* programming languages has emerged again. The object-oriented programming language Ruby [109], accompanied by the Ruby-On-Rails [110] scaffolding framework is rapidly becoming developers' new favourite. In dynamically typed languages such as Ruby, the variables merely need to be employed (without any declaration). Other examples of dynamically typed languages are Smalltalk, Self, Kevo, Objective-C, Scheme, Lisp, Perl, PHP, Visual Basic, and Python. Dynamic typing is often associated with so-called "scripting languages" and other rapid application development (RAD) (see section 2.1.1) environments, and tends to occur more often in interpreted languages. Dynamic type checking is performed at run-time.

In static languages such as C++ and Java, classes are considered to be the nodes in the implementation view. The smallest element of a class is an attribute (instance variable or member). Run-time objects are instances of the classes. However, these instances are not "lively" linked to their classes. Changing the implementation view always demands a compilation step.

In some dynamic class-based languages such as Smalltalk and Ruby, instances are lively linked to their classes. Classes are considered to be nodes in the implementation view, while run-time objects are instances of the classes. Changing the implementation view always is immediately reflected in the run-time objects. As a consequence, in class-based dynamic languages, **Reified Shared Parts** (see definition 3.4) are the classes themselves.

A stand-alone **Synchronisation Trigger Mechanism** (see definition 3.2) is also only efficient in dynamic languages, since in a static language the synchronisation itself demands a compilation step.

The presence of the following programming language characteristics is not considered as a consequence of dynamic typing but they are however present in most of dynamically typed languages:

- Building a **a Bidirectional Connection** (see definition 3.1) can be facilitated by a Model-View-Controller UI that is used for example in Smalltalk and Self environments. This enables to have different views on one underlying object, that can easily be synchronised.
- **Deletion Propagation** (see definition 3.10) is facilitated by the presence of an automatic garbage collector similar to the ones in Smalltalk, Ruby and Self.

3.2.2 Meta-Programming and Reflection

Meta-programming is the activity of a program observing and possibly modifying the structure and behaviour of another program. A meta-program is reflective if it observes or modifies its own structure and behaviour [71]. Reflection in programming languages deals with *Introspection* and *self-modification* [18]. Hence the evaluation between 0 and 2 in table 3.1. Also the language characteristics that were categorized at the meta-level of a programming language (see section 3.1.4) can be categorised by these two kinds of reflection.

Introspection is the ability of a program to examine and reason on the structure and behaviour of itself. The following language characteristics are used for introspection: **An Iteration Mechanism for References** (see definition 3.7), **an Observation Mechanism for State Changes** (see definition 3.6), **an Object Creation Observation Mechanism** (see definition 3.9), and **a Reference Control Mechanism** (see definition 3.8). Java, Smalltalk, Ruby, and Self provide means for introspection.

Self-modification is the ability of a program to change its own structure and behaviour. Examples of self-modifying language characteristics are **Modifiable Shared Parts** (see definition 3.5), and **State and Behaviour Addition and Removal Primitives** (see definition 3.12). Smalltalk, Ruby, and Self provide means for these kinds of self-modification.

3.2.3 Prototypes

Object-oriented programming languages can be divided depending on the basic programming concept that is used to group objects. Today the most frequently used programming languages are class-based. Classes are used to group objects of a similar kind and new run-time objects are created by instantiating the class. Another group consists of prototype-based languages (PBLs) that create new run-time objects by cloning a prototype that represents the default behaviour for a certain concept [69].

Introduction

Prototype theory was founded as a response to some of the shortcomings of the ancient class-based world view. In the late seventies the Artificial Intelligence (AI) community developed the predecessors of PBLs for knowledge representation. Henry Lieberman introduced the prototype-based paradigm into the object-oriented community in his influential OOPSLA paper [69].

In general, PBLs are considered as object-oriented languages without classes. Instead of instantiating classes, *prototypes* are cloned in a PBL. Prototypes are concrete stand-alone objects that include state and behaviour and that can be directly manipulated. The most important characteristics that are encountered in PBLs are

- **Creation ex-nihilo.** Objects can be created ex-nihilo (out of the blue) by writing a number of slots possibly with an initial filler. No template object such as a class is involved.

- **Cloning.** Alternatively new objects are created by cloning prototypes, i.e. shallow copies whose attributes are initialised with references to the same value as in the prototype's attributes.

- **Extension.** A third way to create a new object in a PBL is based on extending an existing object with new attributes or behaviour. This action is also refered to as concatenation.

- **Delegation** Instead of the class-based inheritance and "manual" message forwarding[1] of class-based languages, delegation or object-based inheritance is used to share behaviour. The variant of delegation may vary depending on the PBL. Nevertheless, they all share the idea that when a message sent to an object, is not understood, it should be automatically delegated to the parent(s) of the receiver. The delegation link can sometimes be dynamically changed.

- **Late binding** Delegation goes hand in glove with late binding of the `self` variable at the object level. This means that a method found for a delegated message, is always executed in the context of the original receiver. This dynamic meaning of the `self` variable corresponds to the dynamic meaning of `this` or `self` in a class-based language. The difference however

[1] Often erroneously refered to as "delegation". Similarly the Gang of Four's (GoF) Delegation design pattern [42] employs class-based message forwarding.

is that in this case `self` is only bound per object, instead of at the class level where the value of `self` is known at instantiation time.

- **Parent sharing** Delegation allows for parent sharing: when two objects define another one as their parent, this parent object is really shared between them. This implies that the value of the `self` pseudovariable solely depend on the children objects. Furthermore, parent sharing causes changes in the parent object that were initiated by a child object to become immediately visible in the other children.

Many PBLs have been designed until the late nineties. Examples are Self [112], Agora [102], Kevo [105] and NewtonScript [98]. A taxonomy can be found in [31].

Language Characteristics in PBLs

Since the majority of PBLs is dynamically typed[2] some PBLs such as Self also implement the language characteristics that are categorised at the typing level and are present in most dynamic languages as described in section 3.2.1. Moreover, due to the lack of classes everything *really* is an object, providing an even higher flexibility for run-time programming, compared to dynamic class-based languages.

The presence of Shared Parts in PBLs are not trivial. In a prototype-based programming language, new objects are created by cloning (copying) a prototype that represents the prototypical state and behaviour for objects of this kind. In a “pure” prototype-based programming style, each new clone is a copy of the prototype, including duplicated references to behaviour that can be shared by all objects of this kind. However, for efficiency purposes most PBLs implement a sharing mechanism to group objects of the same kind. In these shared objects, the reusable behaviour such as methods can be stored. The clones then inherit this behaviour from the shared parts.

In Kevo, for example, this is achieved with *clone families* that transparently[3] group objects of the same kind. In Self *traits objects* are made explicit to a developer and store the shared behaviour in an object and let the cloned objects inherit from it. Using traits objects can be considered as a class-based programming style in a prototype-based language. The crucial difference with classes lies in the late binding of the `self` variable, that allows traits objects to parametrize the child objects. As a consequence, methods that are defined in the traits objects, access the variables of the child objects.

[2] There exists at least one PBL that is statically typed, called Omega [12].

[3] Clone families cannot be manipulated or managed by the developer.

Reified Shared Parts (see definition 3.4) and **Modifiable Shared Parts** (see definition 3.5) are present in Self in the form of the aforementioned traits objects, in contrast to the implicit clone families in Kevo that cannot be manipulated by the developer.

An **Object Creation Observation Mechanism** (see definition 3.9) can be implemented in a PBL if it is possible to override the existing methods to create new objects.

In any prototype-based language, **Shared Synchronisation Annotations** (see definition 3.3) can be stored in a prototype. More efficiently these annotations can be stored in an object that is shared by all objects that need to access the annotated information. For example, as the value of a special variable in Kevo, that is shared by an entire clone family. Alternatively, the information can be implemented as an element of a *traits* object in Self.

Note however that C++ and Java support these annotations with static variables, while in Smalltalk and Ruby class variables can be used to store these annotations.

The possibility in languages such as Self and Kevo to add attributes and methods to one object is without doubt beneficial for ARTE. Note however, that Ruby as a class-based exception, partially offers a similar functionality with its singleton classes and singleton methods. In Ruby it is also possible to add or remove instance variables dynamically from run-time objects.

The solution based on **State and Behaviour Separation** (see definition 3.11) again builds on the separation between prototypes and objects that contain shared behaviour such as traits objects in Self. Moreover in Self the inheritance relationships between prototypes including state and traits objects including behaviour are different. For behavioural inheritance between traits objects delegation is implemented via parent pointers. State inheritance between prototypes is implemented with extension (or concatenation). Method lookup follows delegation links but not extension links resulting in a partial solution for the ambiguities caused by dynamic multiple inheritance diamonds.

In the remainder of this chapter we examine to which extent the programming languages categorised in table 3.1, effectively implement the required language characteristics that are demanded by ARTE. The statically typed language C++ is explored in the sections 3.3.1, 3.4.1 and 3.5.1 while Java's language characteristics are evaluated in the sections 3.3.2, 3.4.2 and 3.5.2. Being dynamic languages we discuss Smalltalk in the sections 3.3.3, 3.4.3 and 3.5.3 and Ruby in the sections 3.3.4, 3.4.4 and 3.5.4. The language characteristics of the prototype-based programming Self are introduced in the sections 3.3.5, 3.4.5 and 3.5.5.

3.3 Language Requirements Related to Dynamic Typing

3.3.1 C++

Maintaining a **A Bidirectional Connection** (see definition 3.1) between the modelling view and the implementational view is in principle not impossible, yet extremely hard to implement due to the fact that C++ is a statically typed programming language and that changes made to the nodes in the data modelling view might result in C++ programs that are temporarily incomplete or ill typed. Tackling this with optimistic partial type checkers is a very hard problem to solve. An important reason for this is that C++ code is text based and can in principle be mutilated by programmers in whatever way they want, as the mutilations are only detected at compile-time. In a Round-Trip Engineering environment that is to uphold links between code and more structured views such as a graphical data modelling view, this is a major obstacle.

C++ classes have no identity that survives two editing phases. As a result even small changes that have nothing to do with the actual implementation of a class can cause class versioning problems. For instance, adding or deleting the kind of dependencies and multiplicities outlined in sections 2.5.3 and 2.5.3 requires one to add or delete hidden instance variables needed for run-time checking. This changes the structure of objects and v-tables produced by the compiler and renders existing object populations obsolete. This implies that C++ classes cannot be considered as **Reified Shared Parts** (see definition 3.4) or **Modifiable Shared Parts** (see definition 3.5).

A **Synchronisation Trigger Mechanism** (see definition 3.1) is superfluous as changes always demand a compilation step to become visible in run-time objects.

3.3.2 Java

Concerning the realization of a **Bidirectional Connection** (see definition 3.1) between the nodes in the data modelling view and run-time objects that survive multiple runs of the same application, Java does not outperform C++. Java class files are composed of JVM bytecodes that are executed by the JVM. The implementation objects (forming the actual identity of classes and interfaces) are characteristiced by the virtual machine after loading the byte code.

Java bytecode instrumentation is the process of directly inserting, removing or manipulating the JVM bytecodes that compose Java class files. This transformation process must be strict and should adhere to the constraints imposed by the JVM Specification on the Java class file format. In this way any modification (insertion/deletion) of JVM bytecodes should be reflected on all other

JVM bytecodes within a Java class file. This generic process of instrumenting Java class files allows for dynamic analysis of Java class files.

There is no such thing as a class object whose identity remains the same between two runs of the program. This rules out **Reified Shared Parts** (see definition 3.4) or **Modifiable Shared Parts** (see definition 3.5). In other words, adding a method or a field to a class (or changing an existing one) requires one to recompile the class and this results in class versioning problems between population objects of an old class, and the new class.

Another important drawback is that most modelling languages allow one to "draw" code that is technically not sound w.r.t. static typing.

3.3.3 Smalltalk

A **Bidirectional Connection** is facilitated by the fact that classes and their instances are "alive" in the Smalltalk image. Instances are at all times linked to the object that represents their class, which means that when the class is modified, the instances can be retrieved and immediately updated. Moreover the "openness" of the Model-View-Controller-based user interface allows the construction of new graphical elements.

A **Synchronisation Trigger Mechanism** (see definition 3.2) could be realized by overwriting some behaviour that observes or propagates changes between the Model and the View parts of the user interface.

It is possible to dynamically store **Shared Synchronisation Annotations** (see definition 3.3) in class variables, which are then shared by all instances of that class.

Reified Shared Parts (see definition 3.4) are classes in Smalltalk while **Modifiable Shared Parts** (see definition 3.5) are supported since classes can be modified dynamically, which results in existing and new instances reflecting those modifications. Instances are at all times linked to the object that represents their class, which means that when the class is modified, the instances can be retrieved and immediately updated.

3.3.4 Ruby

Ruby is a class-based, dynamically typed, interpreted scripting[4] language whose syntax resembles those of Eiffel and Ada. It is a pure object-oriented language where everything is an object and all manipulation of objects is initiated by sending messages.

[4] Denoting the activity of connecting diverse pre-existing components to accomplish a new task

As in Smalltalk a **Bidirectional Connection** (see definition 3.1) is supported by the live link between classes and their instances. Also similar to Smalltalk **Reified Shared Parts** (see definition 3.4) are classes while **Modifiable Shared Parts** (see definition 3.5) are supported since the classes can be modified dynamically and changes are reflected immediately in the instances.

Shared Synchronisation Annotations (see definition 3.3) can be stored in class variables that start with two @ symbols.

3.3.5 Self

Self is a dynamically typed prototype-based language that resembles Smalltalk in both syntax and semantics.

A **Bidirectional Connection** (see definition 3.1) is facilitated by the fact that everything really is an object. In this way the difference between prototypes and its clones is blurred.

Objects in Self consist of slots and are created ex-nihilo by putting slot names (together with a possible initial filler value for that slot) between vertical bars, separated by dots. The following code, for example, creates an ex-nihilo `point` object:

```
point = (|parent* = traits clonable. x = 3. y = 4.
            addPoint :point = ((clone x: x + point x) y: y + point y)|)
```

A slot marked with an asterisk is a parent slot and makes the child inherit all the slots of the parent slot. In this way, `point` inherits (its behaviour) from the traits object `clonable`[5]. Further the point object has two data slots containing an `x` and a `y` coordinate. The remaining method slot contains a method for adding two points, by *cloning* `point` and initialize it with the added `x` and `y` coordinates.

Furthermore Self offers facilities to declare the visibility of a slot: it can be marked as `undeclared` (by default), `public` or `private`. However, these annotations have a merely lexical purpose for the developer since it still is possible to access private slots from outside the object in question. This implies that Self supports no real encapsulation of attributes.

Self comes with an extended visual programming environment. The Self user interface is built with the Morphic framework [72] and applies a Model-View-Controller architecture [47]: a real Self object (the model) can be viewed with a graphical representation called an *outliner* (the view) and be manipulated via the appropriate menus on this outliner (the controller).

[5] Most concrete not-unique objects in the SELF world are descendants of the top-level traits object `traits clonable`.

A **Synchronisation Trigger Mechanism** (see definition 3.2) is achieved by overwriting the behaviour that detects changes or is responsible for Synchronisation between the Model en the View part of the Self user interface.

The **Bidirectional Connection** (see definition 3.1) also depends on this user interface as it facilitates the synchronisation between an implementation view and an abstract data view.

Shared Synchronisation Annotations (see definition 3.3) can be stored in variables of traits objects.

Since Self has garbage collection **Deletion Propagation** (see definition 3.10) is achieved by removing all references to the object. When the clones in the data modelling view are explicitly deleted, this causes recursively the explicit deletion of all dependent clones.

3.4 Language Requirements Related to Reflection

3.4.1 C++

It is possible to declare a pointer to any member function but this is the only "reflective" property of the language. This implies that the language characteristics that were dependent of the reflective abilities of a language are hardly or not available in C++.

For example there exists no **Iteration Mechanism for References** (see definition 3.7) since there are no provisions for iterating over the field members of an object. Also an **Observation Mechanism** (see definition 3.6) is absent as it is not possible to monitor state changes performed on an object. Neither it is possible to monitor object creations the way it is defined in the **Object Creation Observation Mechanism**[6] (see definition 3.9). Similarly **State and Behaviour Addition and Removal Primitives** (see definition 3.12) are not present in C++.

Without extensive transformations of the source code, dependency checking and multiplicity checking characteristics cannot be achieved at all. This eliminates language characteristics such as **A Reference Control Mechanism** (see definition 3.8) and **Deletion Propagation** (see definition 3.10).

3.4.2 Java

Java has a number of introspection primitives that outperform the ones of C++ considerably.

Interception of object creation in an **Object Creation Observation Mechanism** (see definition 3.9) and assignments in an **Observation Mechanism** (see definition 3.6) are not supported and as such

[6] Unless a rigourous discipline is applied involving operator and constructor overloading.

the dynamic model checking possibilities in Java are the same as the ones for C++.

The problems of C++ concerning the incorporation of multiplicity and dependency checking facilities, described in **A Reference Control Mechanism** (see definition 3.8) and **Deletion Propagation** (see definition 3.10), remain in Java.

Without extensive code transformations, these characteristics are hard to implement. One possibility is to use Java's class loader machinery which allows one to change the behaviour of a Java program by changing its byte codes before they are executed by the virtual machine. Again, studying the impact of these ideas on Round-Trip Engineering is beyond the scope of this dissertation.

3.4.3 Smalltalk

Smalltalk provides advanced meta-programming capabilities, which are defined as methods on the root class `Object`. As a result, these methods can be called on any Smalltalk object. Examples of such methods are **`perform:`** and its variants for sending a message to a receiver, **`instVarAt:`** for accessing an instance variable of an object and **`instVarAt:put:`** for assigning a value to an instance variable.

However, whereas these methods can be used, they cannot be adapted by means of overriding, since calls to such methods are converted to calls to primitives when code is compiled to byte codes. As such, it is not possible to add observation of state changes induced by assignments in the standard Smalltalk language or development environment. This means that there is no full support for an **Observation Mechanism** (see definition 3.6).

Moreover, Smalltalk supports **An Iteration Mechanism for References** (see definition 3.7) by means of the aforementioned meta-programming capabilities, in particular `allnstVarNames` and `instVarAt:`.

A **Reference Control Mechanism** (see definition 3.8) can retrieve all the objects that refer to a certain object via the `allOwners` method.

This feature in combination with the previous one allows for iteration over all the referring objects of a certain object, to iterate over all their instance variable values and setting the ones referring to the first object to `nil`.

An **Object Creation Observation Mechanism** (see definition 3.9) is not supported in standard Smalltalk.

Deletion Propagation (see definition 3.10) can be ensured by deleting all references to objects that need to be deleted.

Finally **State and Behaviour Addition and Removal Primitives** (see definition 3.12) are available in the methods `addInstVarName:` and `removeInstVarName:` defined in `Class`, respectively.

3.4.4 Ruby

Ruby enables reflection. The use of the term "metaclass" is a controversial issue in Ruby. The class `Class` is the class of all classes, but it is not agreed on as a metaclass. However, every object (including classes) can have a hidden meta-object, which is only revealed via the "<<" singleton class notation, for example `class << Point`. This meta-object is – correctly or not – often refered to as the metaclass. As in Smalltalk the class `Object` is the root of the class hierarchy.

Ruby supports a powerful observation mechanism. Ruby enables the detection of specific Ruby events based on *hooks*, i.e. a technique that enables to "trap" these events. The simplest hook technique in Ruby is to intercept calls to methods in system classes. More advanced hooks support an **Object Creation Observation Mechanism** (see definition 3.9).

This allows for example to wrap the new instance, add or remove methods, and add them to containers to implement persistence. Ruby enables warning of the events via a *callback* method. There are callback methods for adding an instance method or a singleton method, subclassing a class and mixing in a module.

Ruby blurs the distinction between design and implementation by its support for design-level strategies. The Visitor pattern is a way of traversing a collection without having to know the internal organization of that collection. The Delegation pattern implements a way of composing classes more flexibly and dynamically than can be done using standard class-based inheritance. The Singleton pattern is a way of ensuring that only one instantiation of a particular class exists at a time. Finally, the Observer pattern implements a protocol allowing one object to notify a set of interested objects when certain changes have occurred. Normally, all four of these strategies require explicit code each time they're implemented. With Ruby, they can be abstracted into a library and reused transparently.

There exists exclusively single inheritance in Ruby. To extend objects with multiple other objects Ruby provides *mixin modules*. Mixin inheritance is based on the idea that instead of inheriting from an existing super class a mixin class is created to hold the specialised behaviour. Mixin classes are abstract classes. When a Ruby class includes ("mixes in") such a module, that module's methods become available as methods of the class as if it was a superclass.

Therefore an **Observation Mechanism for State Changes** (see definition 3.6) is supported via the Observer pattern, a simple mechanism for one object to inform a set of interested third-party objects when its state changes. In the Ruby implementation, the notifying class mixes in the `Observable` module, which provides the methods for managing the associated observer objects. Run-time objects can be added or removed as an observer on other objects at run-time. It is possible, after a state change, to invoke an update method in each currently associated observer.

Ruby also supports an **Iteration Mechanism for References** (see definition 3.7). Each run-time object can be represented as an array with `to_a`. It is then possible to iterate over this array Ruby's **Reference Control Mechanism** (see definition 3.8) uses the previous characteristic on the `ObjectSpace` module that allows to iterate over all run-time objects. It then consults the type of these elements via `class` or `instanceOf?`.

Furthermore **State and Behaviour Addition and Removal Primitives** (see definition 3.12) are available in `instance_variable_set(@<name>,<contents>)`. In order to delete instance variables from an object `remove_instance_variable(<name>)` is used.

3.4.5 Self

Self has a powerfull mechanism for reflection based on *mirrors* [18]. These objects ensure that the base-level functionality and the meta-level functionality are implemented separately as opposed to languages such as Smalltalk where classes function both as base-level objects e.g. to create new instances and as meta-level objects. Such an approach complicates removing the reflection from applications so it is no longer accessible to programmers. Mirrors have several advantages [18]:

- The ability to write meta-programming applications that are independent of a specific meta-programming implementation such as reflection.
- The ability to obtain meta-data from systems executing on platforms that do not themselves include a full reflective implementation.
- The ability to dynamically add/remove reflection support to/from a running computation.
- The ability to deploy non-reflective applications written in reflective languages on platforms without a reflective implementation for example to save communication time.

The fact that some functionality such as printing does not clearly fall into the base or the meta-level is a disadvantage of using mirrors.

Each Self object returns its mirror by sending it the `asMirror` message or the `_Mirror` primitive. Alternatively the object can be called as argument of the `reflect:` method. The resulting mirror then is manipulated to change the original object (reachable as `reflectee`) at the meta-level. The behaviour of Self mirrors is contained in the `traits mirrors abstractMirror` object. This extended traits object includes mainly categories for introspection, for example to examine the annotations of an object or to iterate over its parents, and for self-modification (including dynamically generated code execution), for example to change objects by adding, removing or changing the contents of slots. For more details on mirrors we refer to [18].

An **Observation Mechanism for State Changes** (see definition 3.6) in self deals with three types of state changes. Self objects can be manipulated 1) via their outliner using the appropriate menu (controller), directly (editing slots and pointer dragging-and-dropping) or by evaluating messages in their context and 2) directly by assignments or meta-programming. We observe the state changes resulting from such manipulations in several ways. Change observation for outliners comes with Self's user interface based on the MVC architecture. In order to immediately update the outliners (views) when necessarry, the Self system constantly checks the real, represented objects (model) for changes. Change observation in the case of meta-programming can be realized in the behaviour of meta-objects (mirrors): e.g. in the method `at:PutContents` and `addSlots:`.

For assignments the situation is more complicated. In Self for each assignable data slot in an object a corresponding assignment slot is added transparently. This assignment slot contains the assignment primitive for which the compiler generates the appropriate code. This implies that it is – to the best of our knowledge – not possible to intercept assignments by changing the assignment slot directly via meta-programming. SelfSync circumvents this via *assignment shadowing*. Between the object containing the assignment and all its referring objects an object is dynamically mixed in that contains a shadowing method that overrides the assignment. In the shadowing method a `resend` (super) is called with the original assignment, after the actions to be associated with the observation of the assignment have been undertaken.

Grabbing all referencing objects is implemented with `referencesOf:` defined on `browse`[7] which can serve as a **Reference Control Mechanism** (see definition 3.8). Mixing in an object is realized by destructively replacing all real references[8] (reachable via `at:` defined on mirrors.) of the object containing the assignment by the object that shadows the assignment with `at:PutContents:`.

[7] An object allowing to browse the entire system based on different criteria.

[8] A number of referencers are *fake* slots (checked with the predicate `isFake`).

In Self an **Iteration Mechanism for References** (see definition 3.7) when the receiver of the message `do:`*[block]* is a mirror object, *[block]* iterates over the slots of the original object the mirror reflects. When slot references between clones are established, we can verify whether there are no more references to objects of a specific type (i.e. the prototype from which it was cloned) than the allowed multiplicity.

The prototype an object was cloned from can be found by sending the message `creatorSlotHint`. Although one of the fundamental ideas of PBLs is shallow copying, this operation is used e.g. for naming object outliners (*a* vector, *a* list, ...). As such, Self allows for a kind of dynamic type checking on clones.

Self allows us to implement an **Object Creation Observation Mechanism** (see definition 3.9) by overwriting the cloning methods of an object to ensure that in case of a dependent clone there is at least one reference to a clone it can depend on.

Based on the slot iteration described above Role Combination Enforcement can be realised. If the prototype of an object in a certain slot (of an object playing roles) is contained in one of the illegal combination sets of the role combination information, all other slots' content will be compared to the other role prototypes in the set.

State and Behaviour Addition and Removal Primitives (see definition 3.12) are in Self the meta-constructions `at:PutContents:` and `removeSlot:` that are used to respectively add and remove data or parent slots.

3.5 Language Requirements Related to Prototypes

3.5.1 C++

Finally **State and Behaviour Separation** (see definition 3.11) is not included in C++ as the description of state and behaviour is gathered in classes. **Shared Synchronisation Annotations** (see definition 3.3) can be stored in static variables.

3.5.2 Java

Code and data cannot be separated in the structure of the code as implemented in a **State and Behaviour Separation** (see definition 3.11).

A small help however is that Java allows one to separate the description of behaviour in interfaces from the actual realization of that behaviour in classes. Nevertheless, behaviour and data still reside together in classes such that the mass of pointers and manual delegation code explained above are also necessary in Java. Hence, interfaces only make the static typing aspects of role modelling slightly easier. The implementation however remains as hard as in C++ as described in [38].

3.5.3 Smalltalk

Although there is no support for **State and Behaviour Separation** (see definition 3.11) role modelling can be simulated in Smalltalk in a cumbersome and ad-hoc way.

However, in a static data modelling view all possible roles and allowed combinations thereof are modelled, but individual run-time objects can each play different actual roles. Therefore, it is not sufficient to extend a class with the instance variables belonging to a role, since this results in all instances of the class playing that role. Instead, one can remedy this by creating a new class based on the old class of an object and the roles that object has to play, and making the object an instance of this class by means of the method `become:`.

However, this method is defined on objects, hence one has to create an instance of the merged class, copy the values of the instance variables of the first object to the new one, and then change the class. Moreover, such a class has to be generated for each combination of roles that objects of a class can have.

This highly complex procedure becomes even worse if particular roles are removed from an object: one has to be able to select the correct generated class that results from removing these roles, and let this object become an instance of this class in a similar way as described above.

3.5.4 Ruby

Ruby is class-based but supports singleton characteristics that simulate a prototype-based programming style. To extend the functionality of a class for just one object Ruby applies the concept of a *singleton class* (notation: “class << *obj*”), i.e. an anonymous class that is created by subclassing the class to be extended.

It is also possible to assign to one method different visibilities in different subclasses. If a subclass changes the visibility of a method in a parent, Ruby effectively inserts a hidden proxy method in the subclass that invokes the original method using super. It then accordingly sets the visibility of the proxy.

Instances can be changed dynamically via instance methods associated with one specific object. These *singleton methods* reside in the object’s hidden meta-clas. The use of such singleton methods implies that instances of the same class can behave differently. Next these changed objects can be copied using the `dup` or the `clone` method resulting in a new generation of objects without the need for a new class. While `clone` is used to duplicate an object, including its internal state (including references), `dup` uses the class of the object to create the new instance.

A **State and Behaviour Separation** (see definition 3.11) in Ruby could be based on singleton methods. The methods of a role class can be added as singleton methods in a run-time object that starts performing a role. Similarly they can be removed when the run-time objects stops playing the role. In this way **State and Behaviour Addition and Removal Primitives** (see definition 3.12) are provided.

Alternatively delegation is considered extending an object with the capabilities of another at run-time. This allows writing decoupled code, as there are no compile-time dependencies between the users of the overall class and the delegates.

The Ruby `Delegator` class implements a simple but powerful delegation scheme, where requests are automatically forwarded from a master class to delegates or their ancestors, and where the delegate can be changed at run-time with a single method call. Using its subclass `SimpleDelegator` results in an object that both has its own behaviour and delegates to different objects during its lifetime. This is an example of the State pattern, as Fowler states in [38] one of the most elegant ways to implement roles in a class-based language.

3.5.5 Self

Traits can be considered **Reified Shared Parts** (see definition 3.4), as they group behaviour (i.e. methods) for all objects cloned from a certain prototype in Self. Traits are persistently stored in a namespace and thus survive different system runs. Traits are **Modifiable Shared Parts** (see definition 3.5) since dynamically changing a method slot in a traits object affects all objects that share this parent. This is achieved with `at:PutContents:` to destructively change (method) slots and with `parseObjectBody` to parse a string into a method body. Deleting a method slot is done similarly with `removeSlot:`*selector*.

Intermezzo on Multiple Inheritance in Self Self allows multiple inheritance. When implementing a conceptual data type in Self, the state (specific for each “instance” of this type) is contained in a prototype while the behaviour (shared by all objects of this data type) is typically gathered in a traits object. All prototypes inherit their behaviour from their own, shared traits object, which in its turn often inherits from `traits clonable`:

```
point = (|parent* = traits point. x .  y.|)


traits point =(|parent* = traits clonable.
               addPoint :point = ((clone x: x + point x)
               y: y + point y)|)
```

To obtain a new point, we clone the point prototype and set the `x` and `y` coordinates.

```
(point copy x: 1) y: 2.
(point copy x: 3) y: 4.
```

Both points now share the `traits point` object since they both contain a copy of the `parent*` pointer of the prototypical `point`, i.e. the most common form of parent sharing.

In order to implement for example a *coloured* point, the state and the behaviour are to be inherited from a normal point. Therefore, a prototypical `coloured point` is created that inherits its behaviour from a corresponding `traits coloured point` object. Naturally, the `traits coloured point` inherits behaviour from the `traits point`, since the behaviour of a coloured point will be a specialisation of a normal point's behaviour.

On the other hand, the `coloured point` prototype can inherit the coordinates of the normal `point`, and extend them with an extra slot to contain the colour, see figure 3.1. However using parent pointer inhertance between prototypes is not feasible. Remark that the multiple inheritance structure in figure 3.1 is a diamond. In this context, the common parent `traits point` is shared by all descendants. Imagine a method `m` in `traits point` that is overridden in `traits coloured point`. When we now send the message `m` to a `coloured point` we get a *name collision*: the method lookup algorithm finds a method slot[9] `m` in `traits coloured point` (overriding method) but also in `traits point` (original method) via the state inheritance link with `point`. In this case, it should be decided which method should be invoked in which context.

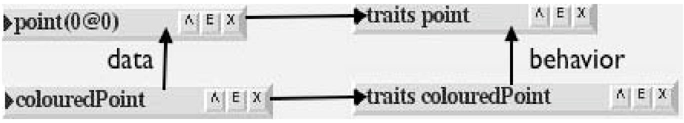


Figure 3.1: `coloured point` inherits state and behaviour from `point`

[9] Both methods and variables are contained in slots, therefore this discussion holds for all attributes

The early version of Self solved this ambiguity with language mechanisms such as *prioritized* parents or the *tie-breaker sender path* rule, which proved to be rather unsatisfying. In the current version of Self ambiguous methods are to be resolved manually by adding a *directed resend* in coloured Point. Calling m = (traits colouredPoint.m) would invoke the overridden method while m = (traits point.m) would return the original method. But then we violate the principle of traits-based inheritance, since we add shared behaviour in a prototype in stead of into the corresponding traits object.

A more acceptable solution is to perform a copy-down of the point prototype: this concatenation mechanism for state inheritance copies (some of) the slots of the receiver into a new object, ensuring that changes (adding/removing slots) to the receiver are propagated to all copied-down children. Next, we override the parent* pointer with the traits colouredPoint object. In this way, colouredPoint inherits all the slots of point except for its parent: this implies that there are no name collision when traits coloured point override methods of traits point In fact, copy-down allows a kind of class-based programming: copy-down can be considered as creating a subclass. The code for a coloured point thus looks like:

```
traits colouredPoint =(|parent* = traits point.
                        print = (...)

colouredPoint = (point createSubclass)
               _AddSlots: (|parent* = traits colouredPoint.
                            colour<-'none'|)
```

A **State and Behaviour Separation** (see definition 3.11) is realized by the distinction described above between prototypes and traits. The data slots corresponding to the roles an object plays can be dynamically copied or removed into and from the object itself. In order to get the desired behaviour, the traits containing the additional behaviour of the role is mixed in between the object and the traits it is currently referring to. When the role is removed, the traits is set back to the original traits and the slots that were added to the object are removed from it. This requires dynamic behaviour modification (in order to switch to the behaviour of the role) and dynamic data additions and removals from objects that do not change the identity of the object.

3.6 Conclusion

Table 3.2 summarizes the presence of the discussed language characteristics in C++, Java, Smalltalk, Ruby and Self.

	C++	Java	Smalltalk	Ruby	Self
Bidirectional Connection	ϕ	ϕ	$\surd$	$\surd$	$\surd$
Shared Synchronisation Annotation	ϕ	ϕ	$\surd$	$\surd$	$\surd$
Synchronisation Trigger Mechanisms	ϕ	ϕ	$\surd$	$\surd$	$\surd$
Deletion Propagation	ϕ	ϕ	$\surd$	$\surd$	$\surd$
Observation Mechanism for State Changes	ϕ	ϕ	ϕ	$\surd$	$\surd$
An Iteration Mechanism for References	ϕ	$\surd$	$\surd$	$\surd$	$\surd$
A Reference Control Mechanism	ϕ	ϕ	$\surd$	$\surd$	$\surd$
An Object Creation Observation Mechanism	ϕ	ϕ	ϕ	$\surd$	$\surd$
Reified Shared Parts	ϕ	ϕ	$\surd$	$\surd$	$\surd$
Modifiable Shared Parts	ϕ	ϕ	$partially$	$\surd$	$\surd$
State and Behaviour Addition and Removal Primitives	ϕ	ϕ	$partially$	$\surd$	$\surd$
Separation of State and Behaviour	ϕ	ϕ	$\surd$	ϕ	$\surd$

Table 3.2: Advanced RTE language characteristics in C++, Java, Smalltalk, Ruby and Self.

We illustrated using the languages C++, Java, Smalltalk, Ruby and Self that the three language characteristics dynamic typing, reflection and prototypes can be considerd as indicators for the presence of the language characteristics demanded by ARTE.

Statically typed class-based languages such as C++ and Java in general lack a live connection between classes and instances and also reflective power. Due to the dynamic character of Smalltalk and its meta-programming capabilities it is in theory possible to simulate the role modelling features of ARTE, but it introduces an enormous management overhead. In essence, this management concerns generation of new classes and making existing objects instances of these classes and back, which are basically complex and error-prone manoeuvres for avoiding classes in Smalltalk.

Ruby takes this one step further with singleton methods and singleton classes that can function as prototypes. However Ruby still lacks straightforward characteristics to implement role modelling with the Warped Inheritance Hierarchies we introduce in chapter 4.

Finally, it is Self that combines the high flexibility of "everything is an object" with the efficiency of traits objects and implements in this way all language characteristics that are required by ARTE.

Chapter 4

A New Practice for Advanced Round-Trip Engineering

In this chapter we introduce a new practice for Advanced Round-Trip Engineering (ARTE) that can be applied in Agile Model-Driven Development (MDD). In order to do so, we instantiate a concrete ARTE approach called myARTE that considers three views on a system under development. myARTE uses different representation for the different views: the analysis view is represented in the Extended Entity-Relation format while both implementation and run-time view are represented in Self[1] (see section 4.1).

The transformations in myARTE are based on a set of default mappings between the elements of the EER view, the Self implementation view and the run-time population view. In section 4.3 we provide mappings between the modelling elements of the EER view and the Self implementation objects represented in the implementation view. We elaborate on the specific mapping from roles in the EER view to a setup of implementation objects in the implementation view, that is based on reversed inheritance hierarchies (see section 4.4).

The population view and the implementation view include both Self objects and thus share the same internal and graphical representation. The mappings from the implementation view to the run-time view are explained in section 4.5. Finally, as one of the contributions of our approach we let the EER view influence run-time population objects. The mappings between the EER view and the population view are summarised in section 4.6.

From a methodologic view point, myARTE consists of two phases: Active Modelling and Inter-

[1] In Appendix B we provide a glossary for the concepts in both views, that are specific for Self and for prototype-based languages. We informally describe the elements of the different views and provide a formal definition of implementation objects and run-time objects in section 4.2.

active Prototyping, that together form an Agile Development cycle (see section 4.7). Based on the default mappings between the views, we systematically construct usage scenarios for myARTE, in order to illustrate the synchronisation process (see section 4.8). The conclusions for this chapter can be found in section 4.9.

4.1 Representation of the Different Views

In this section we describe the different notations and implementation languages that are selected for representing the different views in myARTE and how these representations can be combined.

4.1.1 An EER Notation for the Analysis View

For the static data models in the myARTE analysis view we employ the Extended Entity-Relationship model that was described in section 2.7 together with the notation introduced in section 2.7.1. We implemented our EER modelling language in a graphical drawing editor with the help of the Morphic framework [72]. The supported modelling constructs in EER diagrams thus are

- **Entities**
    - *Strong* entities
    - *Weak* entities
- **Attributes**
    - *Primary* attributes
    - *Simple* attributes
    - *Derived* attributes
- **Relationships**
    - *One-to-one* relationships
    - *One-to-many* relationships
    - *Many-to-many* relationships
- **Specialisations**
- **Categorisations**

- **Aggregations**

Furthermore, we extend this EER model with a new modelling construct. **Role Modelling**[2] represents that an entity can (temporarily) perform the role of other entities at run-time, as illustrated in figure 4.1. We explained our viewpoint on roles as instances to be adjoined to the objects that play them, and the semantics of the role modelling concept in myARTE, in section 2.5.1. The implications of this role modelling concept in the implementation and the population view are explained in section 4.4.

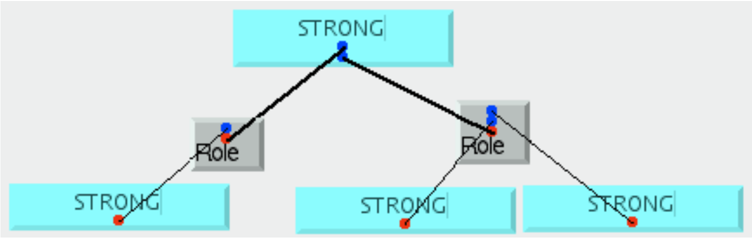


Figure 4.1: Roles of entities.

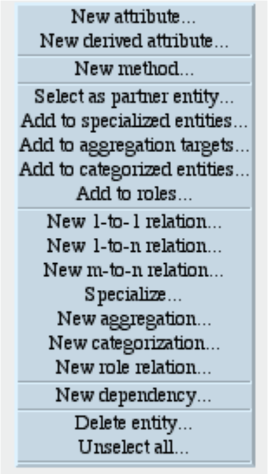


Figure 4.2: A menu to manipulate entities.

On each entity in the drawing editor a menu can be called to manipulate the entity, as illustrated in figure 4.2. The entity can also be manipulated directly by clicking and editing its name or the names of its attributes.

[2] Not to be mistaken with the roles entities play in relationships.

An Example

In Figure 4.3 an EER diagram of a modest banking system is shown. We elaborate the customer entity. A Customer has a primary attribute customerID and simple attributes name, street and city. 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 either SavingsAccount or a CheckingsAccount.

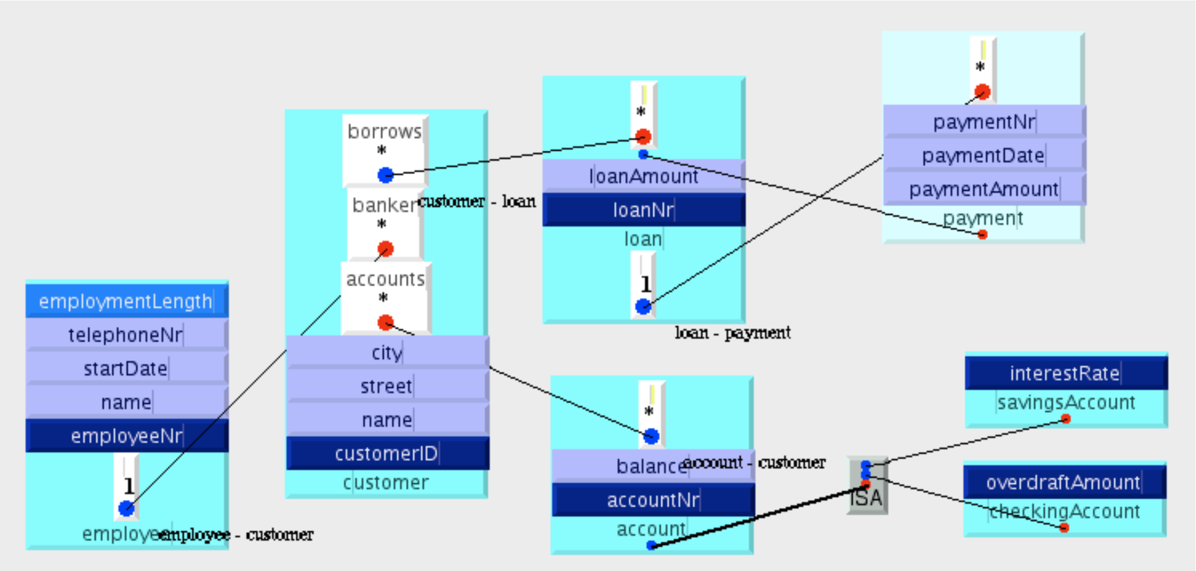


Figure 4.3: An EER diagram for a modest banking system.

The basic elements of the data modelling view are represented with the EER notation described above. In the next section we introduce the concrete implementation and population views of myARTE.

4.1.2 The Implementation and Population Views in Self

To represent the code-time implementation view and the run-time population view, we selected the prototype-based language Self. The different views are shown in figure 4.4. In chapter 3 we elaborated how the basic language constructs of Self can be combined for implementing language mechanisms that are required for supporting myARTE. In the remainder of this chapter we often use a specific terminology that refers to these basic concepts in Self or to concepts of prototype-based languages in general. For the convenience of the reader, a glossary of the most frequently

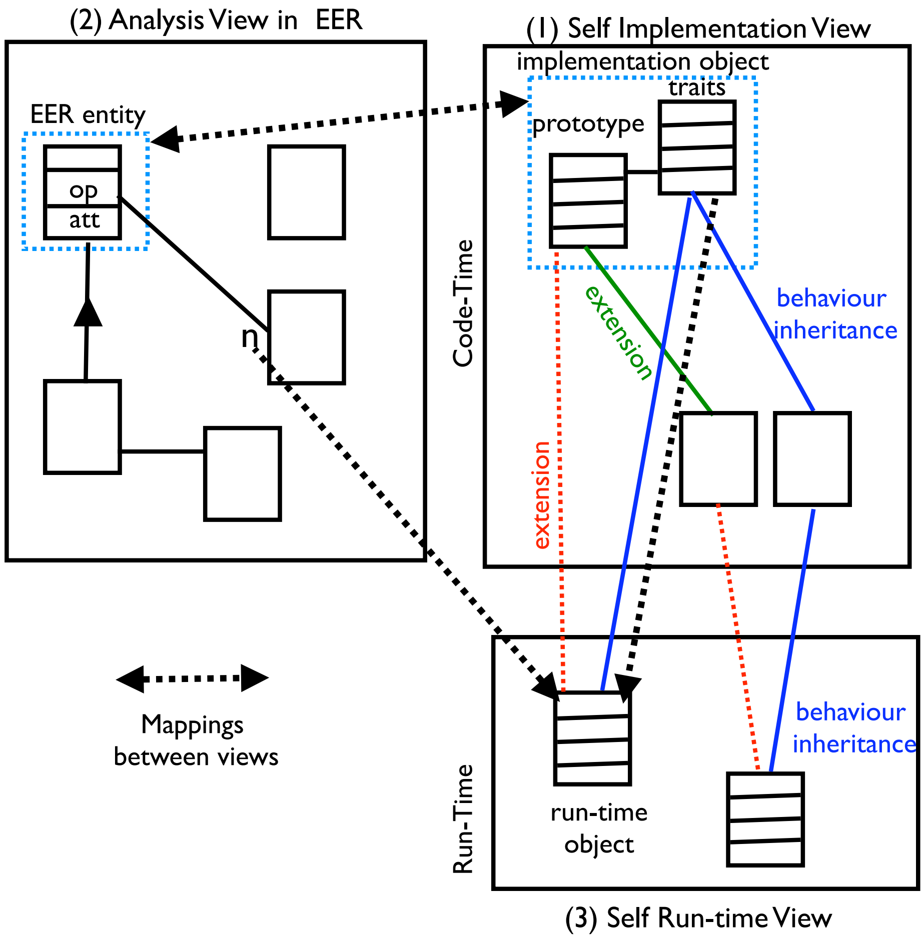


Figure 4.4: The EER view (2), a Self implementation view (1) and a Self run-time population view (3) in myARTE. The bold black arrows denote the direction of the mappings between the different views.

used terms can be found in Appendix B.

The elements of the implementation view and of the population view are both viewed via the user interface of the visual Self programming environment. This user interface is built with the Morphic framework [72] and applies a Model-View-Controller architecture [47]. This means that a real Self object can be viewed with a graphical representation called an *outliner*. An example outliner is illustrated in figure 4.5.

Self outliners are grey boxes consisting of multiple *slots* that can be copied and plugged in an outliner, or anywhere in the Self world. The slots include a slot name on the left side and the name of its contents on the right. The outer right symbol denotes the kind of slot: a semi-colon (:) and an equaly sign (=) denote, respectively, an *assignable data slot* and a *constant data slot*. Slots labeled

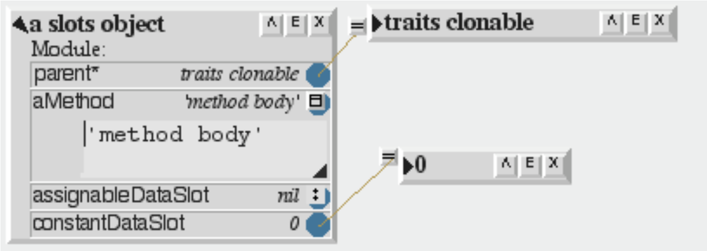


Figure 4.5: A Self outliner is a view on an underlying Self object.

with a rectangular symbol (□) are *method slots*.

Clicking on a data slot symbol in an outliner results in a new outliner on the contained object being opened and connected to the original outliner. Clicking a method slot symbol opens the body of the method to be edited. Slots are ordered alphabetically in an outliner and parent slots are sorted before plain slots. Slots can be gathered into *categories* that can be opened or closed to show or hide the contained slots. Finally a slot can be assigned to a specific module.

4.1.3 High-Level Architecture of a myARTE Tool for EER and Self

The high-level architecture for a myARTE tool that combines an EER view and a Self implementation and population view, consists of:

- A graphical drawing editor for EER diagrams, that supports the EER notation described in section 4.1.1.

- The Self environment it is built on top of.

Since the EER drawing editor is also developed with the help of the Morphic framework[3], the corresponding graphical entities can be considered a second view on Self objects. A Self object is viewed once by the original Self outliner we described in the previous section, and once by a graphical entity. This is illustrated in figure 4.6.

4.2 The Elements of the Different Views

Before we can describe mappings between the different views we need to define the *elements* or the *nodes* of the different views. First we informally describe the elements in the different views. Next, in order to be able to formally define implementation objects and run-time objects we define the

[3] This new combined EER notation was in fact merely a consequence of our choice of development platform Self, and more precisely of its user interface.

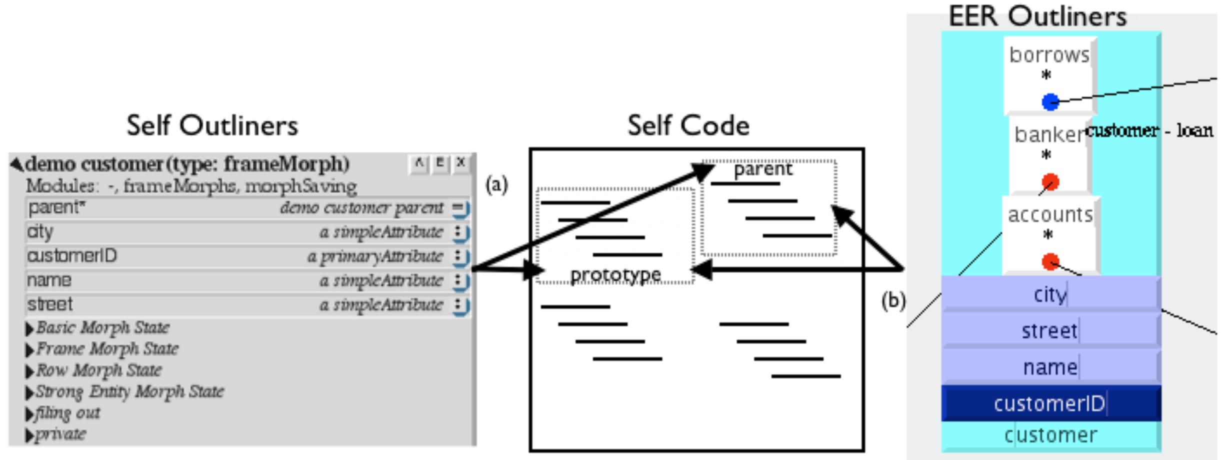


Figure 4.6: Self Outliners: (a) Real Self objects are viewed with a graphical outliner. (b) The EER outliners described in section 4.1.2 function as a second view on the same objects.

domain of Self objects, the message sending semantics in Self, and the extension relation in Self. Finally we formally define implementation objects and run-time objects.

4.2.1 Informal Description of the Elements

The first elements we consider are the elements in the data modelling view or analysis view. An *Entity* represents an object in the data modelling view with the EER notation introduced in section 4.1.1. Alternatively, in the remainder of this dissertation an entity can be called "entity view" or "entity outliner".

Second, we consider the elements in the implementation view. An *Implementation Object* in the implementation view is a *prototype-traits* pair that corresponds to a certain entity in the EER view. A prototype inherits from a traits object that gathers the behaviour to be shared by all objects that are created from this prototype. An implementation object is said to *inherit* from another when their prototypes are in a *state inheritance* relation and their traits objects are in a *behaviour inheritance* relation. One corresponding entity is then said to be a specialisation of the other one. State inheritance is realised via an *extension* mechanism, also referred to as *concatenation*. In Self this mechanism is also called *copy-down*. Behaviour inheritance in Self is implemented via *delegation* in the form of *parent slots* or *parent pointers*.

Third, we consider the elements of the run-time view. A *Run-time Object* is a *clone* of the prototype part of an implementation object. We say a run-time object is *derived from* an implementation object. Run-time objects are also referred to as *population objects*. Two run-time objects are said to be of the same *type* if they are both derived from the same implementation object.

An example of an implementation object and a derived run-time object can be found in figure 4.7. The implementation object has a prototype that includes two slots x and y. The parent slot x references the traits object that contains two method slots $add()$ and $print$. The run-time object has a prototype extends the prototype of the implementation object: it contains three slots: x, y and z. The run-time object also inherits from the traits object and has an additional parent object with method slots $draw()$ and $colour()$.

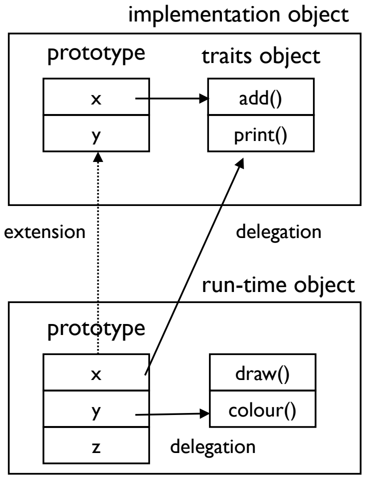


Figure 4.7: Example of an implementation object and a derived run-time object. An implementation object has exact one parent: its traits object. Both the run-time object and the implementation object share the same traits object via delegation. The run-time object inherits the state of the implementation object's prototype and extends it with a slot x. The run-time object in this example has a second parent.

Formally defining the entities in the analysis view is out of the scope of this dissertation. We focus on the definitions of the Self implementation objects and run-time objects. In order to be able to *formally* define implementation objects and run-time objects in Self and the inheritance relations between them, we first define the *domain of Self objects* that are considered in myARTE. Next, we define the semantics of Self's delegation *message sending mechanism* that is used in the context of behaviour inheritance. Finally, we define the semantics of the *extension relation* that is used for state inheritance in Self.

4.2.2 The Domain of Self Objects

The Self objects in myARTE consist of a *prototype*, a *traits* object and possibly more parent objects. As mentioned before, the prototype part of an object is intended to include the state that is specific for the object, while the traits gather behaviour that is shared via delegation inheritance by this object and all objects that are similar to the object. In order to express the nuance between prototypes and traits we consider a Self object as a combination of a *slot list* and a *parent list*[4]. The parent list allows to express the notion of multiple inheritance while the list of slots represents the prototype. In the example that is illustrated in figure 4.7, the run-time object's slot list contains the x, y and z slots and its parent list contains two elements: the traits object with the method slots $add()$ and $print$, and the other parent object with the method slots $draw()$ and $colour()$.

A slot is considered as a pair of an *identifier* and an associated *value*, e.g. $(a, 3)$ or $(double, (| : x| \; x * 2))$. The domain of slot lists $Slotlist$ contains all unique, possible combinations of pairs where the first element is an element of the domain of all identifiers $Identifier$ and the second element is an element of the domain of all values $Attribute$. An attribute can be either a value attribute or a method attribute.

$$Slotlist = (Identifier \times Attribute)^* \tag{4.1}$$

Some of these slots that are contained in Self object are *parent slots* that reference *parent objects*. These parent objects are in their turn slot lists. For formal convenience, we consider Self objects as a pair of a slot list in which we remove the parent slots, and a set of the parent objects that are referenced by the removed parent slots.

Therefore we define Self objects as a pair of a slot list and a list of parent objects. The domain of Self objects $Object$ includes all the pairs where the first element is a slot list and the second one an element of the power set of Self objects:

$$Object = Slotlist \times \mathcal{P}(Object) \tag{4.2}$$

In order to provide a deeper insight in the (multiple) inheritance mechanism of Self we elaborate on message sending and the semantics of method lookup.

4.2.3 Message Sending Semantics in Self

When sending a message m to a Self object O, the message is first *looked up* in O. More precisely a slot (m, x) is searched for in the slot list of O. If it is found there, the corresponding method x

[4] In Self, parents are also represented with slots whose name starts with an asterisk. Therefore, formally it makes sense to make a difference between "slot" and "parent".

is executed. If it is not found in the prototype part, the message is *delegated* to the parents of the receiver. More precisely, method lookup is recursively propagated to all the parents of the object O until the message is found or the root of the inheritance hierarchy is reached. In this last case the message m is not understood by O. The semantics of the method lookup function in Self can be specified as follows:

$$implements : (Identifier \times Object) \rightarrow \{false, true\} \tag{4.3}$$

$$implements(m, (sl, P)) = \begin{cases} true & \text{if } m \in sl \\ true & \text{if } \exists p_i \in P : \text{implements}(m, p_i) \\ false & \text{else} \end{cases} \tag{4.4}$$

Since Self supports multiple inheritance, it can occur that multiple parents of an object both include a method slot for the same message but with different behaviour. If two or more parents include different slots named m, the result of sending this *ambiguous* message is undefined.

Let $O = (sl, (O_1, O_2, ..., O_n)), sl \in Slotlist, O_i \in Object$. The semantics of message sending can be formalised as follows:

$$send : (Objects \times Identifiers \times Objects^*) \tag{4.5}$$

$$send(O, m, Arg^*) = \begin{cases} a(Arg^*) & \text{if } (m, a) \in sl \\ undefined & \text{if } \exists i \neq j : implements(m, O_i) \text{ and } implements(m, O_j), \\ send(O_i, m, Arg^*) & \text{if } \exists! i : implements(m, O_i) \\ false & else \end{cases} \tag{4.6}$$

Notice that these definitions get slightly more complex when the late binding of the `self` variable is taken into account. This is however not necessary in the context of our work. For more details on this matter we refer to [103].

4.2.4 Extension of Self Objects

A run-time object is a *clone*, i.e. a *shallow* copy of an implementation object. In section 4.2.5 we provide a more restricted definition of both run-time and implementation objects but for now we consider the slots of the run-time object to be a copy of the slots of the implementation object, that can be *extended* with new slots. The *extension relation* between a run-time object and an implementation object can be considered as a kind of inheritance [106]. We refer to this kind of inheritance as *state inheritance*. Inheritance in general can be formulated in terms of *record concatenation* [28]. In that context a *record* corresponds to our concept of slot list.

We use the binary record concatenation operator $+_c$ [28], [106], [17] for combining two slot lists into a new slot list. The function c is a conflict resolution function that specifies how two *conflicting* slots have to be combined. Two slots are *conflicting* if they contain the same identifier.

Definition 4.1 $\boxed{\textit{Conflicting Slots}}$

Given $s_1 = (i_1, a_1)$, $s_2 = (i_2, a_2) \in Slotlist$*. Then* s_1 *and* s_2 *are conflicting slots*

$$s_1 \natural s_2 \Leftrightarrow i_1 = i_2 \tag{4.7}$$

Examples of conflict resolution functions are left-preferential slot combination $+_L$ and right-preferential slot combination $+_R$. These operators respectively store the conflicting slots of the first slot list argument or of the second slot list argument in the resulting combined slot list. In Self, the meta-operation `_AddSlots:` is a right-preferential operator for combining two lists of slots while `_AddSlotsIfAbsent:` is a left-preferential operator for combining two lists of slots. For more details on conflict resolution functions we refer to [28].

The extension relation in Self between a run-time object and an implementation object can be formalized as follows:

$$+_c : (Slotlist \times Slotlist) \rightarrow Slotlist \tag{4.8}$$

$$c : ((Identifier \times Attribute) \times (Identifier \times Attribute)) \rightarrow (Identifier \times Attribute) \tag{4.9}$$

Given a slot list $sl = \{(i_j, a_j) | j \in \{1, n\}\}$ then

$$sl +_c \phi = sl_1 \tag{4.10}$$

$$sl +_c (i, a) = \begin{cases} (i_1, a_1), (i_2, a_2), ..., (i_{j-1}, a_{j-1}), c((i_j, a_j), (i, a)), (i_{j+1}, a_{j+1}), ..., (i_n, a_n)) \\ \text{if } \exists (i_j, a_j) \in sl : (i_j, a_j) \natural (i, a) \\ \\ (i_1, a_1), (i_2, a_2), ..., (i_n, a_n), (i, a) \\ \text{else} \end{cases} \tag{4.11}$$

$$sl +_c ((i, a), sl') = (sl +_c (i, a)) +_c sl'. \tag{4.12}$$

We are now able to define implementation objects and run-time objects.

4.2.5 Definition of Implementation Object and Run-time Object

In myARTE an implementation object corresponds to a prototype-traits pair. Such an implementation object does not have multiple parents but inherits all behaviour via its own traits object. Based on the previous definitions, we can define an implementation object as follows:

Definition 4.2 $\boxed{\textit{\textbf{Implementation Object}}}$

An implementation object O is a pair of a slot list and a parent list that contains one element:

$$O = (sl, P) \text{ with } sl \in Slotlist, P \in \mathcal{P}(Object) : |P| = 1 \tag{4.13}$$

The sole element of $P = \{p\}$ is referred to as the traits *of the implementation object.*

A run-time object is a clone of the prototype part of an implementation object. A run-time object cán extend its prototype part with additional slots. At all times a run-time object has to inherit from the traits part of the implementation object it is derived from. However, the run-time object cán inherit from more parents. Based on the previous definitions, a run-time object can be defined as follows:

Definition 4.3 $\boxed{\textit{\textbf{Run-time Object}}}$

A run-time object R derived from an implementation object $O = (sl, \{p\})$ is defined as

$$O \rightsquigarrow R = (clone(sl) +_c sl', \{p\} \cup P) \tag{4.14}$$

with $sl' \in Slotlist$, $P \in \mathcal{P}(Object)$ and c a conflict resolution function.

4.3 Mappings Between the EER View and the Implementation View

Partially due to the visual programming environment of Self there exists in general a reasonably straightforward mapping between the concepts of the data modelling view and those of the code-time implementation view in myARTE. Although this is not a one-to-one mapping, we define a set of default mappings (sometimes optimisations are mentioned) between the modelling elements of the EER view and the Self implementation objects that are represented in the implementation view. The default mappings are illustrated in figure 4.4 by the bidirectional, bold arrows and are presented in table 4.1.

In this section we concisely describe the mappings between the EER view and the implementation view. In the next section we elaborate the mapping between roles in the EER view and a corresponding setup in the implementation view. We summarize the mappings as follows:

EER View	**Self Code-Time Implementation View**
Entity	Prototype and traits object
Attribute (all)	Assignable data slot in prototype
Specialisation	Extension between prototypes + Delegation between parent objects
Relationship	Bidirectional slot reference between prototypes
Dependency	Creation dependency reference + Deletion propagation at run-time
Categorisation	Extension between prototypes + Delegation between parent objects
Aggregation	New slot reference in aggregate to new object containing aggregates
Roles	Warped Inheritance Hierarchies

Table 4.1: Mappings between the EER data modelling view and the Self implementation view. When changes such as creation, deletion, and changes of contents occur at the level of the modelling elements of the left column, these are synchronised in the corresponding implementation elements of the right column and vice versa.

- An entity in the EER view is in fact a view on an underlying new Self object, as illustrated in figure 4.6. In the implementation view this entity corresponds to an implementation object (prototype-traits pair) represented by two Self outliners. When it is a weak entity, the implementation object corresponding to the strong entity it depends on, ensures that when population objects created from the strong entity are deleted, its dependent population objects are deleted along, as elaborated in section 2.5.3.

- An attribute in an entity in the EER view maps to an assignable data slot in the prototype part of the corresponding implementation object in the implementation view and vice versa. The contents of attributes are to be determined at run-time in the population objects.

- A specialisation link drawn between two entities in the EER view maps to inheritance between the two corresponding implementation objects and vice versa. As mentioned before, inheritance between two implementation objects involves state inheritance between their prototypes and a behaviour inheritance between their traits objects. This inheritance setup is illustrated in figure 4.8.

- Relationships in the EER view between two entities do not *directly* affect the corresponding

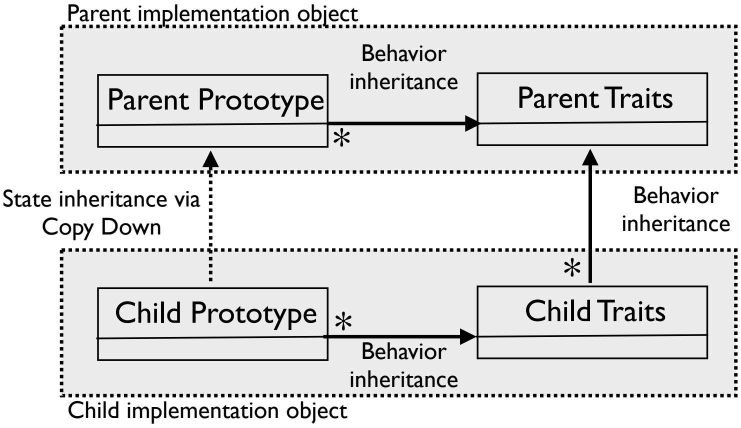


Figure 4.8: Inheritance between two Self implementation objects: extension between prototypes and delegation between traits objects.

implementation objects. Adding a relationship between two entities in the EER view thus results in adding private information in the corresponding implementation objects. More precisely we map such relationships to a separate slot in both participating implementation objects such that the slot in the one implementation object contains a reference the other implementation object. The myARTE cloning process (that is responsible for creating new population objects, elaborated in section 4.7.2.) uses these slots to ensure that the necessary references between derived run-time objects are installed. Therefore the population objects derived from these implementation objects contain also a slot that directly or indirectly (through a collection for example) contains zero, one or more occurrences (dependent on the multiplicities) of run-time objects of the other type and vice versa. This is a rather naive implementation for associations that can easily be optimized through analysis patterns [84].

In the other direction adding a slot in an implementation object, that contains a reference to another implementation object is mapped to drawing a relationship between the two corresponding entities in the EER view.

- Similarly to relationships, a dependency relation between two entities in the EER view does not directly affect the corresponding implementation objects but is mapped to private information in the involved implementation objects. Also similarly, the myARTE cloning process for creating new run-time objects will use this information to enforce object dependency in the run-time objects that are derived from the corresponding dependent implementation objects. This means that when a new dependent population object is created it has to contain at least one reference to a population object it can depend on. In the other direction when a population object is deleted all dependent population objects are deleted as well.

In the other direction of the mapping, adding, changing or removing the private dependency information from an implementation object is mapped to adding, changing or removing the dependency relation in the EER view.

- Categorisations between entities in the EER view maps to private information in the involved implementation objects. Again the myARTE cloning process for creating new run-time objects will use this information to determine the parents of the run-time objects that are derived from the involved implementation objects at creation time.

  In the other direction, adding, changing or removing the private categorisation information from an implementation object is mapped to adding, changing or removing the categorisation relation in the EER view.

- Aggregation in the EER view between an aggregate entity (the whole) and its aggregated entities (the parts) is mapped to private aggregate information in the aggregate implementation object. The myARTE cloning process for creating new run-time objects will use this information to create aggregate run-time objects that include a slot that contains a collection of the aggregated run-time objects.

  In the other direction, adding, changing or removing the private aggregation information from an implementation object is mapped to adding, changing or removing the aggregation relation in the EER view.

- Finally, a role relation between a natural entity and a role entity in the EER view maps to a combination of three things. First, to a slot in the corresponding *natural implementation object*, that contains information on what roles the run-time objects derived from this natural implementation object can play. Second, to a slot in the corresponding natural implementation object, that contains information on which of the roles that the run-time objects derived from this natural implementation object can play are mutually exclusive. Third, to inheritance between the natural implementation object's traits and a traits object that contains general behaviour for playing roles.

  The private information in the two slots is used by the myARTE cloning process for creating new natural run-time objects that play roles that are not mutually exclusive. During their life time these run-time objects can start play roles or abandon roles. This behaviour is defined in the inherited traits object that contains general behaviour for playing roles and also uses the private information in the implementation object to enforce the constraints imposed by mutually exclusive roles.

In the other direction, adding or removing information on a new role to the slot in an implementation object, that contains information on which roles can be played by this object, is mapped to adding or removing a new role relation to this role in the EER view. When the traits object that contains general behaviour for playing roles is removed from a natural implementation object's inheritance hierarchy, all the role relations of the corresponding natural entity in the EER view are removed.

In the following section we elaborate on this non-trivial mapping.

Details of a possible implementation of the above mappings can be found in chapter 5.

4.4 Mapping EER Roles to an Implementation

In this section we elaborate on how myARTE supports role modelling. We start with defining the concept of warped inheitance hierarchies followed by definitions for the implementation objects and run-time objects that correspond to the natural entities and role entities in the EER view. Next we formally define the concept a one run-time object playing a role, based on warped inheitance hierarchies. We end this section with a discussion of the context and the repercussions of this implementation technique.

4.4.1 Definition of Warped Inheritance Hierarchies

In order to implement role modelling in myARTE we apply an implementation technique based on *warped inheritance hierarchies*. Two warped inheritance hierarchies contain two run-time objects and exhibit the opposite order. One of the warped inheritance hierarchies contains the two prototypes of two run-time objects and the other inheritance hierarchy contains the two traits objects of the same run-time objects. Inheritance between prototypes or *state inheritance* is realised via extension. Inheritance between traits objects or *behaviour inheritance* is realised by adding a parent slot in the child traits, that references the traits of the ancestor.

In figure 4.9 (right) we present a simple case of warped inheritance hierarchies. In figure 4.9 (left) the state inheritance hierarchy between the prototypes, and the behaviour inheritance hierarchy between the traits follow the same order. In figure 4.9 (right) the state inheritance hierarchy between the prototypes, and the behaviour inheritance hierarchy follow the opposite order.

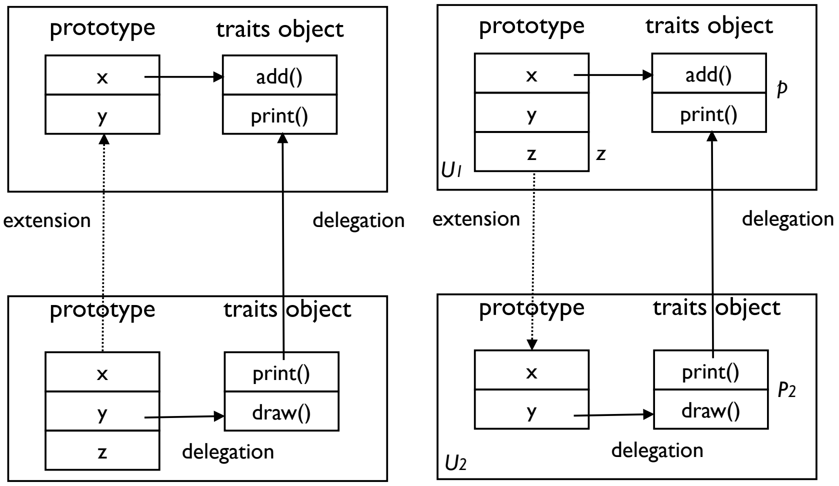


Figure 4.9: (Left) Normal inheritance hierarchies. (Right) Warped inheritance hierarchies.

Based on the definitions we provided in section 4.2.5 we define warped inheritance hierarchies as follows:

Definition 4.4 ***Warped Inheritance Hierarchies***

Consider two run-time objects U_1 and U_2 such that

1) $I_1 = (sl_1, p_1) \rightsquigarrow U_1 = (sl, \{p_1\} \cup P_1)$ *with* $sl = clone(sl_1) +_c x_1$

2) $I_2 = (sl_2, p_2') \rightsquigarrow U_2 = (sl', \{p_2\} \cup P_2)$ *with* $sl' = clone(sl_2) +_c x_2, andp_2 = (s, P_{p_2})$

There exists warped inheritance hierarchies between U_1 and U_2

$$U_1 \downarrow\uparrow U_2 \Leftrightarrow \exists z : sl = clone(sl') +_c z \text{ and } p_1 \in P_{p_2} \tag{4.15}$$

In this definition, the order of the behaviour inheritance goes from U_2 to U_1. This is formulated by requiring that the traits object of the implementation object from which U_1 is derived p_1 is included in the parent list of the traits object of the implementation object from which U_2 is derived, i.e. P_{p_2}. In the example in figure 4.9 (right) the traits object with $add()$ and $print$ represents $\{p_1\}$ and in this case, $\{p_1\} = P_{p_2}$.

The order of the state inheritance goes from U_1 to U_2. In other words, the slot list of U_1 is an extension of the slot list of U_2. This is formulated by requiring that $\exists z : sl = clone(sl') +_c z$. In the example in figure 4.9 (right) the prototype with x, y and z is an extension of the prototype with x and y. In this case the required z of the definition is the slot z.

4.4.2 Definition of Natural Objects and Role Objects

In our viewpoint on roles, that is elaborated in section 2.5.1, roles can be seen as instances that are to be adjoined to the objects that play them. In the same section we explain the notion of natural entities and role enties, and the corresponding natural run-time objects and run-time roles.

Consider a natural entity E_n and one of its role entities E_r in the analysis view. As explained in section 2.5.1 and in the mappings between the EER view and the implementation view, in myARTE both entities correspond to a separate implementation object in the implementation view. These corresponding implementation objects N and R are in a role relation.

Definition 4.5 **_Role Relation_**

Let map be a myARTE mapping between the analysis view and the implementation view. Two implementation objects N and R are in a role relation

$$Role(N, R) \Leftrightarrow \exists \textit{ natural entity } E_n \textit{ and its role entity } E_r : map(E_n) = N, map(E_r) = R \tag{4.16}$$

A *natural implementation object* is an implementation object that participates in a role relationship and that corresponds to a natural entity.

Definition 4.6 **_Natural Implementation Object_**

Given an implementation object $N = (sl, \{p\})$ and an implementation object R such that $Role(N, R)$, then N is called a natural implementation object for R.

A *role implementation object* is an implementation object that participates in a role relationship and that corresponds to a role entity. The behaviour of a role implementation object is considered to be more specialised than the behaviour of the natural implementation object. For example, consider a role relation between a person entity and a manager entity. The role implementation object that corresponds to the manager entity is likely to contain methods that override the methods in the natural implementation object that corresponds to the person entity. As a consequence we require that a role implementation object contains the traits of its natural implementation object in the parent list of its own traits. In the previous example this implies that the traits of manager inherits directly or indirectly from the traits of person.

Definition 4.7 ***Role Implementation Object***

Given an implementation object $N = (sl, \{p\})$ and an implementation object R such that $Role(N, R)$, then R is a role implementation object for N that is of the form

$$R = (sl_R, \{p_R\}) \textit{ with } p_R = (x, P_{p_R}), p \in P_{p_R} \tag{4.17}$$

Adding roles as instances to the objects is realised at run-time. We derive natural run-time objects from natural implementation objects.

Definition 4.8 ***Natural Run-time Object***

A run-time object U is a natural run-time object for R

$$R \rightarrow_{NAT} U \Leftrightarrow \exists N : Role(N, R), N \rightsquigarrow U \tag{4.18}$$

For example, a run-time person object that is derived from a natural implementation object person for the role implementation object manager, is a natural run-time object for the role implementation object manager.

4.4.3 Implementing Roles with Warped Inheritance Hierachies

When a natural run-time object starts playing a role we realise warped inheritance hierarchies between the natural run-time object and a run-time role.

More precisely, a new run-time object is derived from the role implementation object, which is immediately concatenated to the slot list of the natural run-time object: the *state*, i.e. the slots of the role implementation object is *copied* into the slot list of the natural run-time object. The run-time role object does not exists independently from the natural run-time object.

In figure 4.10 the setup of a natural run-time object $person$ that plays the run-time $role$ manager is illustrated. The slots of the run-time role' s prototype, i.e. $salary$, $office$ and $managerP$ are concatenated in the run-time person's prototype that originally contained a slot $name$.

The corresponding natural implementation object $person$ and the role implementation object $managar$ are illustrated in figure 4.11. In figure 4.12 a natural run-time object $person$ is depicted,

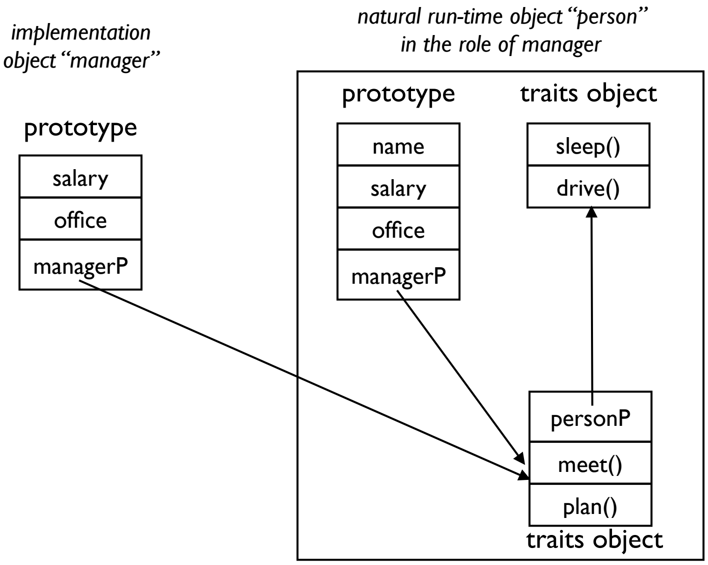
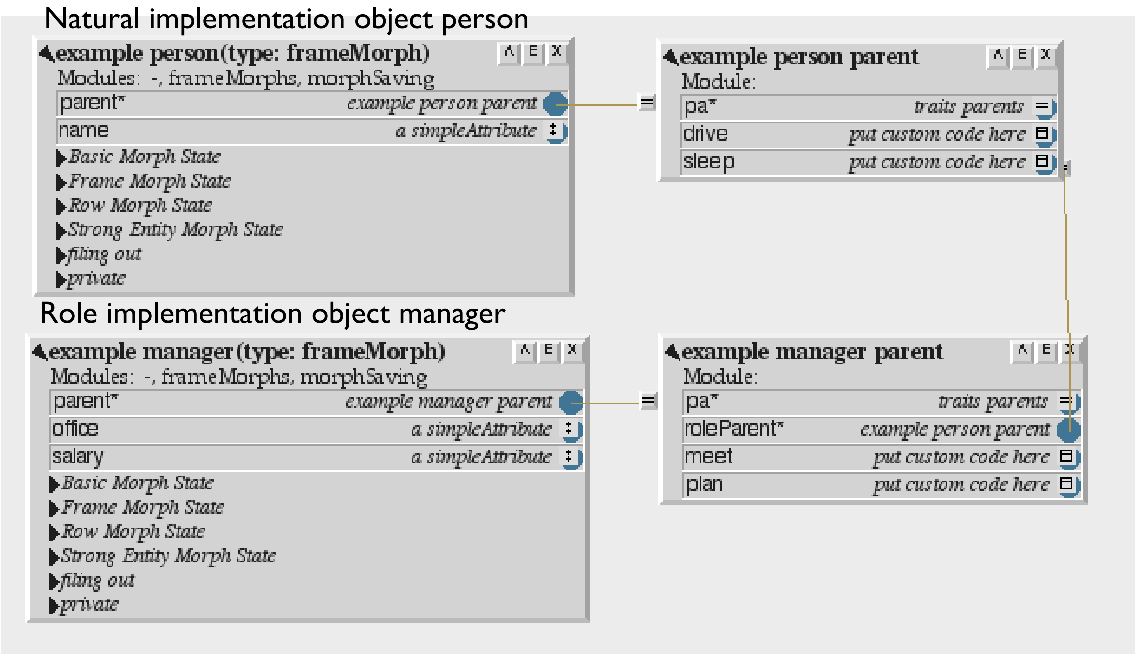


Figure 4.10: A natural run-time object person in the role of manager.


Natural implementation object person
Role implementation object manager

example person(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* example person parent
name a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

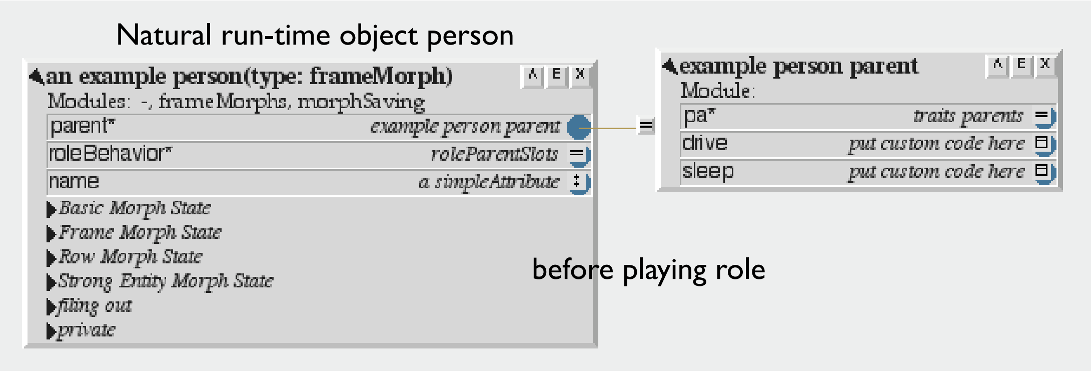
example person parent
Module:
pa* traits parents
drive put custom code here
sleep put custom code here

example manager(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* example manager parent
office a simpleAttribute
salary a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

example manager parent
Module:
pa* traits parents
roleParent* example person parent
meet put custom code here
plan put custom code here

Figure 4.11: A natural implementation object person and a role implementation object manager. The traits of manager inherit from the traits of person. Between the prototypes of person and manager there is currently no inheritance relationship.

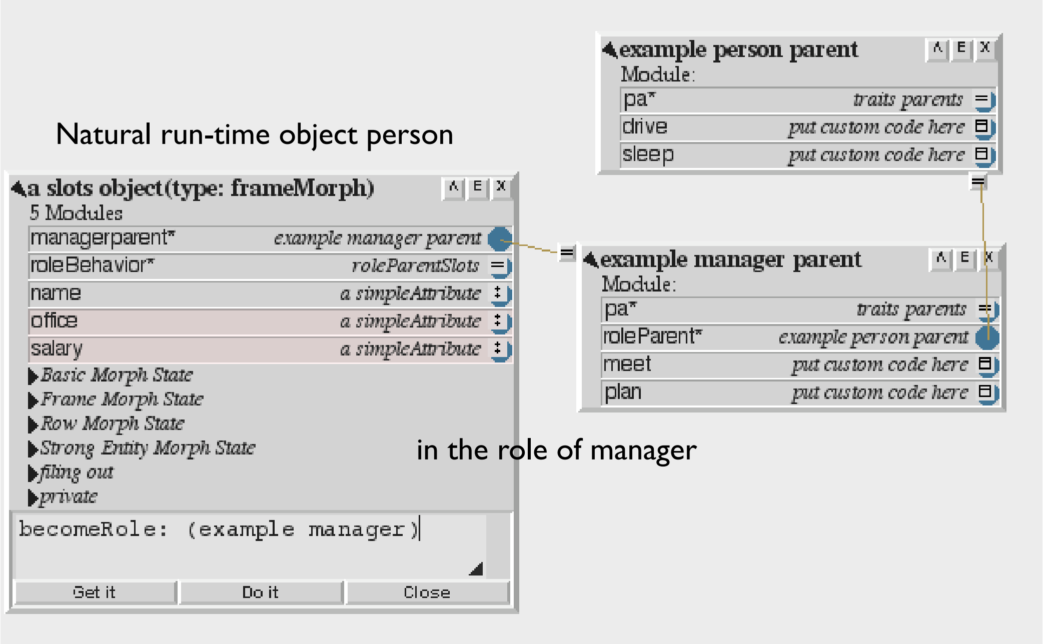
that currently plays no roles. Figure 4.13 illustrates a natural run-time object $person$ in the role of $manager$.



an example person(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* *example person parent*
roleBehavior* *roleParentSlots*
name *a simpleAttribute*
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

example person parent
Module:
pa* *traits parents*
drive *put custom code here*
sleep *put custom code here*

Figure 4.12: A natural run-time object *person*. Currently this object plays no roles and inherits directly from its own traits object.



example person parent
Module:
pa* *traits parents*
drive *put custom code here*
sleep *put custom code here*

a slots object(type: frameMorph)
5 Modules
managerparent* *example manager parent*
roleBehavior* *roleParentSlots*
name *a simpleAttribute*
office *a simpleAttribute*
salary *a simpleAttribute*
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

```
becomeRole: (example manager)
```

example manager parent
Module:
pa* *traits parents*
roleParent* *example person parent*
meet *put custom code here*
plan *put custom code here*

Figure 4.13: A natural run-time object *person* in the role of *manager*. The slots of a run-time role *manager* are concatenated to the slot list of the natural run-time object *person*. This object inherits of the traits of *manager* that inherits in its turn from the traits of *person*.

If the natural run-time object performs no roles yet, it contains a parent slot that references its own traits object. For example a run-time person object contains a parent slot to the traits object of person. Upon starting to play the role, one of the added slots in the natural run-time object is a parent slot that references the traits object of the role implementation object. As explained above a role implementation object contains the traits object of its natural implementation object in the parent list of its own traits object. Therefore we remove the natural run-time object's traits object from the natural run-time object. In this way we avoid that for example a run-time person that plays the role of manager, inherits from its own traits object twice: once directly and once via the parent list of the traits object of manager. Otherwise, based on the method lookup semantics we provided above, sending a message to the run-time person that corresponds to a method in the traits object of person that is overridden in the traits object of manager causes an undefined result.

If the natural run-time object plays at least one role, the slot list of the natural run-time object is simply extended with a copy of the slot list of the role implementation object.

Definition 4.9 $\boxed{\textit{In the Role of}}$

Given a run-time object U *and an implementation object* R *such that*

1) $(sl, \{p\}) \rightsquigarrow U = (clone(sl) +_c sl', \{p\} \cup P)$

2) $R = (sl_R, \{p_R\})$ *with* $p_R = (x, P_{p_R})$ *and* $p \in P_{p_R}$

3) $R \rightarrow_{NAT} U$

We define "U in the role of R" as the run-time object defined by

$$U \lhd R = (clone(sl) +_c sl' +_c clone(sl_R), P \cup \{p_R\}) \tag{4.19}$$

On the one hand, since a run-time object in a certain role is an extension of a run-time role and extension is a kind of inheritance [106], *a run-time object in a role inherits from that role*. For example, the slotlist of a natural run-time object person in the role of manager is extended with a copy of the slot list of the manager implementation object. This copy of the slot list of the manager implementation object satisfies the definition of a run-time object that is derived from the manager implementation object, i.e. a run-time manager object. Therefore the run-time person object in the role of manager inherits from a run-time manager object.

On the other hand, we already mentioned that the behaviour of a role implementation object is more specialised than the behaviour of the natural implementation object. This is reflected in the fact that a role implementation object contains the traits of its natural implementation object in the parent list of its own traits.

This means that in a natural run-time object in a certain role, the order of inheritance between the slot lists of the natural run-time object and the run-time role is opposite to the order of inheritance between the traits of the run-time role and the traits of the natural run-time object. In this way we realise warped inheritance hierarchies between a natural run-time object and each run-time role it plays.

Since in our perspective on roles, natural run-time objects can abandon a role dynamically, we implement this by removing the state inheritance between the natural run-time object and the run-time role that is abandoned.

$$(U \lhd R) \downarrow\uparrow R \tag{4.20}$$

What actually happens when a natural run-time object stops playing a role is that the slots of the role implementation object that were copied into the slot list of the natural run-time object upon starting to play the role, are removed again. When a natural run-time object stops performing its sole role and the appropiate slots are removed from its slot list, a parent slot that references the original traits of this natural run-time object is added again.

Definition 4.10 ***Out of the Role of***
Given a natural run-time object O such that $O = U \lhd R$. We define "O out of the role of R" as U, denoted

$$O \rhd R = U \tag{4.21}$$

4.4.4 Discussion

In this section we discuss the context and the repercussions of using the above implementation technique for roles.

Repercussions of $+_c$ and $send$ on role modelling semantics

The state inheritance that we use between natural run-time object and its run-time role currently applies a slot list combination operator $+_c$. This conflict resolution function c and the message sending function $send()$ are crucial for certain parts of the semantics of our role modelling concept on which we elaborated in section 2.5.1:

4. *An object may play the same role several times, simultaneously.* The slots of a run-time role are added to the natural run-time object with $+_c$ that is also used for deriving run-time objects from implementation objects. If c is for example left-preferential then the slots of the natural run-time object are prioritized compared to the slots of the run-time role, that are in their turn prioritized compared to the slots of future run-time roles of the natural run-time object. This implies that a natural run-time object that plays the same role several times will only include one slotlist, more precisely the slotlist of the first role, for all the corresponding run-time roles.

More precisely the natural run-time object exhibits the same state for each of these role. An analoguous situation occurs, if c is right-preferential. In that case, the slot list of the last role that is played is included in the natural run-time object.

This means that for satisfying the requirement that an object can play the same role several times, simultaneously, it might be necessary to define a new conflict resolution function d for an operator $+_d$ that can be used for the warped inheritance hierarchies that are realised when a natural run-time objects starts playing a role. Examples for d could be based on a sophisticated aliasing mechanism for homonymous slots or on a mechanism that allows homonymous slots to be stored in different Self categories.

11. *Features of an object can be role-specific: attributes and behaviour of an object may be overloaded on a by-role basis.* Overloaded features correspond to homonymous slots in the natural run-time object that plays several (different) roles simultaneously and thus, as explained in the first bullet, depend on the conflict resolution function d for the operator $+_d$ that is used for the warped inheritance hierarchies that are realised when a natural run-time objects starts playing a role.

    Overloaded behaviour in different roles involves homonymous method slots in the traits objects of the run-time roles. The natural run-time object that plays the roles inherits from all these traits objects. Based on the current semantics of our message sending function $send()$ we provided above, sending an overloaded, ambiguous message to the natural run-time object has an undefined result. This means that if the overloaded behaviour part of the requirement has to be satisfied, we need to extend the semantics of the $send()$ function. This is an all but trivial task since there exists no general way for composing two overloaded methods. Default examples of combination operations for two overloaded methods are left-preferential and right-preferential combination and composition of the two methods. However, in the context of roles, the required combination can be quite complex and role-specific. For example, if we need to compute the minimum salary of a natural run-time object with multiple roles, we have to compute the salary of all the run-time roles, by calling all the $salary()$ methods of the different roles sequentially and identify the minimun result.

Related Approaches

One of the most related approaches for supporting the viewpoint of roles as instances to adjoined to the objects that play them, is the Object Specialisation technique [94] where each object is represented as a prototype-based inheritance hierarchy of instances. Each instance represents a role of the root object and inherits all the properties of its ancestors. If a message is sent to an instance and not found, the message is delegated to the parent instance of the instance. One of the main differences with our approach is that the run-time role, i.e. the instances in the hierarchy are independent objects with an identity. This implies that a natural run-time object and the its roles have separate identities. In our appraoch the run-time roles have no separate identity.

Using warped inheritance hierarchies between a natural run-time object and its run-time role involves 1) a state inheritance hierarchy where the natural run-time object in a role is a child of the run-time role and 2) a behaviour inheritance hierarchy where the run-time role is a child of the natural run-time object. In this way our approach acknowledges the paradoxical viewpoint that roles can be seen as both subtypes and supertypes of the natural objects that perform them, that was introduced in section 2.5.1. There exist various approaches that handle the paradoxical situation that roles are both super- and subtypes. We summarize the four techniques that are most relevant to our approach.

The category concept [34] concept is defined as the subset of the union of a number of roles (types). As in myARTE the Entity-Relationship diagram was extended: relationships are not defined on entity types, but on categories.

In [13] roles are considered temporal specializations: statically, a manager is a specialization of a person. However, when a particular person object becomes a manager, its type is changed from person to the subtype employee thereby inheriting all aspects of its new role. In this way reversed specializations that are similar to warped inheritance hierarchies, are realized temporarily.

[99] also separate between static and dynamic type hierarchies: state sharing, behaviour sharing, as in Self, and subset hierarchies are combined into a new specialization modelling concept.

In [59] delegation is used to implement dynamic roles that "import" state and behaviour from their parent objects.

The role modelling concepts in the approaches described above provide suitable alternatives for warped inheritance hierarchies. However, to the best of our knowledge, none of them is integrated in an object-oriented modelling environment that supports automatic synchronization between the modelled roles and a corresponding implementation.

For an in-depth discussion and more related approaches we refer to [101].

4.5 Mappings From the Implementation View to the Run-time View

The run-time view consists of run-time population objects that are created by "instantiating" the code-time implementation objects (cfr. figure 4.4 part (3)). A run-time object is a clone of the prototype part of the implementation object and shares the same parent objects through parent sharing via parent pointers. The mappings from implementation objects to run-time objects can be summarized as follows:

- Since population objects are clones or *shallow* copies of the prototype, changes in the prototypes in the implementation view are not visible for population objects. In other words adding a new method slot, a data slot or a parent slot in the prototype never affects the existing clones.

  In the evolution scenarios 5 and 6 of section 2.3 we stated that by default changes to population objects are not propagated to the other views.

- Clones include the parent pointers of the prototype they are created from: population objects share the same parent objects (traits) as the prototypes. Therefore changes to parent objects in the implementation view do influence population objects. This means that adding a new method slot, a data slot or a parent slot in the parent objects always affects the existing clones.

The creation of population objects happens in the Interactive Prototyping phase described in section 4.7.2.

4.6 Mappings from the EER to the Run-time View

One of the contributions of our RTE approach lies in the fact that changes to the EER view affect **already existing** population objects in certain cases. These mappings always happen via the implementation view: changes occur in the EER view, the implementation view is automatically synchronised and this affects the run-time population objects.

The mappings between the EER view and the population view are represented in table 4.2 and can be summarized as follows:

EER View	Self Run-Time Population View
Entity	-
Attribute (all)	-
Specialisation	Behaviour inheritance between traits objects
Relationship	Constrain number of references based on multiplicities
Categorisation	-
Aggregation	-
Roles	-

Table 4.2: Mappings from the EER data modelling view to the Self population view. Changes to the modelling concepts of the EER view in the left column influence the population view elements in the right column where appropriate. However, changes to the implementation elements or references in the population view in the right column by default do not affect the EER view concepts in the left column, as explained in section 2.5.2 and in scenario 5 of section 2.3.

- Whether changes such as adding attributes and other changes to the EER view, such as the state part for specialisation and categorisation, that are synchronised in the prototypes in the implementation view, are visible to already existing population objects is to be considered.

  On the one hand, it intuitively is expected that these changes in an entity view are propagated to the already existing run-time objects. Such a model change involves the structure of an entity of the EER diagram, so from that point on all appropriate run-time objects have to implement this new structure, including the existing ones.

  On the other hand, the cloned run-time population objects are shallow copies, as explained in section 4.5. In order to respect this prototype-based style, the above changes should not be propagated to existing run-time population objects.

  As both solution are acceptable, we deliberately postpone the decision until the actual implementation of a myARTE tool.

- Changes at the level of specialisations in the EER view result in adding or removing state inheritance between the corresponding prototypes, and in adding or removing behaviour inheritance between the corresponding traits objects in the implementation view. The latter causes the already existing run-time objects to inherit the corresponding traits object in the implementation view. As mentioned before, the changes in the inheriting child prototype are not visible to the already existing run-time objects, due to shallow copying.

- Relationships in the EER view constrain the run-time objects via their multiplicities. This affects the number of allowed references between the corresponding run-time objects, as explained in section 2.5.3.

- Roles in the EER view do not affect run-time objects directly. However, it needs mentioning that adding a new role relation between a natural entity and a role entity in the EER view results in the fact that the already existing natural run-time objects now can start playing a new kind of role.

4.7 A New Agile MDD Practice for myARTE

In this section we propose a new approach to Agile Model-Driven Development (AMDD) [5] (see section 2.1), that can be supported by myARTE tools. In this case the elements of the abstract data modelling view in EER, i.e. an EER diagram, are considered as a platform-independent model (PIM), while the contents of the implementation view in Self composes a platform-specific model (PSM).

Due to an automated transformation between the aforementioned PIM and PSM, our approach can be seen as a practice of what is referred to as "Agile MDA" [5] (see section 2.1.2), as illustrated in figure 4.14.

The innovative aspect of our approach is the use of myARTE tools instead of CASE tools. Moreover, run-time objects are synchronised and constrained by the PIM and the PSM. Since we use EER (instead of the UML) for the PIM, the distance between Phase 0 and Phase 1(a) is minimal.

The iterative AMDD life cycle we consider in this case, consists of three main phases that map to the original six phases introduced in [5]:

- **Phase 0** or *Initial Modelling* includes creating an initial simple, inclusive static data model together with the end-user. This activity is performed in the first iteration.

- **Phase 1** or *Active Modelling* includes in **Phase 1(a)** creating or changing an EER diagram (PIM) (*Model Storming* in [5]). In **Phase 1(b)** the evolution to this EER diagram is automatically synchronised in a corresponding Self implementation (PSM) with a myARTE tool (*Implementation* in [5]).

  In **Phase 1(c)** evolution in the Self implementation is automatically synchronised in the corresponding EER diagram with a myARTE tool.

- **Phase 2** or *Interactive Prototyping*[5] includes deriving a working implementation from the PSM that was developed so far, with the help of an interactive process initiated by a myARTE tool.

[5] The activity of instantiating and initializing a program into a ready-to-use, running system as in rapid prototyping.

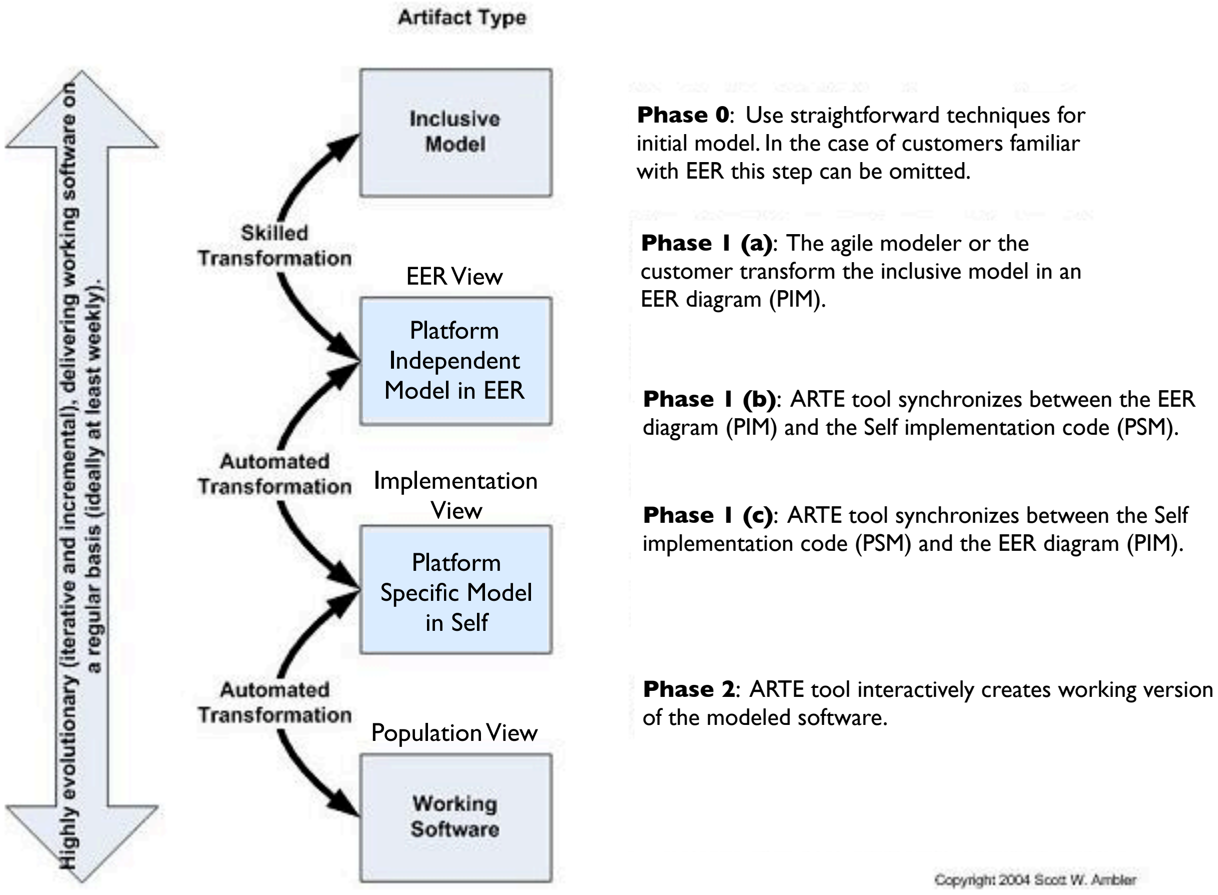

Artifact Type
Highly evolutionary (iterative and incremental), delivering working software on a regular basis (ideally at least weekly).
Inclusive Model
Skilled Transformation
EER View
Platform Independent Model in EER
Automated Transformation
Implementation View
Platform Specific Model in Self
Automated Transformation
Population View
Working Software
Phase 0: Use straightforward techniques for initial model. In the case of customers familiar with EER this step can be omitted.
Phase I (a): The agile modeler or the customer transform the inclusive model in an EER diagram (PIM).
Phase I (b): ARTE tool synchronizes between the EER diagram (PIM) and the Self implementation code (PSM).
Phase I (c): ARTE tool synchronizes between the Self implementation code (PSM) and the EER diagram (PIM).
Phase 2: ARTE tool interactively creates working version of the modeled software.
Copyright 2004 Scott W. Ambler

Figure 4.14: An Agile MDD methodology supported by myARTE tools.

More precisely, creating new run-time population objects is the main purpose of this phase. This phase enables our new practice to integrate run-time objects in a highly dynamic RTE process.

Phase 1 and 2 are typically but not necessarily executed consecutively and occur during any iteration, including the first.

In the following sections we elaborate the Active Modelling and Interactive Prototyping phases.

4.7.1 Phase 1: Active Modelling

In this phase an agile modeler, an end-user or another participant of the myARTE process initiates or makes a change to an EER diagram in the EER view (Phase 1(a)), which is immediately and automatically synchronised in the implementation view (Phase 1(b)) based on the mappings in table 4.1.

Conversely, when the implementation objects in the implementation view are manipulated (Phase 1(c)) the corresponding entities in the EER view are automatically synchronised and population objects are affected where applicable.

Table 4.3 illustrates the presence of the myARTE characteristics introduced in section 2.4, in the Active Modelling phase.

	Active Modelling
Optimal Transformations	$\surd$
Run-time Object Generations	$\surd$
Multiplicity Constraint Enforcement	$\surd$
Object Dependency	ϕ
Role Modelling	$\surd$

Table 4.3: Presence of myARTE characteristics in the Active Modelling phase.

Phases 1(a) en 1(b) coincide due to **Optimal Transformations**, as introduced in section 2.5.2: the synchronisation between the EER view and the implementation view is realised continuously and incrementally and happens *per entity* in the EER data modelling view or *per object* in the implementation view. Furthermore, the synchronisation demands a minimal transformation: e.g. a change to an attribute in the EER view is only synchronised at the level of the corresponding data slot in the corresponding implementation object and vice versa. View-dependent information, such as relationship constraints in the EER diagram and method bodies in the Self objects, is preserved during changes and subsequent synchronization.

Certain changes in the EER view or the implementation view during Phases 1(a) en 1(b) possibly affect (and constrain) already existing population objects, based on the correspondences summarised in table 4.2.

Object Dependency however results in some hidden annotations in the appropriate implementation objects, but does not affect run-time objects during Active Modelling.

The Active Modelling phase can be fully automated as shown in chapter 5.

4.7.2 Phase 2: Interactive Prototyping

Creating population objects is the objective of the second phase. This Interactive Prototyping process allows an agile modeler or developer to create and initialize ready-to-use objects from each implementation object created in the previous phase, thus *populating* the application.

This phase cannot be supported in a fully automatic way since choices need to be made that depend on the preferences of the user. Consider for example the many-to-many relationship between the `Customer` and the `Account` entities in figure 4.3: when an actual `Customer` object is created and initialized, it is up to the user to decide how many `Account` objects this `Customer` is to refer to, which can be any number or unlimited.

Steered from the implementation objects the user is guided to create new population objects that adhere to the structure of the EER diagram. This behaviour is implemented by extending the default cloning behaviour of Self implementation objects that participate in the RTE process and can be summarised as follows:

- **Step 1**: Start with creating a new clone of the prototype of the implementation object that shares the parents.

- **Step 2**: For each relationship of the corresponding entity in the EER view interactively determine the number of actual references to population objects created from the implementation object that corresponds to the entity on the other side of the relationship. Create (interactively) this number of new population objects and add this number of direct or undirect (via a collection for example) slot references to them in the original new object.

  The interval from which this number can be selected lies between zero and the multiplicity that is defined in the EER view. This implies **Multiplicity Constraint Enforcement** (cfr. section 2.5.3) resulting in population objects that always satisfy the multiplicity at creation time. To support **Object Dependency** (cfr. section 2.5.3) the selected number of references that originate from a weak object to an object it can depend on should be at least one.

- **Step 3**: For each specialisation branch originating from the corresponding entity in the EER view interactively select a specialisation (optionally). Add a slot reference to a new automatically cloned population object of the selected type.

- **Step 4**: For each categorisation branch originating from the corresponding entity in the EER view interactively select a categorisation (optionally). Let the new population object inherit (through prototypes and through parent objects) from this selected categorisation implementation object.

- **Step 5**: For each aggregation originating from the corresponding entity in the EER view interactively determine the number of actual aggregated population objects. Create (interactively) this number of new population objects and add a direct slot reference to a collection containing them in the original new object.

- **Step 6**: For **Role Modelling** support (cfr. section 2.5.1) an interaction is generated for each role branch originating from the corresponding entity in the EER view to determine which role is performed at creation time. Realise warped inheritance hierarchies for the selected roles.

The details and a possible implementation of this prototyping operation can be found in section 5.4.

At any point in time during the first phase, the second one can be activated to create a prototype of the system modeled so far. This is considered as a kind of rapid prototyping of EER diagrams.

Table 4.4 summarises the presence of the myARTE characteristics introduced in section 2.4, in the Interactive Prototyping phase.

	Interactive Prototyping
Optimal Transformations	ϕ
Run-time Object Generations	ϕ
Multiplicity Constraint Enforcement	$\surd$
Object Dependency	$\surd$
Role Modelling	$\surd$

Table 4.4: Presence of myARTE characteristics in the Interactive Prototyping phase.

4.8 Round-Trip Engineering Examples

In this section we present Round-Trip Engineering (RTE) scenarios to illustrate the new Agile MDD practice for myARTE proposed in section 4.7. The RTE scenarios are based on the evolution scenarios that were introduced in section 2.3 and illustrate the characteristics of myARTE as defined in section 2.4.

4.8.1 RTE Scenario 1: Adding Entities to Support Insurances in the EER View

We use the EER diagram illustrated in figure 4.3 that represents a simplified banking system as an example. This diagram is extended to support simple insurance entities. We illustrate the scenario that is based on evolution scenario 1 (see section 2.3) 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`

To realise this scenario, the following actions are performed manually in the EER view at code-time by a user, followed by an automated synchronization in the other views performed by a myARTE tool:

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. Schema objects are stand-alone namespaces that are used to persistently store implementation objects. This happens in analogy to the general action of storing Self prototypes in the `globals` namespace.

   The name of the graphical entity view in the diagram is changed to `insurer`; this is propagated automatically onto the viewed implementation object.

2. In analogy 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.

Deleting attributes automatically results in the deletion of the corresponding data slot in the implementation object. Deleting an entire entity view automatically results in the deletion of the implementation object from the banking schema object.

4.8.2 RTE Scenario 2: Adding Relationships to Support Insurances in the EER View

Again we use the banking system (see figure 4.3) as an example EER diagram. This model is extended to support simple insurances. We illustrate the scenario that is based on evolution scenario 2 (see section 2.3) with the following steps:

1. Specialize the entity `employee` into `insurer` in the banking system model

2. Add a new one-to-n relation between the entities `customer` and `insurance` in the banking system model

To realise this scenario, the following actions are performed manually in the EER view at code-time by the user followed by automated synchronization steps performed by a myARTE tool:

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:* The `insurer` implementation object automatically inherits from the `employee` implementation object. Deleting the specialization in the EER view automatically results in the removal of the inheritance between the two implementation objects.

2. Add a new one-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 view automatically results in the deletion of the slot. Next, private constraint information of both implementation objects is adjusted, to ensure that future clones of these implementation objects satisfy the one-to-n cardinality constraint. During the Interactive Prototyping phase, interactions will be generated based on these constraint annotations, to fix the actual number of references between a run-time `insurer` and a run-time `customer`, as explained in section 4.8.5.

After the scenarios 1 and 2 are realised, the evolved part of the banking EER diagram is illustrated in figure 4.15.

4.8.3 RTE Scenario 3: Adding Objects to Support Insurances in the Implementation View

We use the banking system implementation that corresponds to the EER diagram in figure 4.3 as an example. This RTE scenario is based on evolution scenario 3 (see section 2.3).

When a new implementation object is created it is installed in the schema object and its entity view becomes visual in the EER view. 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:

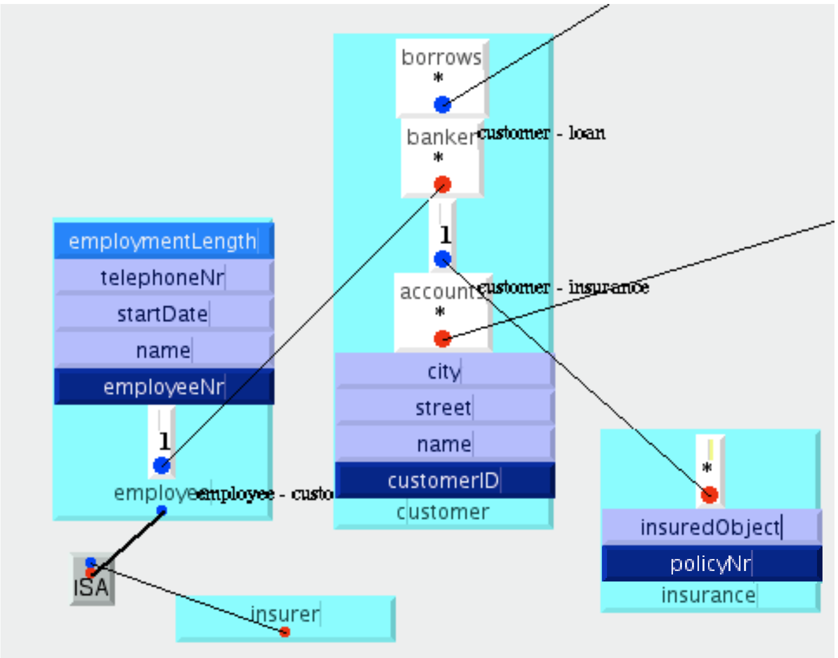


Figure 4.15: Evolved part of the banking information system model after scenario 1 and 2.

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*

To realise the entity evolution scenario, the following actions are performed manually at code-time by the user followed by automated synchronization steps performed by a myARTE tool:

1. Add a new data slot to the `insurance` implementation object via the main Self object menu and name it *insurer*. *Synchronization steps:* The `insurance` entity view in the EER view is automatically extended with a new attribute `insurer`. Note that deleting a data slot in an implementation object automatically results in the deletion of 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:* The `insurer` attribute in the `insurance` entity view in the EER view is automatically renamed to `myInsurer`.

4.8.4 RTE Scenario 4: Adding Relationships to Support Insurances in the Implementation View

We use the banking system implementation that corresponds to the EER diagram in figure 4.3 as an example. We illustrate the scenario that is based on evolution scenario 4 (see section 2.3) 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 realise the entity evolution scenario, the following actions are performed manually at code-time by the user followed by automated synchronization performed by a myARTE tool:

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:* A one-to-one relationship link is automatically drawn between the `insurance` entity view and the `insurer` entity view in the EER view, given no one-to-one link is drawn between them currently, as illustrated in figure 4.16. This synchronization is performed dynamically: when we manually remove the one-to-one link in the EER view, it is automatically re-drawn, each time the Self system updates the slots of

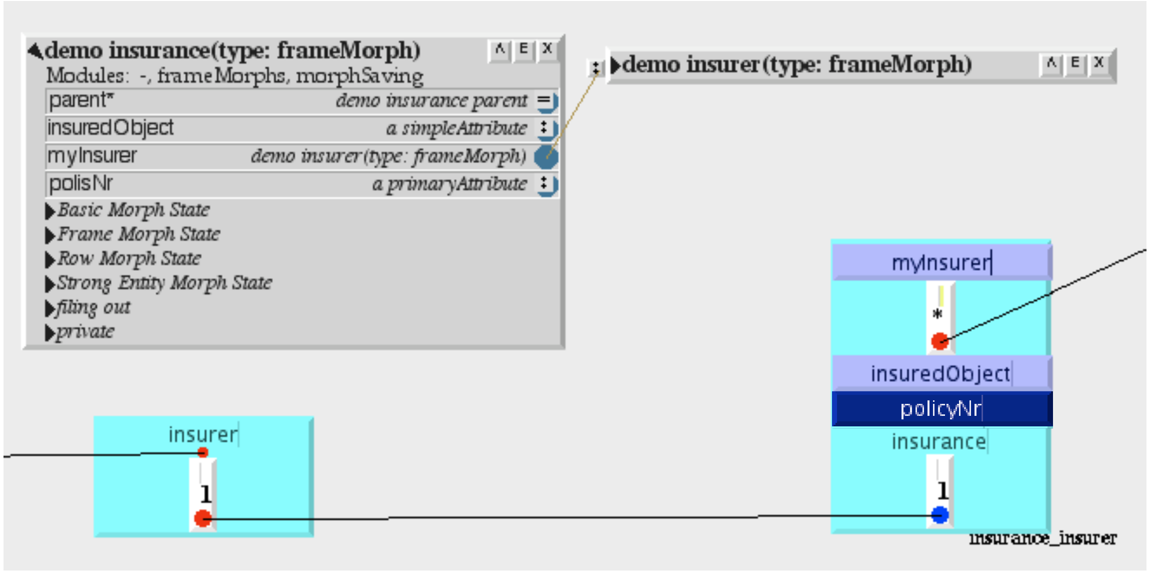
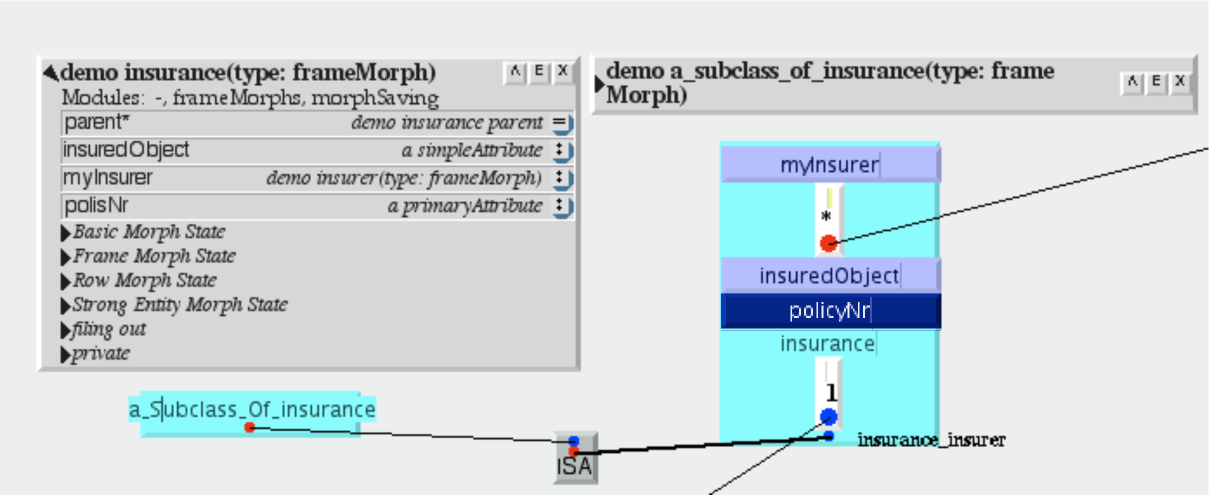


Figure 4.16: Synchronisation of slot reference into a one-to-one relationship in RTE scenario 4.

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:* A new entity view is automatically added to the EER view. Simultaneously, a new implementation object is 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 view, as illustrated in figure 4.17.



demo insurance(type: frameMorph)
Modules: -, frameMorphs, morphSaving

parent*	*demo insurance parent*
insuredObject	*a simpleAttribute*
myInsurer	*demo insurer(type: frameMorph)*
polisNr	*a primaryAttribute*

demo a_subclass_of_insurance(type: frameMorph)

Figure 4.17: Synchronisation of inheritance into a specialisation relationship in RTE scenario 4.

4.8.5 Intermezzo: An Interactive Prototyping Example

We use the EER diagram depicted in figure 4.3 that represents a modest banking system as an example. We illustrate the Interactive Prototyping phase with the following steps:

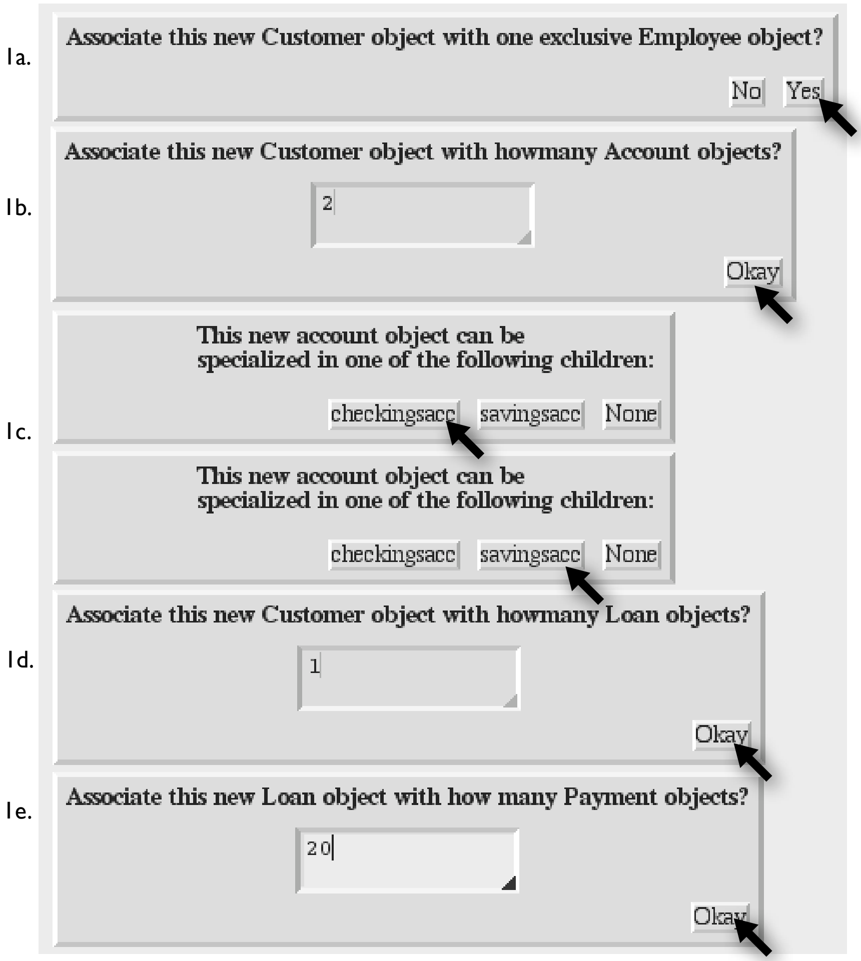
1. Create a new population object `customer` that is associated with one `employee`, that has a `checkingsAccount` and a `savingsAccount`, and pays a `loan` in twenty `payments`.

2. Create a new stand-alone `payment` population object that is not associated it with any `loan`.

To realise these steps, the following actions are performed:

1. The creation of a `customer` population object is initiated by sending the message `copy` to the `customer` implementation object. This results in a clone of the `customer` implementation object that is manipulated in a series of interactions that is are illustrated in figure 4.18.:

1a. The first interaction is generated from the one-to-many relationship between `employee` and `customer` in the EER view. In this case the new `customer` population object is allowed to have zero or one references to an `employee` population object hence the "yes/no" interaction. The "yes" button is selected resulting in a data slot being added in the new `customer` population object that contains a new `employee` population object.



Associate this new Customer object with one exclusive Employee object?

Associate this new Customer object with howmany Account objects?

2

This new account object can be
specialized in one of the following children:

This new account object can be
specialized in one of the following children:

Associate this new Customer object with howmany Loan objects?

1

Associate this new Loan object with how many Payment objects?

20

Figure 4.18: Interactions for creating a `customer` population object during Interactive Prototyping.

1b. Next an interaction is generated based on the many-to-many relationship between `customer` and `account` in the EER view. In this case an integer has to be provided to fix the number of actual slot references between the new `customer` population object and `account` population objects. Providing the integer 2 as the input results in a new data slot being added in the new `customer` population object that contains a vector with two new `account` population objects. In each of these `account` population objects a vector is added based on the many-to-many relationship between `customer` and `account` in the EER view. The first element of this vector is the new `customer` population object that is being created.

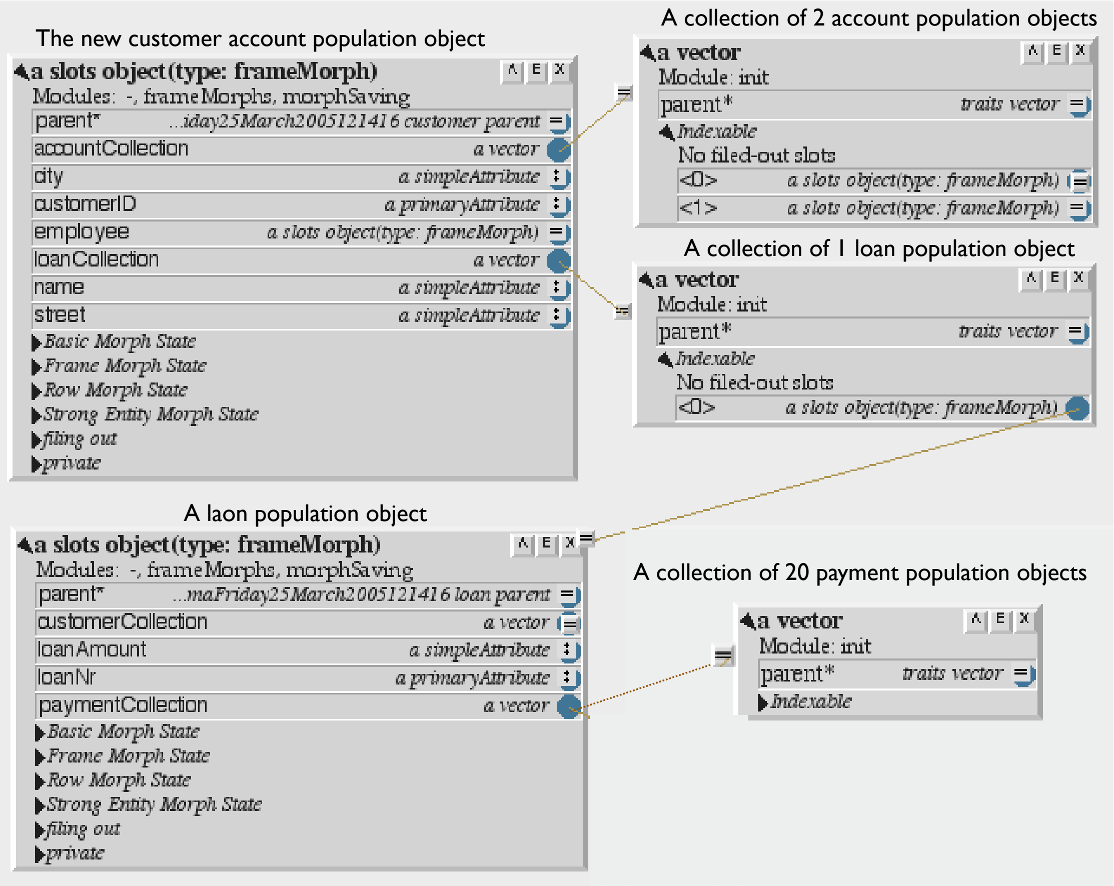
1c. During the creation of these two new `account` population objects the Interactive Prototyping phase is re-initiated. In this case for each new `account` population object an interaction is generated based on the specialisation from `account` to `savingsAccount` and `checkingsAccount` in the EER view. We select one `account` population object to be a `checkingsAccount` while the other is a `savingsAccount`.

This causes the new `account` population objects to dynamically become children of the `checkingsAccount` and the `savingsAccount` implementation objects, respectively. This means that for example the `checkingsAccount` population object will inherit the state of the `checkingsAccount` prototype trough extension while it inherits the behaviour of the `checkingsAccount` traits object via a parent slot.

1d. Next the Interactive Prototyping for the new `customer` population object continues with an interaction based on the many-to-many relationship between `customer` and `loan` in the EER view. Again an integer has to be provided to fix the number of actual slot references between the new `customer` population object and `loan` population objects. Providing the integer 1 as the input results in a new data slot being added in the new `customer` population object that contains a vector with one new `loan` population object. In each of these `loan` population objects a vector is added based on the many-to-many relationship between `loan` and `customer` in the EER view. The first element of this vector is the new `customer` population object being created.

1e. Also during the creation of this new `loan` population objects the Interactive Prototyping phase is re-initiated. An interaction is generated based on the one-to-many relationship between `loan` and the dependent `payment` in the EER view. Providing the integer 20 as the input results in a new data slot being added in the new `loan` population object that contains a vector with twenty new `payment` population objects. In each of these `payment` population objects a vector is added based on the one-to-many relationship between `loan` and `payment` in the EER view. The first element of this vector is the new loan population object that was created.

The resulting customer population object is illustrated in figure 4.19.


The new customer account population object
A collection of 2 account population objects
A collection of 1 loan population object
A laon population object
A collection of 20 payment population objects
A E X

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* ...iday25March2005121416 customer parent
accountCollection a vector
city a simpleAttribute
customerID a primaryAttribute
employee a slots object(type: frameMorph)
loanCollection a vector
name a simpleAttribute
street a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

a vector
Module: init
parent* traits vector
Indexable
No filed-out slots
<0> a slots object(type: frameMorph)
<1> a slots object(type: frameMorph)

a vector
Module: init
parent* traits vector
Indexable
No filed-out slots
<0> a slots object(type: frameMorph)

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* ...maFriday25March2005121416 loan parent
customerCollection a vector
loanAmount a simpleAttribute
loanNr a primaryAttribute
paymentCollection a vector
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

a vector
Module: init
parent* traits vector
Indexable

Figure 4.19: The resulting customer population object after Interactive Prototyping .

2. The creation of a payment population object is initiated by sending the message copy to the payment implementation object. After creating a new clone of the payment implementation object, the result is one interaction to add zero or one references to a loan population object. We provide zero as the input. *Constraint checking steps:* in the case of no references the population object creation mechanism checks the new population object for dependencies. On detecting the dependency of the new payment population object to loan population objects zero is rejected as number of references since at least one association has to be made between the new payment population object and a loan population object it can depend on. Again the interaction for a reference to the loan population object appears. This process will repeat itself until at least one reference is established. This illustrates how **Object Dependency** is

supported during the Interactive Prototyping phase.

4.8.6 RTE Scenario 5: Changing Population Objects

We use the EER diagram depicted in figure 4.3 that represents a modest banking system as an example. We illustrate the scenario that is based on evolution scenario 5 (see section 2.3) with the following steps:

1. Create a new population object `loan` and associate it to one dependent `payment` population object

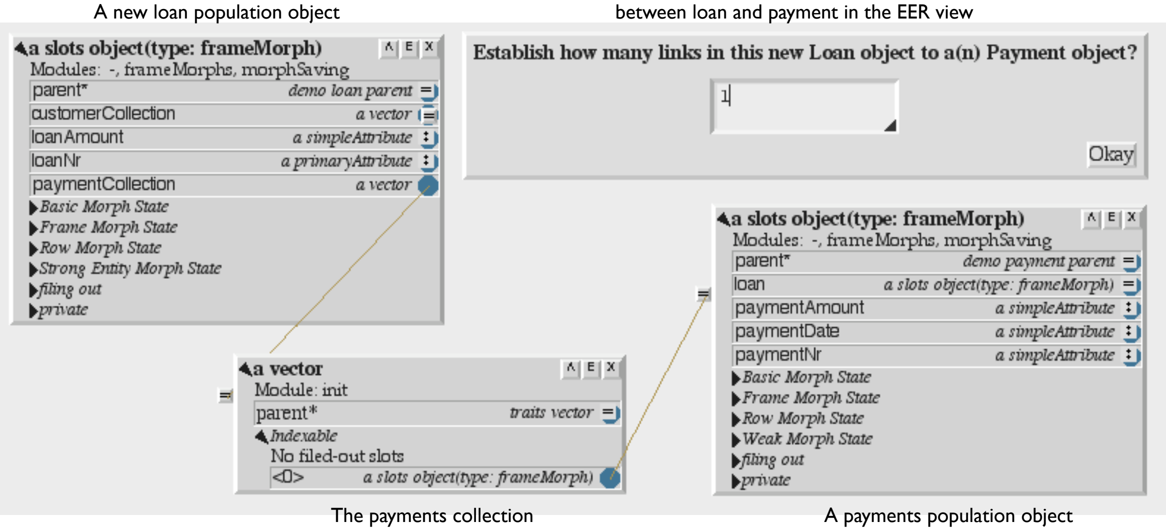
2. Delete the new `loan` population object

To realise this scenario, the following actions are performed manually in the implementation and population view by a user, possibly followed by an automated constraint checking process performed by a myARTE tool:

1. A `loan` population object can be created by sending it the message `copy` to the `loan` implementation object. This initiates the Interactive Prototyping phase. The first interaction is a request for an actual number of references to `payment` population objects. In this case only one reference is added. Next the other interactions are filled in resulting in a new `loan` population object that references one `payment` population object as illustrated in figure 4.20

Interaction based on one-to-many relationship between loan and payment in the EER view

A new loan population object



a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo loan parent
customerCollection a vector
loanAmount a simpleAttribute
loanNr a primaryAttribute
paymentCollection a vector
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

Establish how many links in this new Loan object to a(n) Payment object?
1
Okay

a vector
Module: init
parent* traits vector
Indexable
No filed-out slots
<0> a slots object(type: frameMorph)

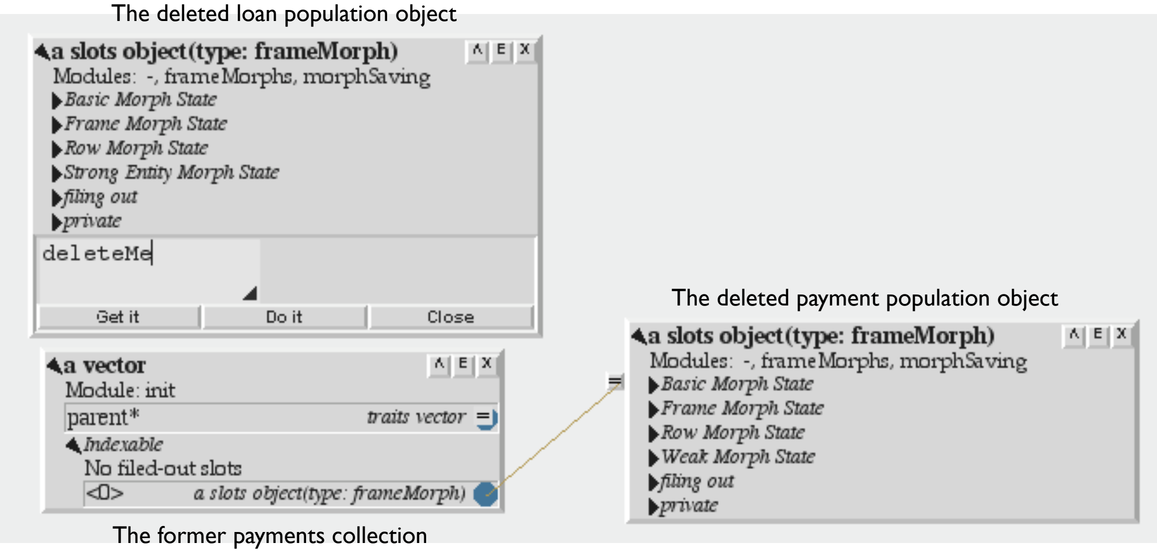
a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo payment parent
loan a slots object(type: frameMorph)
paymentAmount a simpleAttribute
paymentDate a simpleAttribute
paymentNr a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Weak Morph State
filing out
private

The payments collection

A payments population object

Figure 4.20: Creating a new `loan` population object in RTE scenario 5.

2. In the next step we delete the `loan` population object by sending it the message `deleteMe`. *Constraint checking steps:* first all references of the `loan` population object are checked for dependent population objects. The referenced dependent `payment` population object is deleted. This is simulated by removing all the public slots from this object. Next all the public slots of the `loan` population object are removed as well. This situation is illustrated in figure 4.21.



The deleted loan population object

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving

```
deleteMe
```

a vector
Module: init
parent* traits vector
Indexable
No filed-out slots
<0> a slots object(type: frameMorph)

The former payments collection

The deleted payment population object

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving

Figure 4.21: After deleting the `loan` population object in RTE scenario 5.

This illustrates how **Object Dependency** (cfr. section 2.5.3) is supported after the Interactive Prototyping phase.

4.8.7 RTE Scenario 6: Changing Relationships Between Population Objects

We use the EER diagram illustrated in figure 4.3 that represents a modest banking system as an example, except for the many-to-many relationship between Customer and Account that has been changed to a many-to-2 relationship as depicted in figure 4.22.

We illustrate this scenario that is based on evolution scenario 5 (see section 2.3) with the following steps:

1. Create a new `customer` population object that is associated with two `account` population objects.

2. Add a new reference to a third `account` population object

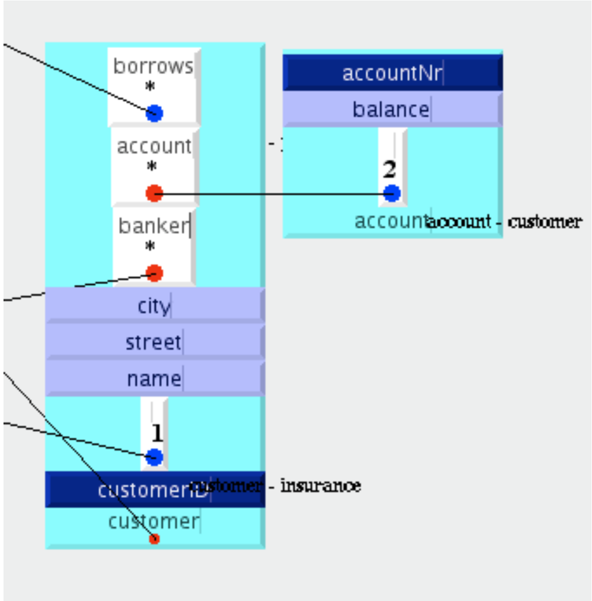
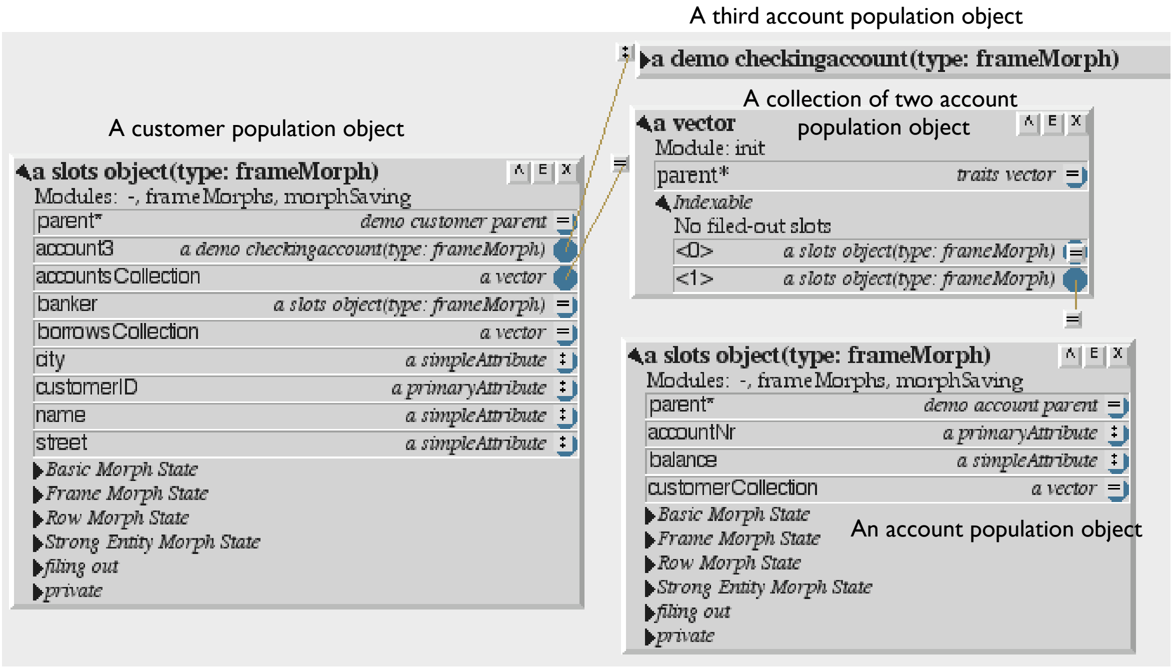


Figure 4.22: A many-to-2 relationship between `customer` and `account` in RTE scenario 6.

To realise this scenario, the following actions are performed manually in the implementation and population view by a user, possibly followed by an automated constraint checking process performed by a myARTE tool:

1. We derive a run-time `customer` from the `customer` implementation object by sending it the message `copy`. This initiates the Interactive Prototyping phase. One interaction that is generated during this phase is a request for an actual number of references to `account` population objects. The interaction already indicates that the number thas to be selected from the interval between 0 and 2. This illustrates how **Multiplicity Constraint Enforcement** (see section 2.5.3) is supported during the Interactive Prototyping phase. In this example two references are added. Next the other interactions are filled in resulting in a new `customer` run-time object that references two run-time `account` objects.

2. We manually add a new reference in the `customer` population object by adding a new data slot that references an existing `account` population object cfr. figure 4.23. *Constraint checking steps:* changing the state of a run-time object by adding a slot that references a constraining run-time object, triggers the first run-time object to be checked for satisfying multiplicity constraints. In this example, first, all the references of the population object are scanned for `account` population objects. When iteratively two more references to a run-time `account`


A third account population object
A collection of two account population object
A customer population object
An account population object

a demo checkingaccount(type: frameMorph)

a vector
Module: init
parent* traits vector
Indexable
No filed-out slots
<0> a slots object(type: frameMorph)
<1> a slots object(type: frameMorph)

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo customer parent
account3 a demo checkingaccount(type: frameMorph)
accountsCollection a vector
banker a slots object(type: frameMorph)
borrowsCollection a vector
city a simpleAttribute
customerID a primaryAttribute
name a simpleAttribute
street a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

a slots object(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo account parent
accountNr a primaryAttribute
balance a simpleAttribute
customerCollection a vector
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

Figure 4.23: Violation of the many-to-2 constraint between `customer` and `account` in RTE scenario 6.

object are detected, a warning is generated.

Note that immediately after a multiplicity change also all appropriate population objects that already exist are checked and for all of them that violate the multiplicity constraint a warning is generated.

This illustrates how **Multiplicity Constraint Enforcement** (cfr. section 2.5.3) is supported after the Interactive Prototyping phase.

4.8.8 RTE Scenario 7: Role Modelling

This scenario merges the evolution scenarios 1, 5 and 6 (see section 2.3) and the interactive prototyping phase, in order to illustrate how Role Modelling is supported in myARTE. We use the EER diagram illustrated in figure 4.3 that represents a modest banking system as an example. We will extend this diagram with the notion that employees cannot be customers. As a consequence employee and customer are two mutually exclusive roles, as illustrated in figure 4.24.

This scenario consists of the following steps:

1. Create a new entity `person` in the EER view.

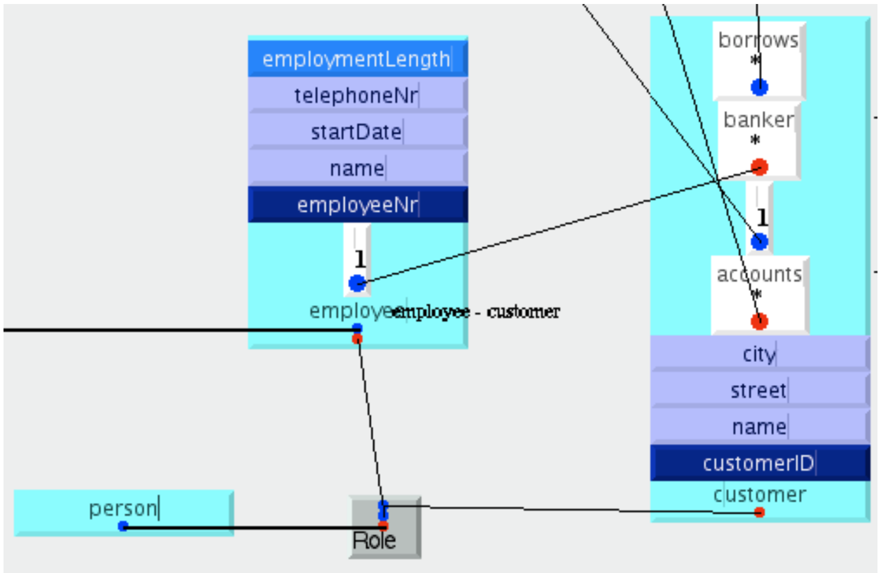


Figure 4.24: Role relation from `person` to `customer` and `employee` in RTE scenario 7.

2. Add a new role relationship from `person` to `employee` and `customer`.

3. Create a new run-time `person` population object.

4. Dynamically let this `person` start performing the role of `employee` in the population view.

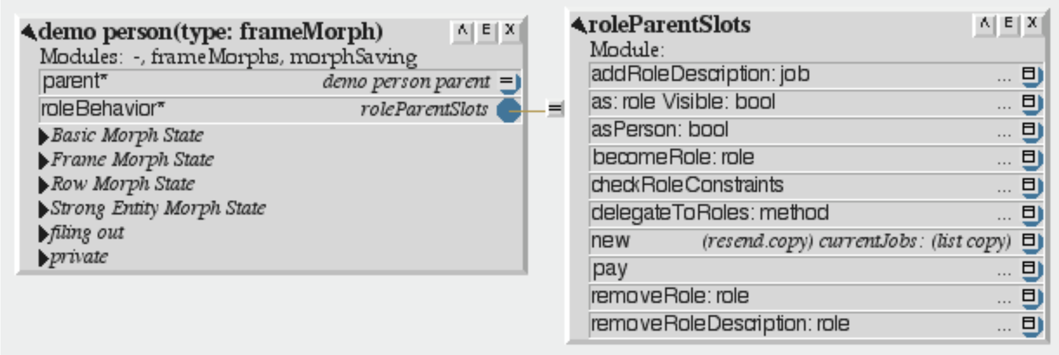
5. Dynamically let this `person` start performing the role of `customer`.

6. Dynamically let this `person` stop performing the role of `employee`.

From the EER View to the Implementation View To realise steps 1 and 2 of this scenario, the following actions are performed manually in the EER view by a user, followed by an automated synchronisation process performed by a myARTE tool:

1. Add a new blank entity view to the EER diagram via the appropriate menu and name it `person`. *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 `person`, this is propagated automatically to the viewed implementation object.

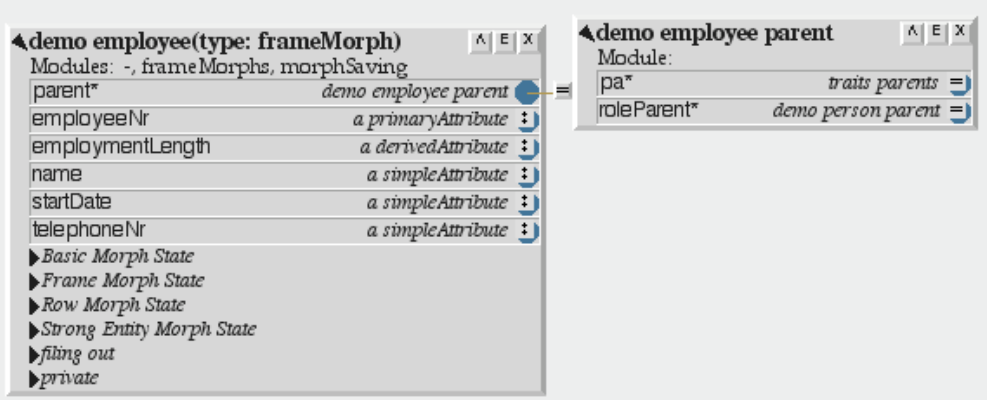
2. Add a new role relation to the EER diagram from the entity view `person` to the entity views `employee` and `customer`, via the appropriate menu. *Synchronization steps:* The `person` implementation object automatically inherits from the `roleParentSlots` object that defines general behaviour for roles (see figure 4.25). The parent part of the `customer` and `employee` implementation objects inherit from the parent part of the `person` implementation object as illustrated in figure 4.26. In the private slots of `person` a role set with the names of the `employee` and the `customer` role is added to denote that they were part of the same role branch and thus are mutually exclusive.



demo person(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo person parent
roleBehavior* roleParentSlots
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

roleParentSlots
Module:
addRoleDescription: job ...
as: role Visible: bool ...
asPerson: bool ...
becomeRole: role ...
checkRoleConstraints ...
delegateToRoles: method ...
new (resend.copy) currentJobs: (list copy)
pay ...
removeRole: role ...
removeRoleDescription: role ...

Figure 4.25: The `person` implementation object inherits general role behaviour in RTE scenario 7.



demo employee(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo employee parent
employeeNr a primaryAttribute
employmentLength a derivedAttribute
name a simpleAttribute
startDate a simpleAttribute
telephoneNr a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

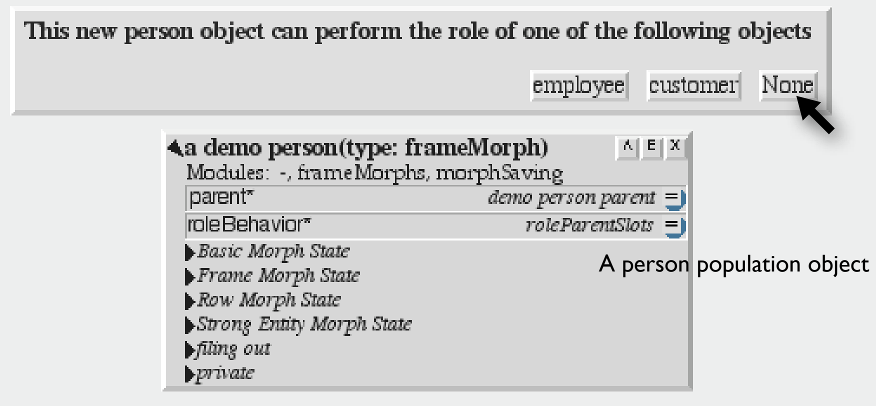
demo employee parent
Module:
pa* traits parents
roleParent* demo person parent

Figure 4.26: The parent part of the `employee` implementation object inherits from the parent part of the `person` implementation object in RTE scenario 7.

Deleting the role in the EER view would automatically result in removing the three inheritance relations in the implementation objects and adjusting the aforementioned private slots.

From the Implementation View to the Population View To realise step 3 of this scenario, the following actions are performed manually in the implementation view by a user, followed by a semi-automated, interactive prototyping process performed by a myARTE tool:

1. A `person` population object can be created by sending the message `copy` to the `person` implementation object. This initiates the Interactive Prototyping phase. The only interaction is a request to determine the role that this person performs at creation time (see figure 4.27). This interaction is based on the private slots of `person` that contain a role set with the names of the `employee` and the `customer` roles. We decide that this new `person` population object performs no roles yet.



This new person object can perform the role of one of the following objects

a demo person(type: frameMorph)

Modules: -, frameMorphs, morphSaving

parent* demo person parent

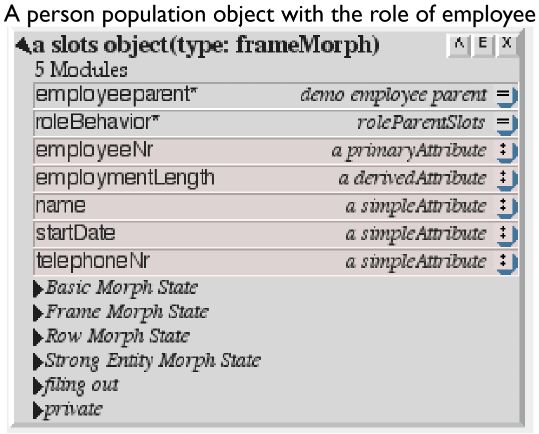
roleBehavior* roleParentSlots

Figure 4.27: The interactive prototyping of a new `person` population object in RTE scenario 7.

In the Population View To realise steps 4, 5 and 6 of this scenario, the following actions are performed manually in the population view by a user, followed by an automated checking process performed by a myARTE tool:

1. The run-time `person` object starts performing the role of employee when it receives the message `becomeRole:'banking employee'`. The slots in the prototype of the `employee` implementation object are automatically copied down in the prototype of the run-time `person` population object. In the prototype of the run-time `person` population object a parent slot is added that contains the parent part of the `employee` implementation object. This is illustrated in figure 4.28.

2. The run-time `person` object starts performing the role of customer when it receives the message `becomeRole:'banking customer'`. *Constraint checking steps:* based on the `employee`

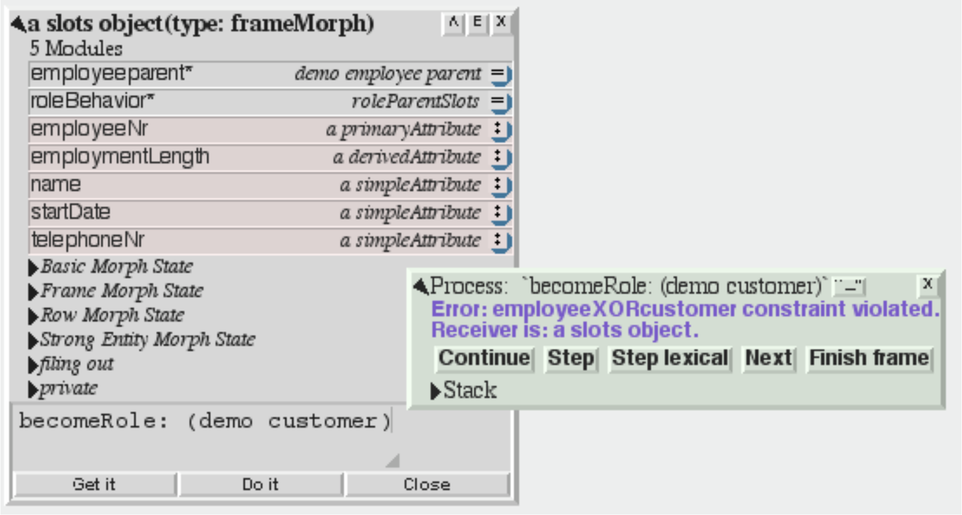


A person population object with the role of employee

a slots object(type: frameMorph)
5 Modules

```
employeeparent*         demo employee parent
roleBehavior*           roleParentSlots
employeeNr              a primaryAttribute
employmentLength        a derivedAttribute
name                    a simpleAttribute
startDate               a simpleAttribute
telephoneNr             a simpleAttribute
```

Figure 4.28: A `person` population object with the role of `employee` in RTE scenario 7.



a slots object(type: frameMorph)
5 Modules

```
employeeparent*         demo employee parent
roleBehavior*           roleParentSlots
employeeNr              a primaryAttribute
employmentLength        a derivedAttribute
name                    a simpleAttribute
startDate               a simpleAttribute
telephoneNr             a simpleAttribute
```

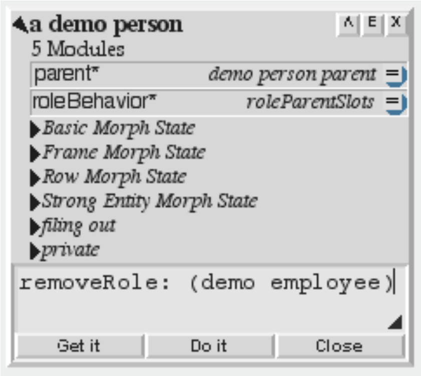
```
becomeRole: (demo customer)
```

Process: `becomeRole: (demo customer)`
Error: employeeXORcustomer constraint violated.
Receiver is: a slots object.

Figure 4.29: A `person` population object with the role of `employee` is not allowed to perform the role of `customer` in RTE scenario 7.

role that is already performed it is automatically checked whether `employee` and `customer` were part of the same role branch in the EER view. This is achieved by consulting the private slots of the run-time `person` population object, that contain a role set with the names of the `employee` and the `customer` roles. Since both roles are in the same role set they are mutually exclusive and therefore automatically an error is generated as illustrated in figure 4.29.

3. The run-time `person` object stops performing the role of employee when it receives the message `removeRole:'banking employee'`. In this case all the copied slots of the `employee` implementation object are automatically removed from the run-time `person` population ob-



a demo person
5 Modules
parent* demo person parent
roleBehavior* roleParentSlots
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

```
removeRole: (demo employee)
```

Figure 4.30: A `person` population object after removing the role of `employee` in RTE scenario 7.

ject. The parent slot that contained the parent object of the `employee` implementation object is also removed. Since `employee` was the only role performed by person, again a parent slot that containts the parent object of `person` is added automatically. This is illustrated in figure 4.30.

4.9 Conclusion

We conclude this chapter with the following statements:

- An Extended Entity-Relationship (EER) model that combines existing EER notations with Role Modelling concepts, is used to represent a graphical data modelling view. This model is implemented in a graphical drawing editor that is developed with the Morphic framework and is integrated in the Self programming environment.
- We defined a general solution to implement roles with the warped hierarchies implementation scheme.
- The Self language is selected for the implementation and the population views. Implementation objects are prototype-traits pairs while population objects are clones of these implementation objects.
- A set of default mappings between the three views was defined.
- We propose an Agile MDD practice that can be supported by myARTE tools that synchronise between an analysis view and an implementation view, and support run-time RTE programming.

- The two most important phases are Active Modelling and Interactive Prototyping. The characteristics of myARTE are present in both phases.
- During Active Modelling there is a continuous and incremental synchronisation between the EER view and the implementation view. This phase can be fully automated.
- At any point in time during Active Modelling the second phase Interactive Prototyping can be activated. During this phase population objects are interactively created based on the structure of the EER view.
- Population objects are constrained by the multiplicities of the relationships, and the dependencies between weak and strong entities in the EER view.

In the next chapter we introduce a possible implementation for an myARTE tool that automatically synchronises between an EER view and a Self implementation and population view, as proposed in the new Agile MDD practice.

Chapter 5

SelfSync: An ARTE Implementation

In this chapter we introduce an implementation of an ARTE tool called SelfSync. SelfSync is entirely written in Self and extends it with an ARTE environment. The SelfSync tool supports all the RTE scenarios that were described in section 4.8. The SelfSync UI includes a data modelling view that employs a graphical drawing editor for EER diagrams and is built with the Morphic framework (see section 5.1).

In SelfSync entities and implementation objects are represented by outliners that are in fact two views on one and the same underlying Self object. Internally SelfSync objects are represented by prototypes that combine the state of these two views and with a double traits inheritance hierarchy that also illustrates the duality of the two views (see section 5.2).

In SelfSync the Active Modelling phase is fully automated and applies Optimal Transformations between the different views (see section 5.3). In the EER view SelfSync employs a dynamically built entity menu whose actions result in annotations and synchronisation in the implementation view (see section 5.3.2). The invoked behaviour resides in the aforementioned double traits hierarchy. In the implementation view changes are realised via the Self menu, via the Self UI, and literally via the code. The invoked behaviour is scattered at different levels of the Self language: in the outliners, in the underlying slot models and in the mirrors at the meta-level (see section 5.3.3).

Run-time Object Generations are created during a semi-automated Interactive Prototyping phase (see section 5.4). The associated behaviour is implemented in a new cloning method for SelfSync objects, that generates interactions depending on their annotations.

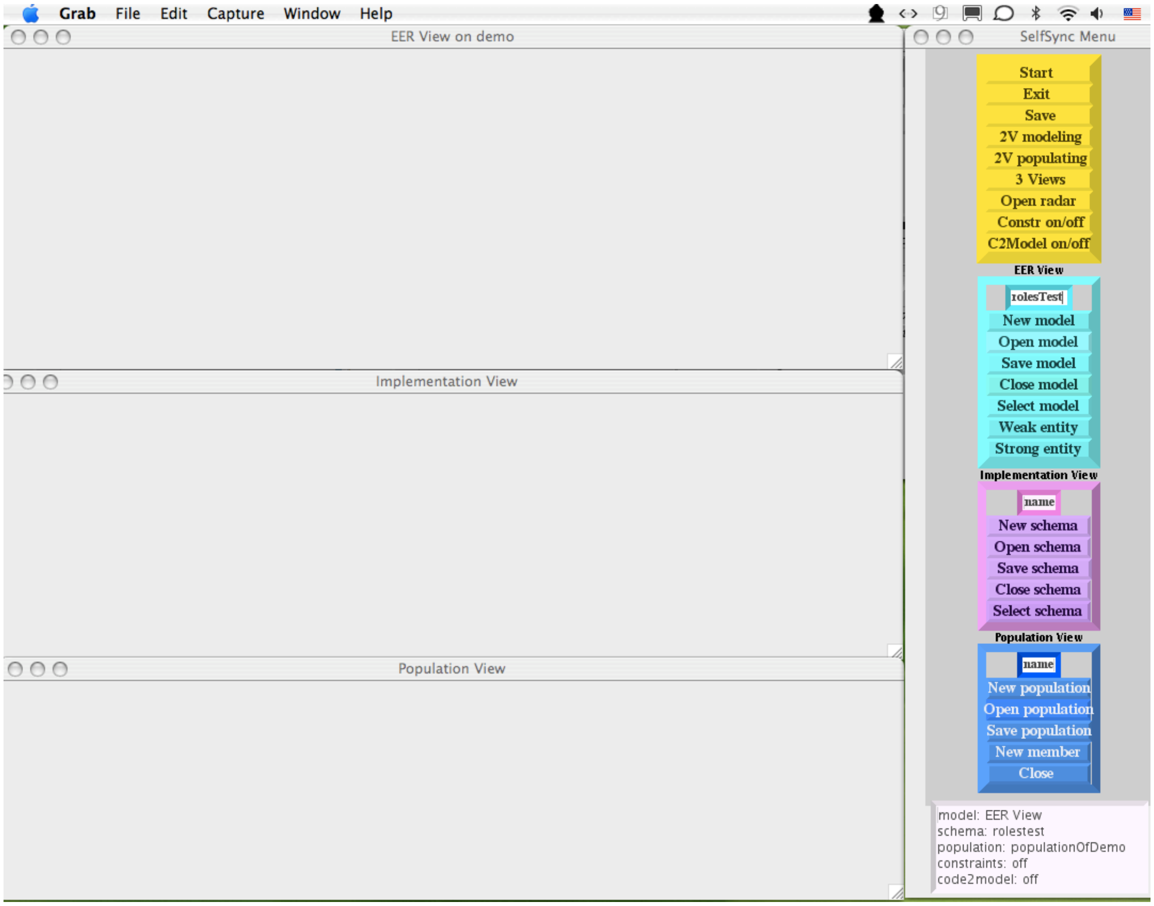
SelfSync implements Multiplicity Constraint Enforcement (see section 5.5.1) and Object Dependency (see section 5.5.2) with meta-programming constructs that mainly compare implementation objects' references with their annotated objects. Furthermore, this ARTE tool allows advanced Role

Modelling by implementing general role behaviour and a mechanism to enforce role combination constraints (see section 5.6).

We will not go into every detail of the SelfSync implementation but provide a general insight in the representations and mechanism of the tool.

5.1 The User Interface

The SelfSync User Interface (UI) is illustrated in figure 5.1. The three horizontal windows represent



model: EER View
schema: rolestest
population: populationOfDemo
constraints: off
code2model: off

Figure 5.1: The User Interface of the SelfSync ARTE tool.

an EER data modelling view, an implementation view and a population view. The EER view consists of a drawing editor for EER diagrams that use the EER notation we defined in section 4.1.1. The elements of the implementation and population views are represented by the default Self outliners.

The vertical menu window consists of 5 parts:

1. A general menu to start or exit, resize and combine views and to set options like constraint enforcement and code to model synchronisation.

2. A menu for the EER view including opening, creating, saving models and adding new entities to them.

3. A similar menu for the implementation view.

4. A menu for the population view.

5. A status window for the current EER model, the current schema that stores the implementation objects, and the selected options.

5.2 The Internal Representation of SelfSync Objects

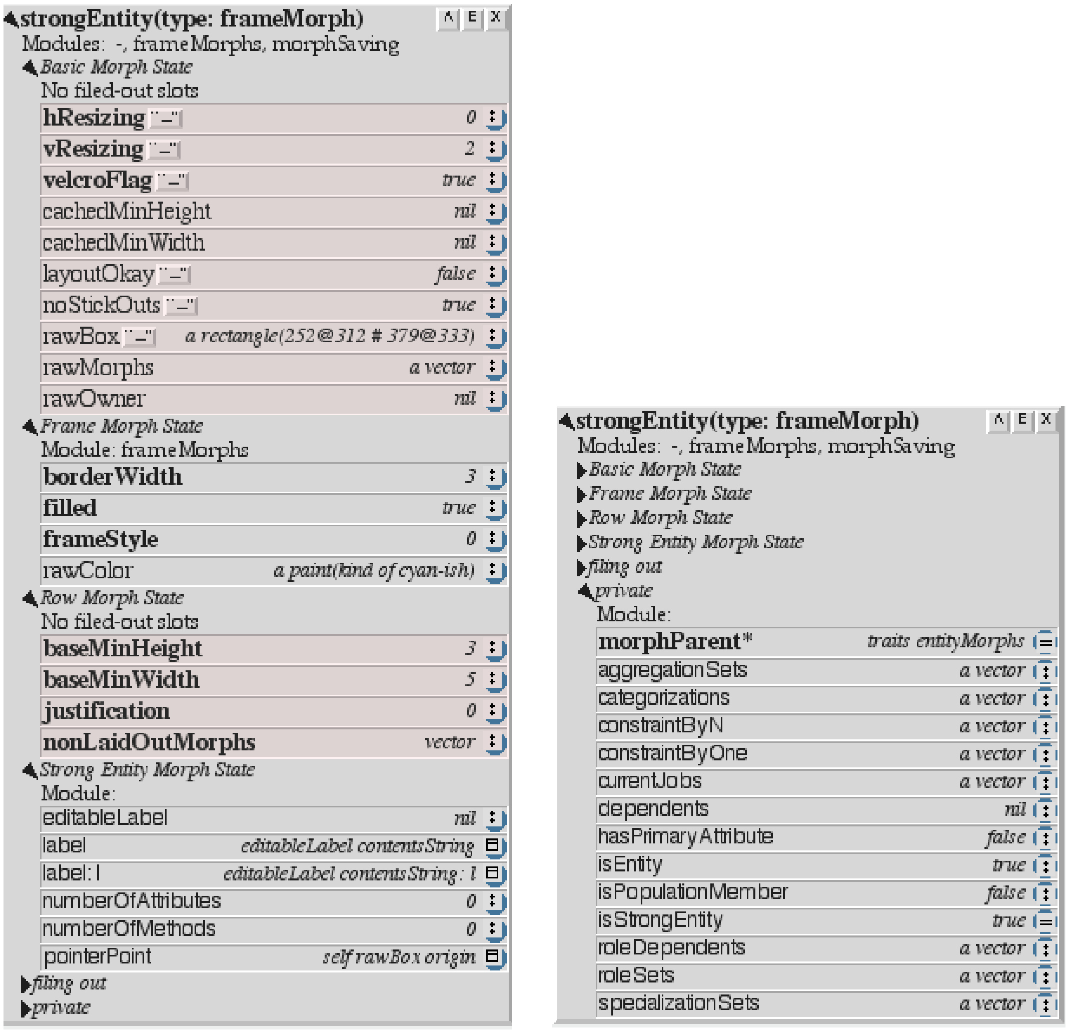
In this section we introduce the internal representation of SelfSync objects. The core of SelfSync consists of the entities of the EER view that are in fact also the implementation objects in the implementation view. Due to the Model-View-Controller (MVC) architecture of the Self User Interface, as illustrated in section 4.1.2, figure 4.6 the EER outliner that represents the entity, as well as the already existing Self outliner that represents the implementation object, are two views on one and the same underlying Self object. We first describe the main prototypes for entities (see section 5.2.1) followed by the double traits hierarchy that implements the main behaviour of SelfSync objects (see section 5.2.2).

5.2.1 Prototypes for Entities and Implementation Objects

As a consequence of entities and implementation objects being both views on the same underlying Self object, these SelfSync objects implement both state for a graphical *morph* object, and state for a real Self implementation object. This is illustrated with the prototype for strong entities[1] in figure 5.2.

The different slots are divided into categories. The categories *Basic Morph State*, *Frame Morph State* and *Row Morph State* already existed in Self and were inherited in this prototype. In SelfSync the categories *Strong Entity Morph State* and *private* were added. In *Strong/Weak Entity Morph State* the state of strong or weak EER entities is contained such as the editable `label` in an entity that represents its name.

[1] Weak entities are implemented similarly except for the fact that they cannot have a primary attribute and that they include an extra attribute for future references to a strong population object they can depend on.



```
strongEntity(type: frameMorph)
  Modules: -, frameMorphs, morphSaving
  Basic Morph State
    No filed-out slots
    hResizing "–"                                 0
    vResizing "–"                                 2
    velcroFlag "–"                             true
    cachedMinHeight                             nil
    cachedMinWidth                              nil
    layoutOkay "–"                            false
    noStickOuts "–"                            true
    rawBox "–"      a rectangle(252@312 # 379@333)
    rawMorphs                              a vector
    rawOwner                                    nil
  Frame Morph State
    Module: frameMorphs
    borderWidth                                   3
    filled                                     true
    frameStyle                                    0
    rawColor             a paint(kind of cyan-ish)
  Row Morph State
    No filed-out slots
    baseMinHeight                                 3
    baseMinWidth                                  5
    justification                                 0
    nonLaidOutMorphs                         vector
  Strong Entity Morph State
    Module:
    editableLabel                               nil
    label            editableLabel contentsString
    label: l       editableLabel contentsString: l
    numberOfAttributes                            0
    numberOfMethods                               0
    pointerPoint              self rawBox origin
  filing out
  private
```

```
strongEntity(type: frameMorph)
  Modules: -, frameMorphs, morphSaving
  Basic Morph State
  Frame Morph State
  Row Morph State
  Strong Entity Morph State
  filing out
  private
    Module:
    morphParent*             traits entityMorphs
    aggregationSets                     a vector
    categorizations                     a vector
    constraintByN                       a vector
    constraintByOne                     a vector
    currentJobs                         a vector
    dependents                               nil
    hasPrimaryAttribute                    false
    isEntity                                true
    isPopulationMember                     false
    isStrongEntity                          true
    roleDependents                      a vector
    roleSets                            a vector
    specializationSets                  a vector
```

Figure 5.2: The strong entity prototype with state for a graphical entity (left) and for a SelfSync implementation object (right).

In the *private* category the internal state of SelfSync implementation objects is contained. For example, `hasPrimaryAttribute` is a predicate to indicate whether the current implementation object already contains a primary attribute in the EER view.

The `constraintBy1` and `constraintByN` collections include implementation objects that constrain the current implementation object based upon relationships in the EER view. For example, if the entities `Mother` and `Child` are in a 1-to-3 relationship in the EER view then first `Child`'s `constraintBy1` includes `'mother'` and secondly, `Mother`'s `constraintByN` includes the pair `('child',3)`.

5.2.2 A Double Traits Inheritance Hierarchy

As said before SelfSync represents EER outliners and Self outliners that view the same underlying Self object. This duality of graphical EER outliner and implementation object is also reflected in the behaviour of these objects, and is implemented in a double traits inheritance hierarchy from which the prototypes for strong and weak entities inherit their behaviour. This setup is illustrated in figure 5.3. The right top branch represents the behaviour of the graphical EER outliners that is based on Self's built-in traits objects such as `traits rowMorph`. The remaining left hierarchy consists of `traits entityMorphs`, `traits ERMorphs` and `traits entity`.

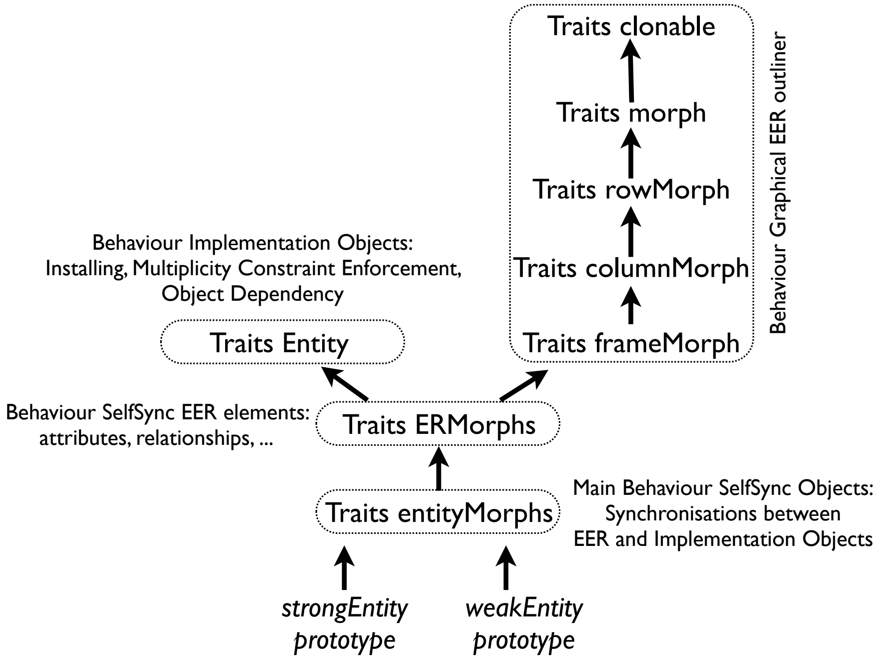

Traits clonable
Traits morph
Traits rowMorph
Traits columnMorph
Traits frameMorph
Behaviour Graphical EER outliner
Behaviour Implementation Objects:
Installing, Multiplicity Constraint Enforcement,
Object Dependency
Traits Entity
Behaviour SelfSync EER elements:
attributes, relationships, ...
Traits ERMorphs
Traits entityMorphs
Main Behaviour SelfSync Objects:
Synchronisations between
EER and Implementation Objects
strongEntity
prototype
weakEntity
prototype

Figure 5.3: The main inheritance hierarchy in SelfSync.

The `traits entityMorphs` is the core traits object of SelfSync objects at both the EER and the implementation view and is responsible for the synchronisation from the EER view to the implementation view. `traits ERMorphs` defines more general behaviour that is inherited from the other elements of the EER view: attributes and relationships for example. Finally `traits entity` is responsible for installing the implementation objects persistently is a schema namespace, for Multiplicity Constraint Enforcement and for Object Dependency.

We briefly discuss the different kinds of behaviour:

- **Traits entityMorphs** (see figure 5.4): this object implements the main synchronisation behaviour from the EER view to the implementation view[2] during the Active Modelling phase and the creation process during Interactive Prototyping phase. In the public part of this object we find methods such as `buildEntityMenu` that dynamically builds the menu to manipulate EER entities, that was illustrated in section 4.1.1, figure 4.2. Based on the internal state of the Self object that is viewed with the current EER outliner an appropriate menu is generated. For example when it concerns a strong entity the menu will contain the option to add a primary attribute as opposed to the case of a weak entity.

Another example of a method in this traits object is the `new:l` method that is used to create and initiate a new implementation object with name `l`.

The methods `constrOne` and `constrN` dynamically access the names of the constraining objects that are contained in the `constraintBy1` or `constraintByN` collections of the receiver implementation object.

The modelling and synchronisation actions that are demanded during Active Modelling are included in the categories *entities*, *relationships* and *targets*. In the category *entities* mainly behaviour is stored to add new attributes to entities.

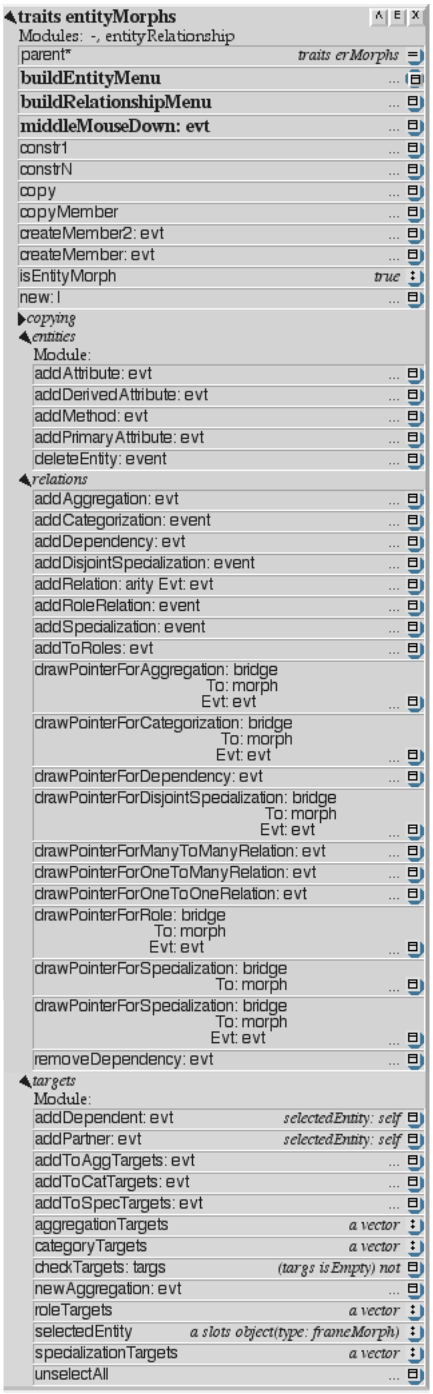
The category *relationships* methods are included to actually establish relationships, including specialisations, etc., when such an action is selected from the EER menu. For example when a new one-to-one relationship is created from an entity, the method `addRelation:1Evt:evt` is called, that in its turn will call `drawPointerForOneToOneRelation:evt`. The latter will actually draw the new link in the EER view after annotating the appropriate implementation objects.

In *targets* methods and collections are contained for (un)selecting targets of relationships. For example when a new one-to-one relationship is created from an entity it is demanded that beforehand a partner entity was selected with `addPartner:evt` as the target of this new relationship.

Interactive Prototyping behaviour can be found in the category *copying*.

- **Traits ERMorphs**: includes some predicates to determine the kind of EER element, such as `isAttributeMorph`. The other morphs in the EER view for attributes, relationships, etc. inherit from this object.

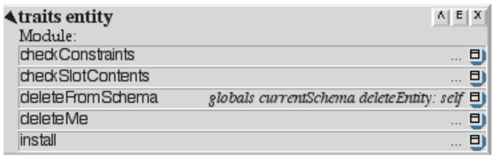
[2] The synchronisation from the implementation view to the EER view is scattered among different already existing traits objects as described in section 5.3.3.



```
traits entityMorphs                                     ^ E X
Modules: -, entityRelationship
parent*                                      traits erMorphs
buildEntityMenu                                          ...
buildRelationshipMenu                                    ...
middleMouseDown: evt                                     ...
constr1                                                  ...
constrN                                                  ...
copy                                                     ...
copyMember                                               ...
createMember2: evt                                       ...
createMember: evt                                        ...
isEntityMorph                                           true
new: l                                                   ...
copying
entities
  Module:
  addAttribute: evt                                      ...
  addDerivedAttribute: evt                               ...
  addMethod: evt                                         ...
  addPrimaryAttribute: evt                               ...
  deleteEntity: event                                    ...
relations
  addAggregation: evt                                    ...
  addCategorization: event                               ...
  addDependency: evt                                     ...
  addDisjointSpecialization: event                       ...
  addRelation: arity Evt: evt                            ...
  addRoleRelation: event                                 ...
  addSpecialization: event                               ...
  addToRoles: evt                                        ...
  drawPointerForAggregation: bridge
                        To: morph
                       Evt: evt                          ...
  drawPointerForCategorization: bridge
                           To: morph
                          Evt: evt                       ...
  drawPointerForDependency: evt                          ...
  drawPointerForDisjointSpecialization: bridge
                                   To: morph
                                  Evt: evt               ...
  drawPointerForManyToManyRelation: evt                  ...
  drawPointerForOneToManyRelation: evt                   ...
  drawPointerForOneToOneRelation: evt                    ...
  drawPointerForRole: bridge
                 To: morph
                Evt: evt                                 ...
  drawPointerForSpecialization: bridge
                           To: morph                     ...
  drawPointerForSpecialization: bridge
                           To: morph
                          Evt: evt                       ...
  removeDependency: evt                                  ...
targets
  Module:
  addDependent: evt                    selectedEntity: self
  addPartner: evt                      selectedEntity: self
  addToAggTargets: evt                                   ...
  addToCatTargets: evt                                   ...
  addToSpecTargets: evt                                  ...
  aggregationTargets                               a vector
  categoryTargets                                  a vector
  checkTargets: targs                (targs isEmpty) not
  newAggregation: evt                                    ...
  roleTargets                                      a vector
  selectedEntity         a slots object(type: frameMorph)
  specializationTargets                            a vector
  unselectAll                                            ...
```

Figure 5.4: The main `traits entityMorphs` object in SelfSync.

- **Traits Entity** (see figure 5.5): implements methods to install or delete an implementation object in a schema namespace, to iterate over the slots of an implementation object, and to enforce the constraints that are valid for an implementation object.



```
traits entity
Module:
checkConstraints                                             ...
checkSlotContents                                            ...
deleteFromSchema           globals currentSchema deleteEntity: self
deleteMe                                                     ...
install                                                      ...
```

Figure 5.5: The `traits entity` object responsible for among other things Multiplicity Constraint Enforcement in SelfSync.

- **Traits frameMorph, Traits columnMorph, Traits rowMorph, Traits morph**: implement methods for specific kinds of morphs. These traits objects already exist in Self and are reused to implement the graphical representation of entities in the EER view.
- **Traits clonable**: The traits object that implements behaviour for objects that can be cloned. Most Self objects inherit from it.

Now that the core structure of SelfSync is introduced we continue with highlighting how SelfSync implements Active Modelling, Interactive Prototyping and constraint enforcement for multiplicities, dependency and roles.

5.3 The Active Modelling Implementation

In this section we describe how SelfSync implements Active Modelling. The start of this phase is introduced in section 5.3.1. How SelfSync implements modelling in the EER view is described in section 5.3.2. How changes are implemented in the implementation view can be found in section 5.3.3.

5.3.1 Initial Setup

During Active Modelling in SelfSync, initially a new EER diagram is created in the EER view by using the appropriate buttons in EER menu of the menu window (see figure 5.1). The future entities of this diagram will correspond to future implementation objects. The representation of these entities and implementation objects are in fact EER outliners and Self outliners that view the same underlying Self object. For each new created EER diagram, a new schema is automatically created to store future implementation objects that correspond to the future entities of the new EER diagram. As said before such a schema functions as a namespace to persistently install new implementation objects that correspond to one EER diagram. Creating a new EER diagram mainly involves window operations. Creating a new schema implies cloning SelfSync's `schema` prototype.

For each new entity that is added to the diagram with the EER menu, the schema in the implementation view is extended with a new slot[3] that contains the corresponding implementation object for each newly created entity.

To create new entities, clones are made from the `strongEntity` or `weakEntity` prototype with the help of a `new:1` method implemented in `Traits entityMorphs`. Initially new entities have the label `STRONG`. Next we install the new entity as implementation object in the schema with `install` for which behaviour is implemented in `Traits Entity`. This results in a new slot in the schema with a random name such as `strong_entity917`. The contained implementation object is then persistently saved to carry the same name. When the name of the entity in the EER view is changed, this name change is propagated to the aforementioned slot in the schema and thus also to the contained implementation object.

5.3.2 Modelling in the EER View

Active Modelling in the EER view mainly involves invoking the EER menu on entities. This menu is dynamically built for this entity with `buildEntityMenu`. The actions of the resulting menu are linked to the methods in `Traits entityMorphs` that are stored in its categories

- *entities* that include methods for adding new attributes to an entity or deleting an entity. These methods do not need previously selected targets.
- *relationships* that contains methods to establishing a new 1-to-1, 1-to-n or m-to-n relationship or a new dependency, a new specialisation, a new aggregation, a new categorisation or a new role relationship. These methods need previously selected targets.
- *targets* with methods and collections for (un)selecting targets in relationships, specialisation, etc.

For example when the action `"One-to-One Relationship"` is selected from the menu the method `addRelation:1 Evt:evt` of the *relationships* category is called.

The default order of actions is as follows:

1. Middle-click on an entity in the EER view. This results in the EER menu (see figure 4.2) being dynamically built for the underlying implementation object with `buildEntityMenu`. This menu contains a number of actions based on the left column of the mappings in table 4.1,

[3] Analoguously to Self where each new prototype is installed in a new slot in the `globals` namespace.

that are appropriate for the selected entity. Each action corresponds to a method in `Traits entityMorphs`. Next the menu appears in the EER view.

In the case of actions that do not demand a target being selected previously, we immediately continue with step 3. In the other cases a target is necessary we continue with step 2.

2. The entity is selected as a target via the EER menu. Such a target can be a

   - Partner entity for a future 1-to-1, 1-to-n or m-to-n relationship
   - Srong partner entity or a future dependency
   - One of the specialised targets for a future specialisation
   - One of the aggregated targets for a future aggregation
   - One of the categorised targets for a future categorisation
   - One of the role targets for a future role relationship

   The methods to select the targets can be found in the *targets* category. The collections to contain the targets themselves are also implemented as attributes in the `Traits entityMorphs`. It is also possible to remove all previously selected targets. Next continue with step 1.

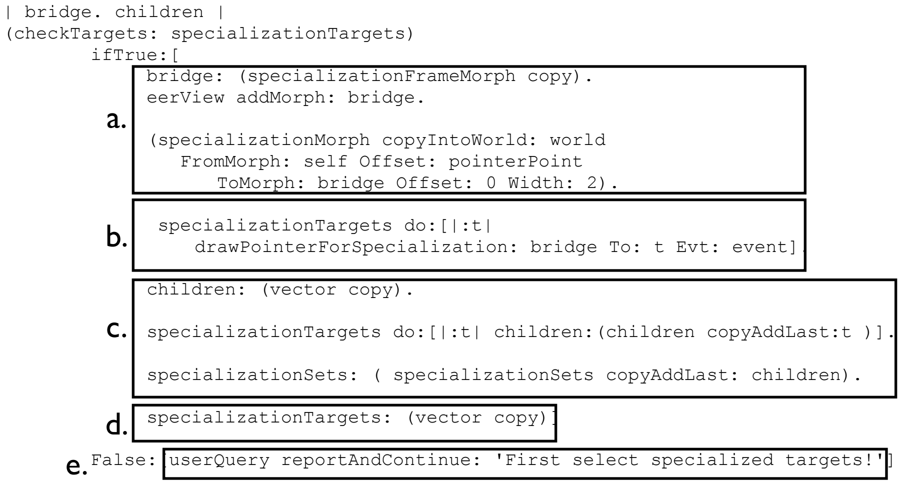
3. Select an action or relation in the EER menu that is linked to a method in `Traits entityMorphs`.

4. Calling such a mehod results in a number of automated actions. We illustrate them with the implementation of the method to create a new specialisation, cfr. figure 5.6.

   a. The method starts with making part of the initiated change visible in the EER view (cfr. figure 5.6 (a)). More precisely a new symbol for a specialisation relationship is added to the EER view, that is connected to the general entity with a link.

   b. Next automated synchronisation steps are initiated in the implementation view (cfr. figure 5.6 (b)). This implies that each of the selected child implementation objects actually starts inheriting from the parent implementation object. Specialisation links are also drawn from each specialised entities to the symbol of the specialisation relationship in the EER view.

   c. To ensure that future population objects are created based on the specialisation relation that is established here, the implementation object is annotated with information on which specialised targets are members of the same specialisation branches (cfr. figure 5.6



```
| bridge. children |
(checkTargets: specializationTargets)
       ifTrue:[
            bridge: (specializationFrameMorph copy).
            eerView addMorph: bridge.

            (specializationMorph copyIntoWorld: world
               FromMorph: self Offset: pointerPoint
                  ToMorph: bridge Offset: 0 Width: 2).

             specializationTargets do:[|:t|
                drawPointerForSpecialization: bridge To: t Evt: event].

            children: (vector copy).

            specializationTargets do:[|:t| children:(children copyAddLast:t )].

            specializationSets: ( specializationSets copyAddLast: children).

            specializationTargets: (vector copy)]

          False: userQuery reportAndContinue: 'First select specialized targets!']
```

Figure 5.6: The implementation for creating a new specialisation relationship in SelfSync.

(c)). For example if an entity `fruit` is specialised into `apple` and `pear` in one specialisation relation then the set (`apple`, `pear`) will become member of the `specialisationSets` of `fruit`. Based on the members of `specialisationSets` the Interactive Prototyping process will generate interactions to decide in which object a new run-time `fruit` population object will be specialised at creation time. This kind of annotating is the default manner how changes in the EER view affect future population objects. If the annotations happen in parent objects, existing population objects can also be affected.

d. It is enforced that specialised targets are already selected (cfr. figure 5.6 (d)).

e. Finally the targets are "released" (cfr. figure 5.6 (e)).

5.3.3 Changes in the Implementation View

There exist three ways to change SelfSync implementation objects:

- Menu changes via the Self menu: slots are added to an implementation object or a new child object is created from an implementation object.

- UI changes via the Self outliners: slot names and references in implementation objects are manually changed via the outliners in the UI.

- Literal changes via assignments or meta-programming: references in implementation objects are changed by evaluating code for assignments or by manipulating mirror objects.

Before continuing we need to explain the way we implement the new behaviour that is responsible for synchronizing the implementation view with the EER view. The implementation for the synchronisation from the EER view to the implementation view, explained in section 5.3.2 gave rise to a new kind of Self object that participates in the SelfSync ARTE process. The strong and weak entity prototypes are implemented as specific empty Self objects with an additional EER outliner, a number of hidden categories, and their own specific SelfSync traits objects that override the more general behaviour of Self objects such as the methods in `traits clonable`.

In this case however we extend already existing methods with a test that checks whether the receiver participates in ARTE, and with the appropriate new behaviour. The reason for this decision is that the changes in the implementation view are intercepted at three different levels in the Self language. A more appropriate object-oriented approach that applies inheritance and overrides the original behaviour with the new behaviour is definitely part of our future work. This would result in a new kind of outliner to view implementation objects, a new kind of mirror object to reflect on objects in the ARTE process and a new kind of model object for Self slots that are contained in implementation objects.

We describe the implementation of the synchronisation in the EER view in the three cases summarised above.

Menu Changes

The first way to manipulate implementation objects in a certain schema namespace employs Self's object menu that is illustrated in figure 5.7. We extended this menu at the bottom with two new actions to add new slots. The motivation for this extension is that the original actions still allow to add information in implementation objects that is not propagated to the entity view.

When manipulating implementation objects via the Self menu, the default order of actions is as follows:

1. Middle-click on an implementation object in the implementation view. This results in the Self menu (see figure 5.7) being dynamically built for this implementation object. Next the menu appears in the implementation view.

2. Select an action in the Self menu. The actions that initiate a synchronisation process in the EER view are "subclassing"[4] the object, adding a data slot and adding a method slot. The methods

[4] This refers to the state inheritance between prototypes with the copy down mechanism that simulates subclassing.

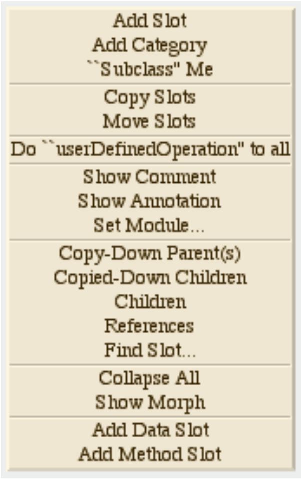


Figure 5.7: The extended Self menu for implementation objects in SelfSync.

corresponding to these actions already exist in Self and are extended with synchronisation behaviour.

3. The called method is executed and when necessary, synchronisation steps are automatically realised in the EER view. In order to illustrate this we elaborate on the "subclassing" of an implementation object. Selecting this action eventually results in sending the `createSubclass` method to the implementation object. The code for this method is illustrated in figure 5.8. The rectangles mark the added behaviour.

The original `createSubclass` method has been extended with a test to check whether the implementation object is part of the ARTE process in SelfSync. If the subclassed object does not participate in the ARTE process the original behaviour is called in the `False:` branch of the `createSubclass` method. Otherwise five actions are undertaken:

a. First a new entity is created with the name `a_Subclass_Of_....` This new entity is then visualised in the current EER view (see figure 5.8 (a)).

b. Second this new entity or implementation object is added in the current schema where it is saved persistently (see figure 5.8 (b)).

c. Next the specialisation relationship is drawn in the EER view between the appropriate entities (see figure 5.8 (c)).

d. Next the inheritance relationship is actually established between the implementation objects

```
   | bridge. copySelector. newChild. newTraits. parentsToOmit. pname = 'parent' |
   newChild: copyRemoveAll isComplete: false.

   (reflectee isEntity) ifTrue: [
   newChild: (reflect:(strongEntity new: ('a_Subclass_Of_', (reflectee label)))).
a. (reflectee world) addMorph: (newChild reflectee).

b. newChild reflectee install.

   bridge: (specializationFrameMorph copy).
   (reflectee world) addMorph: bridge.

c. (specializationMorph copyIntoWorld: (reflectee world)
       FromMorph: reflectee Offset: (reflectee pointerPoint)
       ToMorph: bridge Offset: 0 Width: 2).

d. reflectee drawPointerForSpecialization: bridge To: (newChild reflectee).

   reflectee specializationSets: (reflectee specializationSets copyAddLast: (vector
e. copyAddLast: (newChild reflectee))).
   newChild]

   False:[
   newTraits: reflect: () _Clone.
   parentsToOmit:  asSet copyFilteredBy: [|:s| s isParent ].

   copySelector:
    copyDowns isEmpty
     ifFalse: [copyDowns first selector]
        True: [|cdc. sels. selCounts|
               cdc: browseWellKnown copyDownChildrenOfReflectee: self Limit: 1.
               sels: cdc asVector copyMappedBy: [|:m| m copyDowns first selector].
               sels at: 0 IfAbsent: 'copy'
     ].

   copySelector: userQueryMorph askString: 'Copydown selector?'
                             DefaultAnswer: copySelector
                                     Event: process this birthEvent.
   newChild copyDownFrom: self Sending: copySelector Omitting: parentsToOmit.

   do: [|:s|
     s isParent ifTrue: [
       newTraits addSlot: s copyHolderForModule: ''
     ]
   ].
```

Figure 5.8: The implementation for creating a new copy down child of a prototype in SelfSync.

(see figure 5.8 (d)). Remark that it is the same `drawPointerForSpecialisation:To:` method of figure 5.6 (b) that is used here. This method actually extends the original behaviour we are currently overwriting with actions in the EER view such as drawing links.

e. Finally the implementation object is annotated with a specialisation set that contains the new subclassed implementation object (see figure 5.8 (e)).

UI Changes

The second way to manipulate implementation objects in a certain schema is via its outliner. One can change references, i.e. the object's slots and their contents via the Self UI. First the name of a slot can be changed in an outliner. Second a pointer from an implementation object to another can be dragged and dropped onto another implementation object. Since these are default Self actions we overwrote the associated behaviour for objects involved in ARTE, while the original behaviour was maintained for other Self objects.

When a slot's name is changed manually as illustrated in figure 5.9 this eventually results in the message `acceptChangingNameTo:Editor:Event` being sent to `selfSlotModel parent`. The latter is the object that defines the behaviour of the underlying Self slots (the models) that are viewed with the Self outliners. The code for this method is illustrated in figure 5.10. The rectangles denote the added behaviour. First the default behaviour is invoked to change the name of the slot. Next a test makes a distinction between method slots (see figure 5.10 (a)) and data slots (see

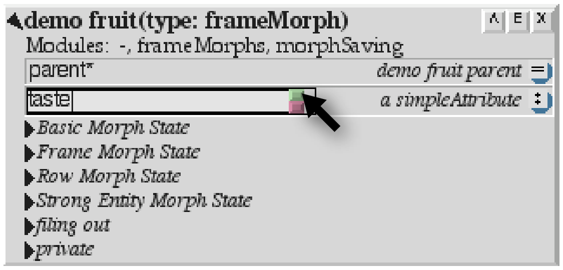
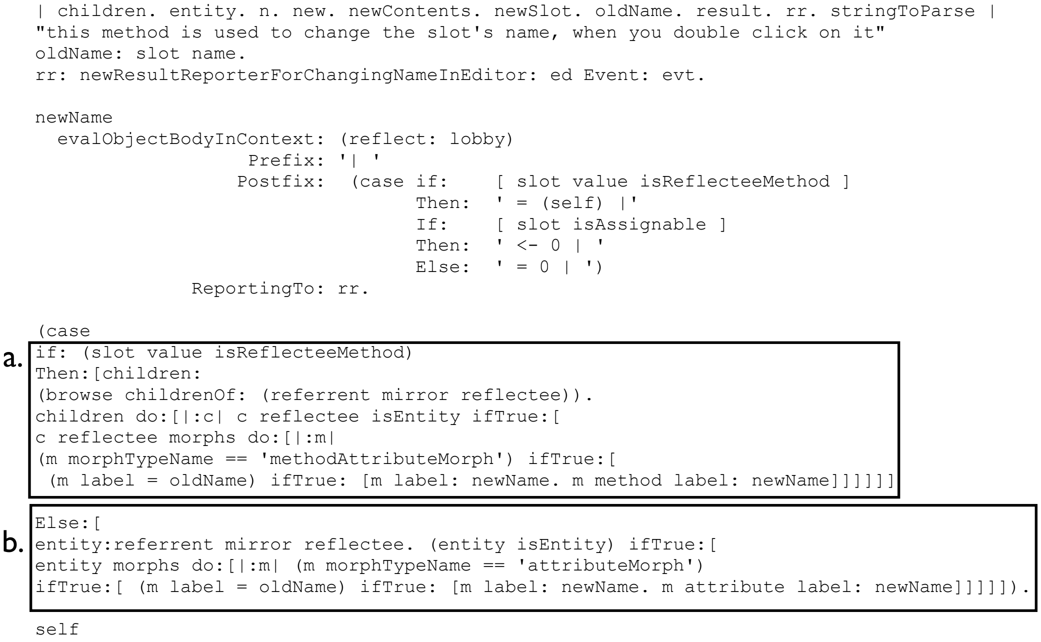

demo fruit(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent* demo fruit parent
taste a simpleAttribute

Figure 5.9: Manually changing a slot's name in Self.



```
| children. entity. n. new. newContents. newSlot. oldName. result. rr. stringToParse |
"this method is used to change the slot's name, when you double click on it"
oldName: slot name.
rr: newResultReporterForChangingNameInEditor: ed Event: evt.

newName
  evalObjectBodyInContext: (reflect: lobby)
                 Prefix: '| '
                Postfix:  (case if:    [ slot value isReflecteeMethod ]
                                Then:  ' = (self) |'
                                If:    [ slot isAssignable ]
                                Then:  ' <- 0 | '
                                Else:  ' = 0 | ')
            ReportingTo: rr.

(case
if: (slot value isReflecteeMethod)
Then:[children:
(browse childrenOf: (referrent mirror reflectee)).
children do:[|:c| c reflectee isEntity ifTrue:[
c reflectee morphs do:[|:m|
(m morphTypeName == 'methodAttributeMorph') ifTrue:[
 (m label = oldName) ifTrue: [m label: newName. m method label: newName]]]]]]
Else:[
entity:referrent mirror reflectee. (entity isEntity) ifTrue:[
entity morphs do:[|:m| (m morphTypeName == 'attributeMorph')
ifTrue:[ (m label = oldName) ifTrue: [m label: newName. m attribute label: newName]]]]]).
self
```

Figure 5.10: The implementation for changing a slot's name in SelfSync.

figure 5.10 (b)). In both cases it is first tested whether the object that includes the changed slot is an ARTE implementation object. If so and in the case of a data slot the name change is propagated to the appropriate attribute in the corresponding entity. What happens in the case of method slots is explained in chapter 6.

When a pointer is dropped onto another object (see figure 5.11) the contents of that slot is changed to include this object. Self intercepts these kinds of changes via the UI's MVC mechanism. More precisely, by dragging and dropping a pointer the view on this implementation object is manually changed. This is automatically propagated to the underlying model of this implementation object, where the actual slot change is realised. Next this change is updated in the view of the implementation object where the slots now contains the new object. This updating is implemented the `updateDo:` method of the `generalCategoryModel slotsUpdater parent` object. We added a call to the `checkSlotContents` method of `Traits Entity`, that is partially illustrated in figure 5.12. The entire method can be found in Appendix A in section A.1. This `checkSlotContents` method ensures that when one implementation object references another implementation object, this is automatically synhronised in the EER view by drawing a relationship link between the two corresponding entities. Moreover the actual number of referenced objects should

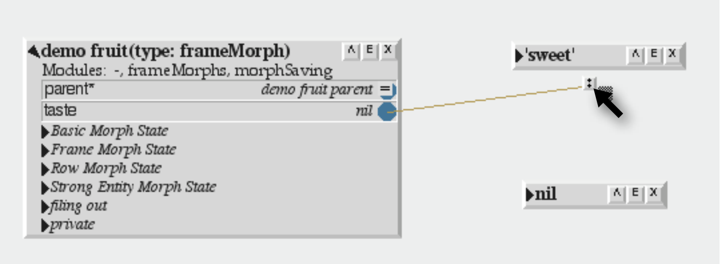


Figure 5.11: Manually changing a slot's contents in Self.

```
"check whether there exist already a link between the two entities"
(self morphs) do:[|:m|(m morphTypeName == 'relationFrameMorph')

   ifTrue:[((m relationName = (relLabel1 myLabel))
            || (m relationName = (relLabel2 myLabel)))
        ifTrue:[check
          ifTrue:[
                 "update existing cardinalities"
                  m morphs do:[|:m|(m morphTypeName == 'pointerTailMorph')
                    ifTrue:[(m headMorph rawOwner setCardinality: num)]].
                  m morphs do:[|:m|(m morphTypeName == 'pointerHeadMorph')
                     ifTrue:[(m tailMorph rawOwner setCardinality: num)]].
```

Figure 5.12: An implementation excerpt of the `checkSlotContents` method in SelfSync.

be reflected in the multiplicity of that relationship. In the code excerpt in figure 5.12 it is tested whether there is already a link drawn between the two entities. If so the multiplicity is updated on the correct side of the relationship link in the EER view.

Literal Changes

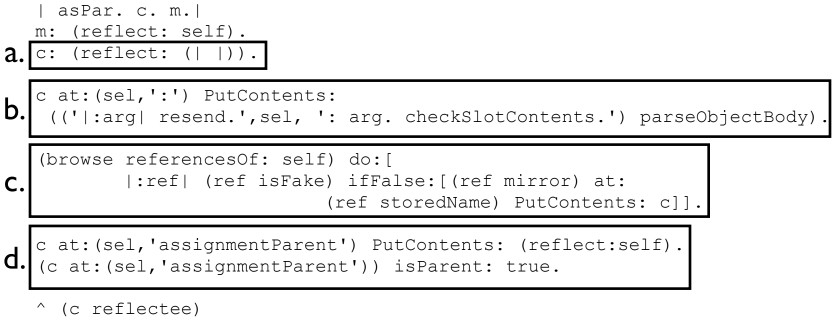
Here code is written literally to change an object's slots. This can involve calling a certain method or by meta-programming with mirror objects. For example, `fruit taste:'sweet'` corresponds to the meta-variant `fruit asMirror at:'taste' PutContents:(reflect:'sweet')`. Both statements change the contents of the slot `taste` in the `fruit` prototype to the string `'sweet'`.

For meta-programming statements the interception is easy. We extend the `at:PutContents: ifFail` method that is sent to a mirror object with a call to the `checkSlotContents` method for the reflected implementation object.

For assignments the situation is more complicated. In Self a corresponding assignment slot is added transparently for each assignable data slot in an object. This assignment slot contains the *assignment primitive* (←) for which the compiler generates the appropriate code. This implies that it is – to the best of our knowledge – assignments cannot be intercepted by changing the assignment slot directly via meta-programming.

SelfSync intercepts assignments via *assignment shadowing*. Between the object containing the assignment and all its referring objects an object is dynamically mixed in that contains a shadowing method that overrides the assignment. In the shadowing method a `resend` (super) is called with the original assignment, followed by a call to the `checkSlotContents` method.

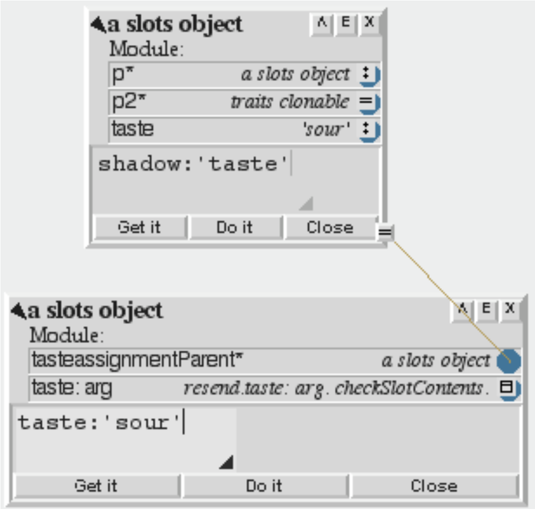
The code for a shadow method is illustrated in figure 5.13.



```
| asPar. c. m.|
m: (reflect: self).
c: (reflect: (| |)).

c at:(sel,':') PutContents:
 (('|:arg| resend.',sel, ': arg. checkSlotContents.') parseObjectBody).

(browse referencesOf: self) do:[
        |:ref| (ref isFake) ifFalse:[(ref mirror) at:
                           (ref storedName) PutContents: c]].

c at:(sel,'assignmentParent') PutContents: (reflect:self).
(c at:(sel,'assignmentParent')) isParent: true.

^ (c reflectee)
```

Figure 5.13: The implementation for assignment shadowing in SelfSync.

a. First a new empty object is created (see figure 5.13 (a)) upon which is reflected.



p* a slots object
p2* traits clonable
taste 'sour'

```
shadow:'taste'
```

tasteassignmentParent* a slots object
taste: arg resend.taste: arg. checkSlotContents.

```
taste:'sour'
```

Figure 5.14: After assignment shadowing in SelfSync.

b. Next a new method is added in this object that merges a super call to the original assignment with a call to the `checkSlotContents` method (see figure 5.13 (b)).

c. Next all references to the original object containing the assignment are replaced by a reference to the new object (see figure 5.13 (c)).

d. Finally a slot is added in the new object containing the old object. This slot is then declared to be a parent slot (see figure 5.13 (d)).

In figure 5.14 the `taste:` assignment has been shadowed.

5.4 The Interactive Prototyping Implementation

The Interactive Prototyping of implementation objects is implemented with the help of the `copy` method in `Traits entityMorphs`. This method overrides the more general `copy` method that is understood by all clonable objects.

The implementation of `copy` is depicted in figure 5.15. First a clone is made of the receiver of the `copy` method via a super call. Next interactions are generated for the aggregations (with `resolveAggregates:`), the categorisations (with `resolveCategorisations:`), the specialisations (with `resolveSpecialisations:`), and the roles (with `resolveRoles:`) this object's entity is involved in the EER view. Similarly interactions are generated for the relationships that originate from the new object with multiplicity 1 (with `resolveOneConstraints:`) and with

```
| new |
new: resend.copy.
"take a plain copy and next resolve all constraints"
resolveAggregates:new.
resolveCategorizations: new.
resolveSpecializations: new.
resolveRoles: new.
new: (resolveOneConstraints: new).
new: (resolveNConstraints: new).

new isPopulationMember: true.

currentPop addMember: new.
currentPop save.

new
```

Figure 5.15: The implementation of the `copy` method in SelfSync.

multiplicity n (with `resolveNConstraints:`). These auxiliary methods heavily depend on the annotations that were added during Active Modelling as described in the fourth step of the list of synchronisation actions in section 5.3.2. Next the new run-time object is declared to be a population object and is installed in the current population.

In figure 5.16 a part of the code of the `resolveCategorisations:` method is illustrated. In this code excerpt an interaction is generated for each annotated categorisation branch. This interaction shows an option window to let the new object inherit from one of the categorisation parents, at creation time. At the moment of selection, the selected categorisation parent's prototype is "subclassed" in the new run-time object. A parent slot is also dynamically added in the new object, that contains the parent of the selected categorisation parent. The complete implementation of all auxiliary methods can be found in Appendix A in section A.2.

```
catPar: userQuery askMultipleChoice: 'This new ',
((self label) uncapitalizeAll),
 ' object can be categorized in one of the following parents: '
                          Choices: categorizationNames
                          Results: categorizationParents.
(reflect: catPar) createSubclassIn: new.
(reflect:new) at: (((catPar label),'Parent') uncapitalizeAll)
PutContents: (reflect: (catPar parent)).
((reflect:new) at: (((catPar label),'Parent')uncapitalizeAll))
isParent: true.].
```

Figure 5.16: A code excerpt of `copy`'s auxiliary method to resolve categorisations in SelfSync.

5.5 Constraint Enforcement Implementation

The part of the SelfSync implementation that deals with constraint enforcement for multiplicities and dependencies is gathered in `Traits entity` as illustrated in figure 5.5.

5.5.1 Multiplicities

To implement Multiplicity Constraint Enforcement SelfSync applies a `checkConstraints` method that is partially listed in figure 5.17. This code excerpt deals with relationships that originate from the checked population object that initiated the checking mechanism and that have a multiplicity n, strictly larger than one, for the partner entities.

```
"For each (1/n)-to-N relation I can point to at most N partner"

constrN do:[|:c|((names1 occurrencesOf:
        (c at: 0)) asInteger > ((c at:1) asInteger)) ifTrue:
            [userQuery show:
                'Constraint of x-to-',((c at: 1) asString), ' relation between the',
                (self label uncapitalizeAll capitalize),' and ',
                 ((c at:0) capitalize),' objects has been violated!
                 I am pointing to more than ',((c at: 1) asString),' ',((c at: 0)
                 capitalize),' objects...' ForSecs:1]]].

"For each (1/n)-to-N relation at most N partners can point to me"

constrN do:[|:c|((names2 occurrencesOf:
        (c at: 0)) asInteger > ((c at:1) asInteger)) ifTrue:
            [userQuery show:
                'Constraint of x-to-',((c at: 1) asString), ' relation between the',
                (self label uncapitalizeAll capitalize),' and ',((c at: 0) capitalize),'
                objects has been violated!   More than ',((c at: 1) asString),' ' ,
                ((c at: 0) capitalize),' objects are pointing at me...' ForSecs:1]].
```

Figure 5.17: A code excerpt of the `checkConstraints` method to enforce multiplicities in SelfSync.

During Active Modelling the names of the partner implementation objects that constrain an implementation object are annotated in the latter object's private slots, more precisely in the `constraintByOne` and `constraintByN` slots. For example if in the EER view a `mother` entity is in a one-to-3 relationship with a `child` entity, then the pair `('child',3)` will automatically become a member of the vector in the `constraintByN` slot in the `mother` implementation object. Moreover `'mother'` will automatically become a member of the vector in the `constraintByOne` slot in the `child` implementation object.

In the code excerpt in figure 5.17 the methods `constrOne` and `constrN` implemented in `traits entityMorphs`, dynamically access the names of these constraining objects, in the current population object. In the first part it is checked whether this population object references at most n other population objects of the constraining kind. This is achieved by gathering in `names1` the names of the implementations object from which the currently referenced population objects were cloned. If the name of a constrained implementation object of `constrOne` and `constrN` occurs more than the allowed multiplicity n times in the `names1` collection, a warning is generated.

In the second part the above process is repeated but for the names of the implementation objects of the population objects that reference the checked population object themselves. The entire implementation of the `checkConstraints` method can be found in Appendix A in section A.3.1.

5.5.2 Object Dependency

The enforced creation behaviour of Object Dependency is implemented in SelfSync in the `resolveOneConstraints:` and `resolveNConstraints:` auxiliary methods used in the `copy` method discussed above. The complete code can be found in Appendix A in section A.2. We list the relevant piece of code in figure 5.18. This particular code is called when the actual number of references in a population object is interactively set to zero during Interactive Prototyping when resolving relationships in the `copy` method. In this case when the new population object is a dependent weak object, a warning is generated that at least one actual reference to a strong population object has to be established. Next the `copy` method is re-initiated.

```
isStrongEntity
 ifFalse:[
     (entityOwner == s)
        ifTrue:[userQuery report:'This new ',
           (((new label)uncapitalizeAll) capitalize),
           ' object is dependent to this ',
           (s label uncapitalizeAll capitalize),' object:
           establish at least one association!'.
           ^copy]
        False:[(reflect:new) at: slotName PutContents:(reflect: nil).
                (reflect:new) removeSlot:(s label)]]
```

Figure 5.18: Excerpt of code to enforce Object Dependency at creation time in SelfSync.

The cascading delete behaviour of Object Dependency is implemented in SelfSync with the help of the `deleteMe` method in `Traits entity`, that is listed in figure 5.19. In part (a) of figure 5.19 each slot[5] of the population object that received the `deleteMe` method is checked to contain dependent population objects. In the case of a collection, the checking mechanism iterates upon all its elements. If a referenced object is a dependent population object, deletion is realized by removing all real slots of that object.

In part (b) of figure 5.19 finally the appropriate slots of the population object that received the `deleteMe` method are deleted. The entire implementation of the `deleteMe` method can be found in Appendix A in section A.3.2.

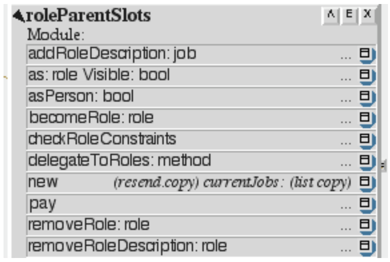
[5] Public slots that are not gathered in any category.

```
     dependents do:[|:d|
       (reflect:self) do:[|:s|
         ((((s category == 'private')  ||
          ...
         (s category == 'Frame Morph State')
         || (s isParent)) || (s isMethod)) ||(s isAssignment))
           ifFalse:[(((reflect:(s contents reflectee)) creatorSlotHint name) == 'vector')
                   ifTrue:[(s contents reflectee) do:[|:e|
a.                           (((reflect:d) creatorSlotHint name) ==
                             ((reflect:e) creatorSlotHint name))
                              ifTrue:[e deleteMe] False:['']]]

                               False:[(((reflect:(d )) creatorSlotHint name) ==
                             ((reflect:(s contents reflectee)) creatorSlotHint name))
                              ifTrue:[(s contents reflectee) deleteMe]]]]]

             False:[(entityOwner isNil)
b.                 ifFalse:[(entityOwner) dependents:
                       ((entityOwner dependents) remove: (self label))]].
     (reflect:self) do:[|:s|
         ((s category == 'private')  ||
          ...
         (s category == 'Frame Morph State'))
      ifFalse:[(reflect: self) removeSlot: (s name)]].
```

Figure 5.19: The partial implementation of the `deleteMe` method to enforce Object Dependency in SelfSync.

5.6 Role Modelling Implementation

For Role Modelling SelfSync supports the RTE scenario for roles, that was described in section 4.8.8. We will focus on the traits object for general role behaviour called `roleParentSlots`, cfr. figure 5.20.



roleParentSlots
Module:
addRoleDescription: job ...
as: role Visible: bool ...
asPerson: bool ...
becomeRole: role ...
checkRoleConstraints ...
delegateToRoles: method ...
new *(resend.copy) currentJobs: (list copy)*
pay ...
removeRole: role ...
removeRoleDescription: role ...

Figure 5.20: The parent object that gathers general role behaviour in SelfSync.

We discuss the methods to dynamically create or remove a role in an object and the method that checks role combinations.

5.6.1 Creating a New Role in SelfSync

In figure 5.21 the code for the `becomeRole:` method is listed. In this method the *warped inheritance hierarchy pattern* (cfr. section 4.4) is constructed. In the first statement the population object that

```
addRoleDescription: (role description).

(reflect: role) createSubclassIn: self.

(reflect: self) at: ((role description),'parent')
                PutContents: (reflect:(role parent)).

((reflect: self) at: ((role description),'parent')) isParent: true.

(reflect: self) removeSlot: 'parent' IfFail:''.
```

Figure 5.21: The behaviour for dynamically creating a role in SelfSync.

is the receiver of this method is annotated with a description of the new role. Next the reversed state inheritance between the prototype of the receiver and the prototype of the role is realised. The following two statements add the inheritance between the receiver and the traits of the role. The last statement removes the inheritance link to the receiver's own traits if still present, i.e. when this is the first role to be performed.

5.6.2 Removing a Role in SelfSync

In figure 5.22 the code for the `removeRole:` method is listed. In the first statement the annotations

```
removeRoleDescription: (role description).

(reflect: self) removeSubclassFrom: role.
(reflect: self) removeSlot: ((role description),'parent') IfFail:''.

((currentJobs size) == 0) ifTrue: [(reflect: self) at:
        'parent' PutContents:
             (reflect:(((reflect:self) creatorSlotHint) contents reflectee parent)).
((reflect: self) at: 'parent') isParent: true.].
```

Figure 5.22: The behaviour for dynamically removing a role in SelfSync.

for the role to be stopped performing are removed from the receiver. Next the reversed inheritance between the prototype of the receiver and that of the role are annihilated. In the next statement the inheritance link from the receiver to the role's traits is deleted. The following test checks whether the removed role was the only role performed by the receiver. If so the inheritance link to the receiver's own traits, that was removed upon performing its first role is established again.

5.6.3 Constraining Role Combinations in SelfSync

Finally in figure 5.23 the code for the `checkRoleConstraints` method is listed. This methods makes use of the `roleSets` attribute of the receiver. In this attribute that is inherited from the implementation object the receiver was cloned from, annotations were added during Active Mod-

```
currentJobs do:
  [|:r| roleSets do:
         [|:rs| rs do:[|:rse| ((rse description) = r)
                ifTrue:[rs do:
                      [|:or| ((or description) = r)
                      ifFalse: [(currentJobs includes:(or description))
                                 ifTrue:[error: r,'XOR',(or description),
                                                   ' constraint violated'.
                                         currentJobs:(currentJobs remove: r).]]]]]]]
```

Figure 5.23: The behaviour for dynamically checking role combinations in SelfSync.

elling to represent the permissible role combinations. Consider the manager-salesman-engineer example in figure 5.24.

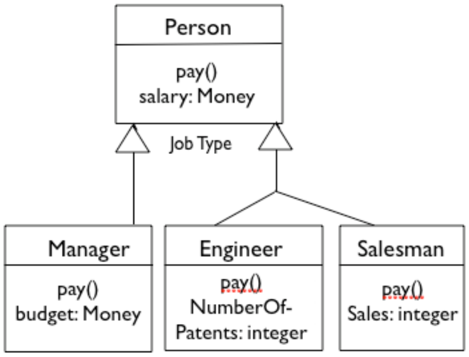


Figure 5.24: Role example.

When this setup is modelled in the EER view during Active Modelling, the `person` implementation object's `roleSets` collection contains two elements. The first one is the set `(manager)`, while the second one is the set `(salesman,engineer)`.

The `checkRoleConstraints` method iterates upon the current roles of the receiver, in `currentJobs` and checks them against the annotated combinations in the receiver's `roleSets`. If at least two of the current roles are members of the same role set an error is generated. The `checkRoleConstraints` method is called in the `becomeRole:` method when annotating the receiver with the new role's description.

5.7 Conclusion

SelfSync is a proof-of-concept implementation in Self, that supports our instance of ARTE. Its UI includes a graphical drawing editor for EER diagrams. Entities and implementation objects are represented by an EER outliner and a Self outliner that both view one underlying Self object. Self-

Sync objects are implemented with prototypes that merge these two views. A double traits inheritance hierarchy also illustrates the duality of the two views. In SelfSync the Active Modelling phase is fully automated and applies Optimal Transformations between the different views. In the EER view SelfSync uses a dynamically built entity menu whose actions result in annotations and/or synchronisation in the implementation view. Changes in the implementation view can be realised via the Self menu, via the Self UI, and literally via the code. The invoked synchronisation behaviour is scattered at different levels of the Self language: in the outliners, in the underlying slot models and in the mirrors at the meta-level. Run-time Object Generations are created during a semi-automated Interactive Prototyping phase. The prototyping behaviour is implemented by overriding the default cloning operation of Self with a new cloning method for SelfSync objects, that generates interactions depending on their annotations. Multiplicity Constraint Enforcement and Object Dependency is implemented with meta-programming constructs that compare implementation objects' references with their annotated objects. SelfSync implements Role Modelling with the help of the warped hierarchies pattern and supports a mechanism for enforcing role combination constraints.

Chapter 6

Extending ARTE with Advanced Method Synchronisation: Towards Executable Models and Visual AOP

In this chapter we extend our ARTE approach with *Advanced Method Synchronisation* (AMS). More precisely we consider operations in the data modelling view, that have their corresponding methods in the implementation view. Methods and their bodies are made visual in the data modeling view and are synchronized with the implementation and population views (see section 6.1).

We envision that AMS comes with two repercussions for ARTE. The first track we can follow is that AMS allows to evaluate operations in the context of the data model. This increases the feasibility to extend SelfSync with behaviour in the data model, that can be "run". In this way, we envisage that it is possible for EER models in SelfSync to become *Executable Models* (see section 6.2).

Second, AMS makes it possible to "inject" or remove Self code into operations, from inside the data modeling view. We foresee that these `before/after` constructs can be consolidated in SelfSync, in such a way that this extension can be considered as a first step in the direction of *Visual Aspect-Oriented Programming*.

6.1 Advanced Method Synchronisation in ARTE Tools

As a consequence of the Optimal Transformations (see section 2.5.2) and of the fact that data and methods both are represented with slots, SelfSync allows, next to atributes, to include *operations* in the entities of the graphical data modeling view, such as the operations in UML class diagrams. These operations are linked to the method bodies of the corresponding methods in the implementation objects.

We define Advanced Method Synchronisation (AMS) as the synchronisation between the operations in a data modeling view and methods in an implementation view. Method bodies can be edited in the data modeling view, which is to be automatically synchronized with the actual method bodies in the implementation view, and vice versa.

In the remainder of this section we define the language characteristics required to implement AMS (see section 6.1.1) and describe their presence in programming languages (see section 6.1.2). Next, we illustrate how the data modelling view in SelfSync is extended with operations (see section 6.1.3), followed by mappings for these operations in the other views (see section 6.1.4). Finally, we provide an overview of how AMS is implemented in SelfSync (see section 6.1.5.

6.1.1 Language Constructs for AMS

In this section we discuss the language constructs that are required to implement AMS.

The method synchronisation itself is similar to the synchronisation of attributes. The new language characteristics we need, are a representation for methods in the data modeling view, and mechanisms to convert the representations of methods from the implementation view to the data modeling view and vice versa.

Language Mechanism 6.1 ***A View for Methods*** *is a graphical representation in the data modeling view, that views a method and its body in the implementation view,*

Additionally, AMS requires meta-programming means to grab a method, to pretty-print a method body, etc. Via this link methods are viewed and edited in both directions. Therefore we introduce the following two language constructs.

Language Mechanism 6.2 ***A Method Reification Mechanism*** *makes methods of the implementation view visible in the data modelling view.*

The method body in the implementation view can be unparsed into a string, for example, in order to incorporate them into the data modelling view.

Language Mechanism 6.3 ***A Method Absorption Mechanism*** *implements the converse operation of the reification mechanism of definition 6.2.*

This mechanism is required because methods can be edited and saved from inside the data modelling view. After editing the new string representation of the method is parsed again into a method body, and added to the implementation view.

Similar to the synchronisation of attributes, AMS uses:

- The Bidirectional Connection, as introduced in definition 3.1, that can also implement a correspondence between an operation residing in the data modelling view and the body of the corresponding method in the implementation view.
- For the method synchronisation itself, again we use the Synchronisation Trigger Mechanism, introduced in definition 3.2.

6.1.2 Presence of AMS Constructs in Programming Languages

The new language constructs that are demanded to implement AMS in ARTE require:

- **Environment level constructs**: A View for Methods
- **Meta-level constructs**: A Method Reification Mechanism and a Method Absorption Mechanism

The presence of a View for Methods heavily depends on the User Interface of a programming language environment. Smalltalk and Self environments, for example, that employ an MVC UI, can easily implement this language construct.

As mentioned before in section 3.4.1, the poor reflective abilities of C++ implies that the Method Reification Mechanism and the Method Absorption Mechanism, are not available in C++.

Although Java has quite rich a reflection API, it does not allow a programmer to absorb new code into an existing Java program as defined by the Method Absorption Mechanism.

However, the Method Reification Mechanism and the Method Absorption Mechanism are present in Smalltalk, Ruby and Self:

- Smalltalk supports a Method Reification Mechanism and a Method Absorption Mechanism with respectively `sourceCodeAt:` and `compile:`.
- Ruby has support for a **Method Reification Mechanism** with run-time access to methods.

  A **Method Absorption Mechanism** is supported via an `eval` method (and its variations such as `class_eval, module_eval`, and `instance_eval`) that will parse and execute an arbitrary string of source code as a method.

- Self implements a **Method Reification Mechanism** via its mirrors with the method *mirroredObject* `at:`*selector* `contents sourceString`. A **Method Absorption Mechanism** in Self is based on the construct *body* `parseObjectBody` that parses a string into a method body. Adding it to an object in a new method slot is implemented with `at:PutContents:`.

6.1.3 An EER Model with Operations

In this section we describe how the EER view in SelfSync is extended to support AMS.

First, we extend SelfSync's drawing editor for EER models with a new modeling construct: **Operations** (see figure 6.1) that are similar to those in the UML class diagrams. The EER operations are represented by red rectangles that carry the name of the operation. Operations have their own specific menu that is dynamically built with the following actions, as illustrated in figure 6.2:

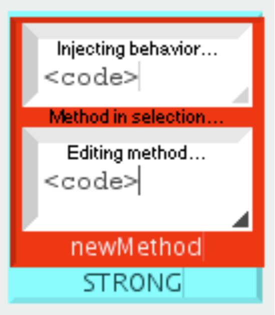


Figure 6.1: An operation in a strong entity.

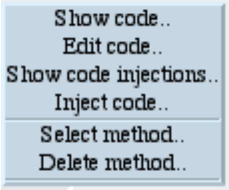


Figure 6.2: An operation menu.

- **Show code**: to open a window containing the code that is associated with this operation.
- **Edit code**: to open an editor window containing the code that is associated with this operation. The edited code can then be saved, in order to synchronise with the method's body.
- **Delete method**: to delete an operation in the data modeling view. This action is followed by the deletion of the corresponding method in the implementation view.

The remaining menu items are related to one of the two extensions based on AMS, i.e. Visual AOP, that we propose later. The menu items are discussed in section 6.3.2.

6.1.4 Mappings between Operations and Methods

In this section we discuss the mappings between operations in the EER view and the elements of the other views.

From the EER View to the Implementation View Operations in the EER view map to method slots in the related parent objects in the implementation view. The contents of operations, i.e. the method bodies in the EER view correspond to the contents of the method slots in the parent objects in the implementation view. This is a direct consequence of slots in Self.

From the Implementation View to the Population View When a method body is changed, this will affect certain run-time objects that compose a Run-time Object Generation, as explained in section 2.5.3. More precisely, all population objects that were cloned from the implementation object where the method change took place, will adhere to the new behavior.

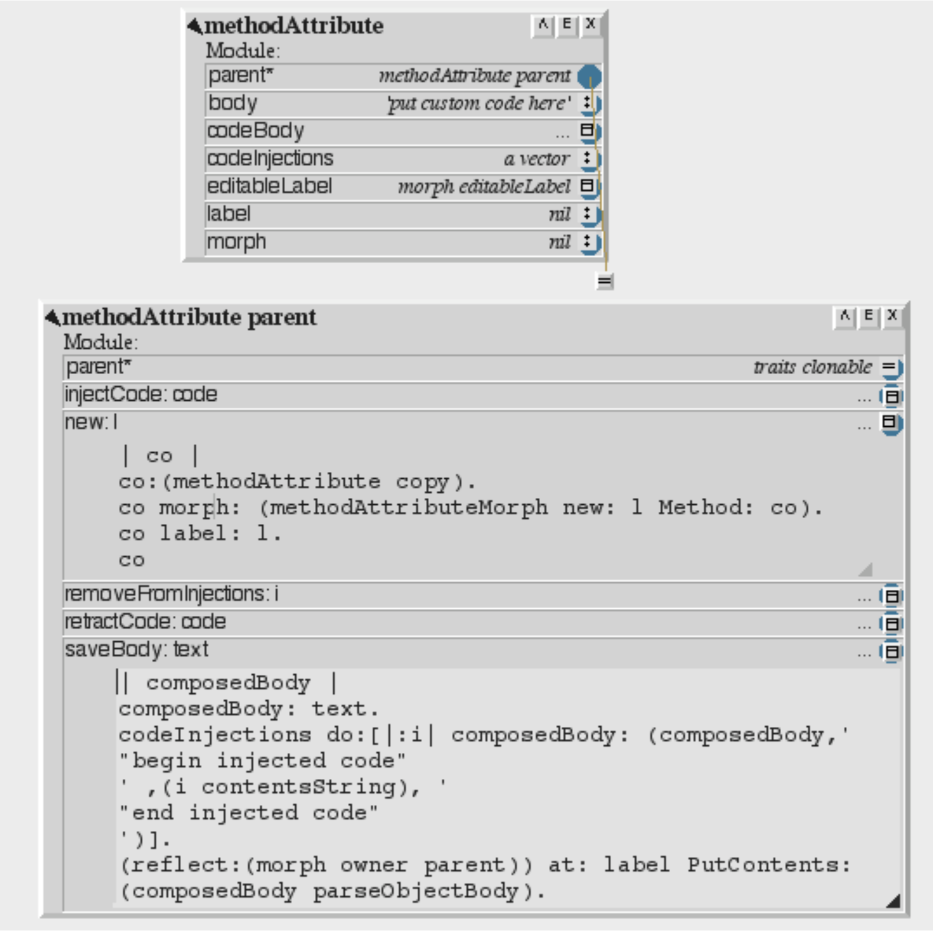
From the EER View to the Population View By adding, removing, renaming, and changing an operation to an entity view, all run-time population objects that are created from the entity view's code-time implementation object, are affected. This is a consequence of adding corresponding method slots in the 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.

6.1.5 SelfSync Implementation for AMS

In this section we briefly describe the implementation of AMS in the SelfSync ARTE tool.

Method Synchronisation In the EER View

The graphical representation of operations in the EER view is handled by the `methodAttributeMorph` prototype and its traits object (see figure 6.3). These objects deal with the graphical representation of operations and with window and menu management. Moreover they implement behaviour to synchronise operations and methods and to manage code injections, as explained in section 6.3.5.



methodAttribute
Module:

parent*	methodAttribute parent
body	'put custom code here'
codeBody	...
codeInjections	a vector
editableLabel	morph editableLabel
label	nil
morph	nil

methodAttribute parent
Module:

parent*	traits clonable
injectCode: code	...
new: l	...

```
| co |
co:(methodAttribute copy).
co morph: (methodAttributeMorph new: l Method: co).
co label: l.
co
```

removeFromInjections: i	...
retractCode: code	...
saveBody: text	...

```
|| composedBody |
composedBody: text.
codeInjections do:[|:i| composedBody: (composedBody,'
"begin injected code"
' ,(i contentsString), '
"end injected code"
')].
(reflect:(morph owner parent)) at: label PutContents:
(composedBody parseObjectBody).
```

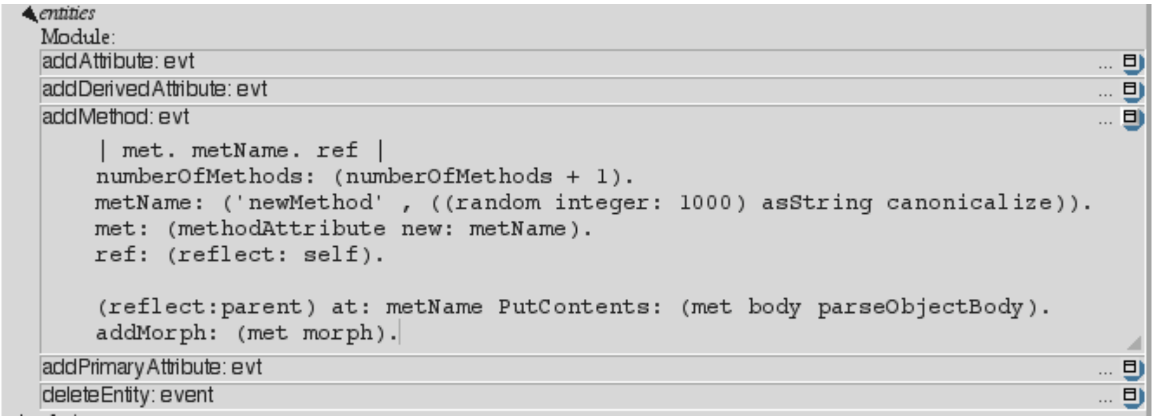
Figure 6.3: The `methodAttribute` prototype and its parent.

Adding a new operation in an entity in the EER diagram is realised with the dynamically built entity menu, more precisely with the `addMethod:` method. The code for this method is illustrated in figure 6.4. In this method, a new `methodAttribute` object with a random name[1] such as `method952` (see figure 6.3) is cloned.

This new `methodAttribute` object contains a default body `'put custom code here'`. A new method slot containing that body is added in the implementation object that corresponds to the entity where the operation is added. The graphical representation of the new operation is then added to the entity in question, with the help of a new `methodAttribute` clone.

When the name of the operation is changed in the EER view, it is saved with the `acceptChanges:Evt:` method for editable labels, that synchronises the new name in the corresponding method slot.

[1] Note that currently the names of operations do not include arguments. This is not a real problem since Self allows for an argument-less notation for methods.



```
entities
Module:
addAttribute: evt
addDerivedAttribute: evt
addMethod: evt
    | met. metName. ref |
    numberOfMethods: (numberOfMethods + 1).
    metName: ('newMethod' , ((random integer: 1000) asString canonicalize)).
    met: (methodAttribute new: metName).
    ref: (reflect: self).

    (reflect:parent) at: metName PutContents: (met body parseObjectBody).
    addMorph: (met morph).
addPrimaryAttribute: evt
deleteEntity: event
```

Figure 6.4: The `addMethod:` method in `traits entityMorphs` for adding an operation in the EER view.

The body of an operation is always stored in the `codeBody` data slot of the corresponding `methodAttribute`, that is made visual when consulted in the EER view.

When this operation's body is edited and saved in the EER view, the `saveBody:` method in `methodAttribute parent` performs the actual synchronisation to the method in the implementation view. If necessary, this saving process composes a new body from the previous body and its injections.

Method Synchronisation In the Implementation View

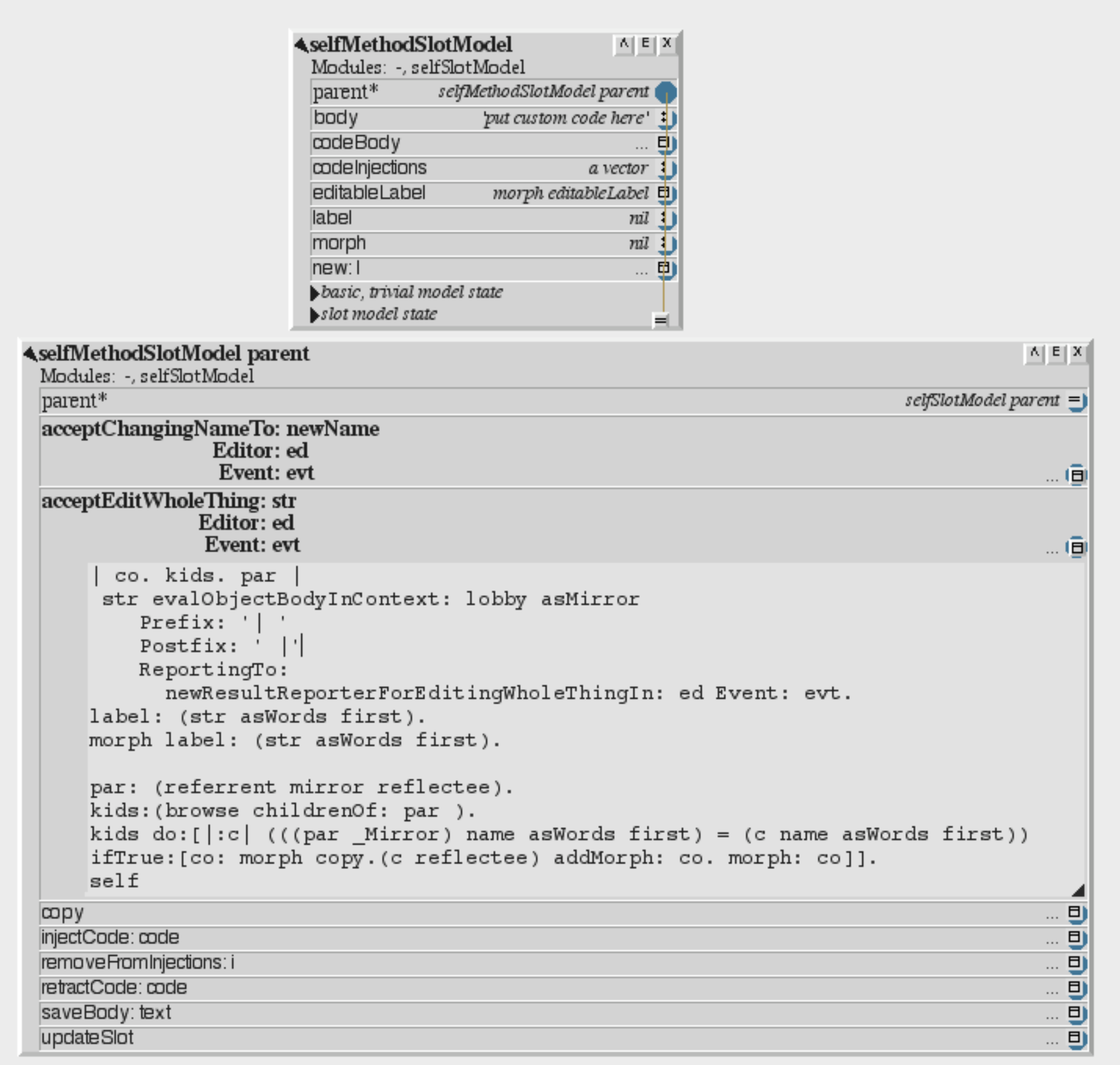
When a new method slot is added in an implementation object, the extended Self menu (see figure 5.7) allows to choose between propagation of this operation to the EER view, or to implement a method that is not visible in the EER view.

In the first case, Self's built-in mechanism to add method slots is reused. We however made a distinction between adding data slots and method slots since these actions require different kinds of synchronisation in the EER view.

Our solution involves the creation of a new prototype `selfMethodSlotModel` that is illustrated in figure 6.5. Each new method slot in an implementation object requires a clone of the `selfMethodSlotModel` prototype.

We provide the `selfMethodSlotModel parent` that is illustrated in figure 6.5, with a new `acceptEditWholeThing:Editor:Event:` method. This method extends the default Self behaviour to accept the code for a new method slot, with the actual synchronisation in the EER view.

By default we add new methods in the parent part of implementation objects. A part of the new



selfMethodSlotModel
Modules: -, selfSlotModel

parent*	*selfMethodSlotModel parent*
body	*'put custom code here'*
codeBody	...
codeInjections	*a vector*
editableLabel	*morph editableLabel*
label	*nil*
morph	*nil*
new: i	...

basic, trivial model state
slot model state

selfMethodSlotModel parent
Modules: -, selfSlotModel

parent*	*selfSlotModel parent*

acceptChangingNameTo: newName Editor: ed Event: evt ...

acceptEditWholeThing: str Editor: ed Event: evt ...

```
| co. kids. par |
 str evalObjectBodyInContext: lobby asMirror
    Prefix: '| '
    Postfix: ' |'
    ReportingTo:
      newResultReporterForEditingWholeThingIn: ed Event: evt.
label: (str asWords first).
morph label: (str asWords first).

par: (referrent mirror reflectee).
kids:(browse childrenOf: par ).
kids do:[|:c| (((par _Mirror) name asWords first) = (c name asWords first))
ifTrue:[co: morph copy.(c reflectee) addMorph: co. morph: co]].
self
```

copy ...
injectCode: code ...
removeFromInjections: i ...
retractCode: code ...
saveBody: text ...
updateSlot ...

Figure 6.5: The selfMethodSlotModel prototype and its parent object.

code (the last five lines) in the `acceptEditWholeThing:Editor:Event:` method are responsible for tracking the prototypes that inherit from the changed parent implementation object. It is in these prototypes' visual entity view that the synchronisation is realised, by adding a graphical representation for the operation.

When a method body is changed in an implementation object, this becomes visible in the EER view at the moment that the body of an operation is consulted with the operation menu. Therefore we do not need to implement any additional synchronisation behaviour.

The `acceptChangingNameTo:Editor:Event` method propagates the renaming of a method slot in the implementation view to the name of the operation in the EER view.

6.2 Towards Executable SelfSync Models

In this section we first introduce our approach to Executable Models in the context of the ARTE approach. Next, we show an embryonical example of how operations in a SelfSync EER model can be evaluated and run, in order to illustrate that it is feasible to construct Executable ARTE Models. Finally, an overview of a corresponding implementation in SelfSync is provided.

6.2.1 Executable Models

One of the direct consequences of implementing AMS, is that behaviour becomes visible in an abstract data model. This allows us to envisage that SelfSync EER models are suitable to become executable. An *Executable Model* [20, 88, 104] is a model that acts like code and can be "run", but is defined at a higher level than the implementation language. An Executable Model is a working prototype. Running an Executable Model can be considered as testing it and results in accepting the model as it is or in (semi-automatically) extending and refining it.

Whereas Model-Driven Development (MDD) focuses on abstract models from which actual software is derived semi-automatically, Agile Development aims at delivering small "slices" of working code as soon as possible. Agile MDD combines the tools that are used in MDD with the Agile Development process. On the one hand, Agile MDD can be troubled by the fact that a model being actually "correct" or satisfying, becomes only clear when the derived code is both compiled and run. On the other hand, working code fragments still contain concepts, such as vectors, that are of no use for customers. Executable Models can offer a solution to both problems.

The main reason why AMS can be implemented in a language such as Self, is that, at compile-time, it is merely the syntactical correctness of method bodies that is checked. On the one hand, this implies that the overall correctness of a method's body becomes only clear at call-time. Unfortunately, this approach of evaluation allows to write nonsensical operation bodies in the data modelling view. On the other hand, Agile MDD advocates intensive customer collaboration, especially at the modelling level. Customers can be involved in the EER model storming, as they are often familiar with the domain model. Therefore it is plausible to directly "implement" some use cases in the EER model, in terms of that EER model.

In order to do so, we need a mechanism to evaluate the correctness of the implemented use cases in the context of the EER model. An *Entity checker* extends Self's method evaluation at compile-time, with an additional partial evaluation technique [60] that checks whether added operation are syntactically correct and will "run" in the context of the EER model. In the remainder of this section, we describe how ARTE EER models can be considered as Executable Models. More precisely, it is possible to evaluate an operation in the EER view in the context of the entity it is part of.

In Executable SelfSync Models, edited operations are evaluated based on the presence of attributes, operations and relationships in the entity it is part of.

Before saving an operation's body with `saveBody:` (see figure 6.3) we invoke an evaluator on the operation's expressions, in the context of the entity that contains the operation.

The evaluator for operations checks whether the expressions in the code statements of the operation are present in the entity as:

- An attribute
- An operation
- A role[2] in a relationship to another entity

Crucial in our notion of Executable Models is that, if an expression is not present, it is created interactively as one of the above items. The evaluation process requires interactions to decide to add the missing expressions as an attribute, as an operation or as a relationship.

6.2.2 Example of an Executable SelfSync Model

We illustrate how Executable SelfSync Models can work with a specific example that is illustrated in figure 6.6.

When the code for `sell` is saved, the evaluation proceeds correctly since `vendor` is in a relationship with `supplier` that has the role `supplier` in that relationship. Moreover, `supplier` already contains an operation `supply`.

When, however, we save the code for operation `supply` in `supplier`, the evaluator is interrupted twice. The first time the evaluator observes that `supplier` has no attribute or operation with the name `factory` nor a relationship to another entity that has the role `factory` in this relationship. In this case, and since it does not exist yet in the EER view, a new entity `factory` is added to the EER view interactively. Next a new one-to-one relationship is established between `factory` and `supplier` with the appropriate roles. This new setup is depicted in figure 6.7.

[2] The named place in the relationship, not the role modeling concept.

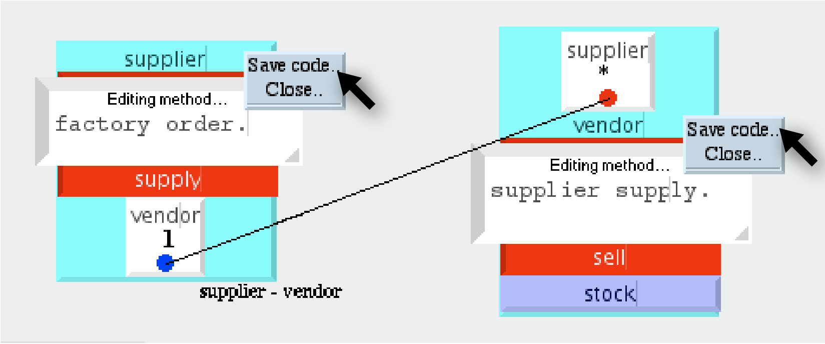


Figure 6.6: Saving code for a valid `sell` operation (right) and for an invalid `supply` operation (left).

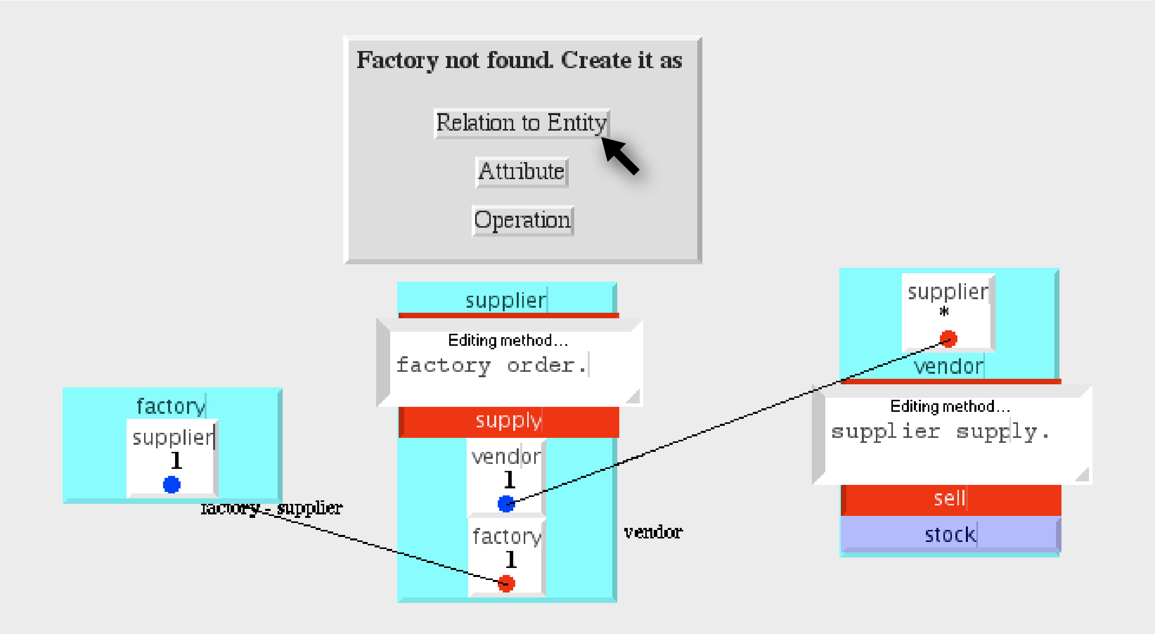


Figure 6.7: Adding a new relationship to a new `factory` entity.

Next, `order` is evaluated in the new entity `factory`. Since `order` is not part of the entity as attribute, method or relationship, in this case, `order` is added as a new operation, cfr. figure 6.8.

The two actions for adding `factory` and its operation `order` in the EER view, are synchronised in the implementation view.

6.2.3 Implementation Scheme of an Operation Evaluator

In this section we informally discuss how an evaluator for operations in SelfSync Executable Models can be implemented.

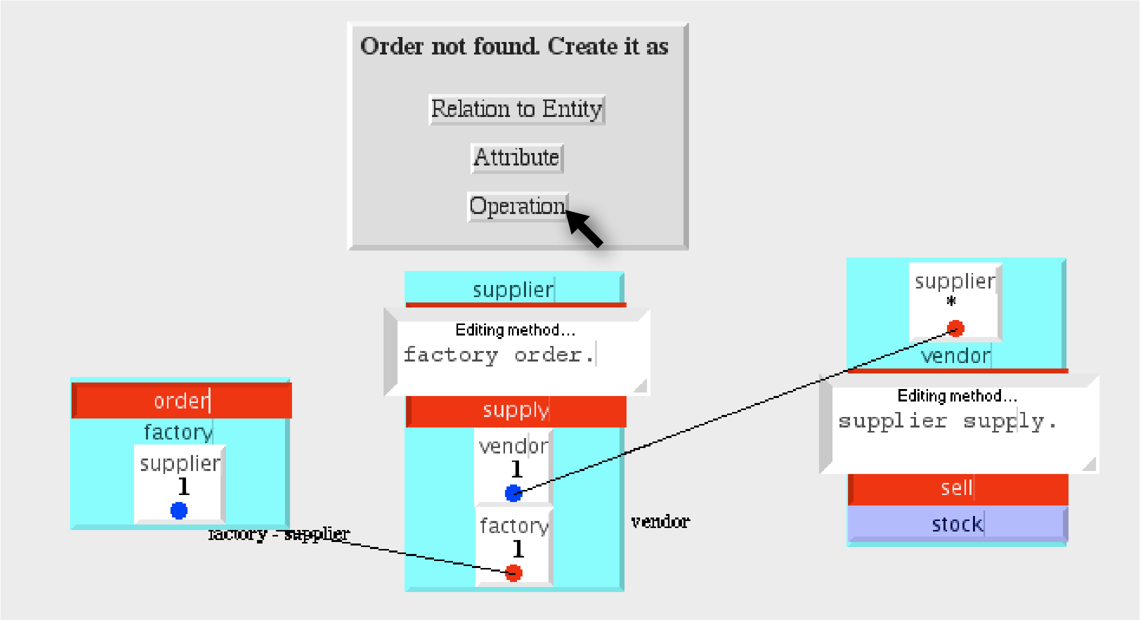


Figure 6.8: Adding a new operation `order` in `factory`.

Such an evaluator does not "evaluate" the Self expressions in the operations' bodies in the original sense of the word, but rather checks the "correctness" of these expressions in the context of the EER view. In order to do so, the evaluator requires:

- An *environment* that functions as the context to evaluate an expression in. Such an environment consists of modeling elements of the EER view.
- An *eval* function that checks whether the elements of the evaluated expressions are visible or "known" in a certain context. If necessary, this function can trigger the interactive creation of new elements in an entity.
- An *apply* function that checks whether the message sends in the expressions of the operations' body are valid, by evaluating the message selectors in their appropriate context.

The Environment

The *environment* that is used during evaluation, is in fact the set of attributes, operations and relationships of the original entity where the operation is evaluated. We also add the local variables of the operation's body to the environment.

The set of attributes and operations can be generated dynamically.

A first appproach is to generate this collection from the implementation objects. This is based on the assumption that in SelfSync, attributes are synchronised as data slots in the public part of the prototype of the implementation object, while all operations are synchronised to method slots in the parent part of the implementation object.

As mentioned before, the roles of the relationships in an entity are annotated in the private category of the implementation object. For example, in the slot `roleInRelation_vendor_supplier`, the role `'supplier'` is stored. In the slot `m_to_n_Relationship_vendor_supplier`, the implementation object `supplier` is stored. Based on advanced string operations, we can find whether the entity of the evaluated operation has a relationship, with the string in question as role.

Alternatively, the set of attributes and operations can be generated from the graphical representation of the entity itself. By navigating the graphical morphs the entity is composed of, we can construct a set with the names of attribute morphs, the operation morphs and the labels of the relationship links.

The Eval Function

The `eval` function itself works as follows :

- For simple expressions, such as integers and strings ⇒ `(exp eval) ='true'`. Nothing has to be checked, since these simple expressions are known in the `globals` namespace.

- For variables, such as `stock` ⇒ `(exp eval) = (exp lookupIn:environment)`. We have to look up these variables in the current environment. Due to the universal access to data and method slots, variables can denote attibutes, relationships as well as operations. Also assignments such as `stock:` are looked up as attributes. If found, `'true'` is returned. If not found, we call `createNewVariable`.

  The `createNewVariable` method interactively creates a new attribute, operation or relationship. The interactions are similar to the ones we implemented for the Interactive Prototyping phase. Based on the decision of the user, the EER view is extended and SelfSync automatically synchronises the implementation view with new code.

- For applications, such as `stock + 5` or `supplier factory order` ⇒ We consider three kinds of messages that are evaluated with an `apply` function:

  - Unary message: `(exp`$_1$ `op) eval = apply:(op exp`$_1$`)`

  - Binary message: `(exp`$_1$ `op exp`$_2$`) eval = apply:(op exp`$_1$ `exp`$_2$`)`
  - Keyword messages: `(exp`$_1$ `op`$_1$ `exp`$_2$ `op`$_2$`...op`$_n$ `exp`$_{(n+1)}$ `eval = apply:((op`$_1$ `op`$_2$ `...  op`$_n$`) exp`$_1$ `exp`$_2$`...exp`$_{(n+1)}$`)`

The Apply Function

The `apply` function works as follows:

- First, the arguments are evaluated one by one in the current environment.
- Second, the operator is evaluated. Therefore, the first argument is looked up in the current environment. Depending on where in the environment it is found, the following actions are taken:
  - **The first argument is created or found as an attribute**: the application stops here. The reason is that in the implementation view, all attributes are by default clones of the `simpleAttribute` prototype. What the future value of the attribte will be, is only known at run-time. Therefore, there is no use to apply the operator to the attributes. E.g. `stock + 5`: this will only work when stock already contains an integer.
  - **The first argument is created or found as an operation**: the application stops here. The reason is, also in this case, that in the implementation view the outcome of a method is only known at call time. Due to the late binding in the dynamically typed Self language, we cannot always determine what the result of a method will be. Applying operators to that unknown result has no use. E.g. `doSomething size`.
  - **The first argument is created or found as a relationship**: the application is to be realised in the new environment of the entity that is referenced by this new relationship. For example, `supplier factory order`: then `factory order` is to be evaluated in the context of the `supplier` entity. Therefore we switch the environment to the new entity. Next the operator `factory order` is evaluated in the new environment.

Example of An Operation Evaluation

As a example consider the expression `supplier factory order` that is evaluated in a standalone the `vendor` entity:

- `(supplier factory order) eval`

  ⇒ *First evaluate the expression. In this compound unary message,* `supplier` *is the argument and* `factory order` *is the operator.*

- `apply:(((factory order) eval)(supplier eval))`

  ⇒ *First,* `apply` *starts with the argument* `(supplier eval)`*.*

- apply:((apply:(order eval)(factory eval)))**(supplier eval)**

  ⇒ *This results in creating a new relationship: the environment switches to* `supplier`*. The operator is evaluated in this new environment.*

- apply:**((apply:(order eval)(factory eval))**(supplier eval))

  ⇒ *First,* `apply` *starts with the argument* `(factory eval)` *that results in creating a new relationship: the environment switches to* `factory` *and continues.*

- apply:((apply:(order eval)**(factory eval)**)(supplier eval))

  ⇒ *Next,* `apply` *evaluates the operator argument* `(order eval)` *in the new environment* `factory` *that results in creating a new operation* `order` *in* `factory`*.*

6.3 Towards Visual Aspect-Oriented Programming

A second extension to AMS is *Code Injections* that implement `before/after` constructs in a number of scattered operations in a data modeling view. These operations can be selected graphically as join points. Code Injections combined with the visual selection of join points can be considered as a first step in the direction of Visual Aspect-Oriented Programming (AOP).

6.3.1 Code Injections

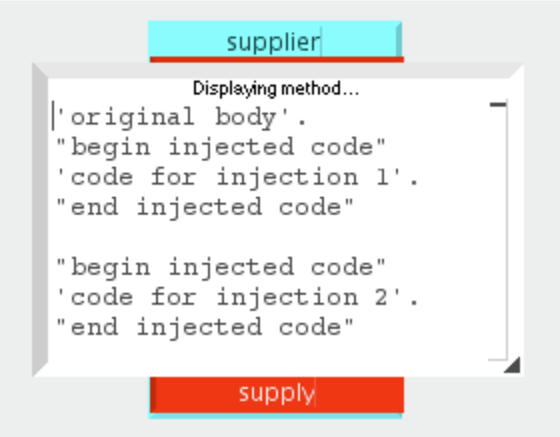
Code Injections involve behaviour that is "injected" before or after one or more selected operations in the EER diagram. This is a simple *visual* version of Aspect-Oriented Programming [62], where static join points are selected graphically in a diagram instead of described by a pointcut. This new piece of code is again automatically added at the beginning or end of the method bodies of all selected operations in the data modeling view.

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. However, code statements that are similar to the injections but were added outside the injection mechanism will not be deleted.

6.3.2 Code Injections in the EER View

In this section we illustrate how the EER view can be adapted to include Code Injections.

We provide a notation for an operation's code, that clearly distinguishes between the original body and its injections. An editable operation with two injections is illustrated in figure 6.9.

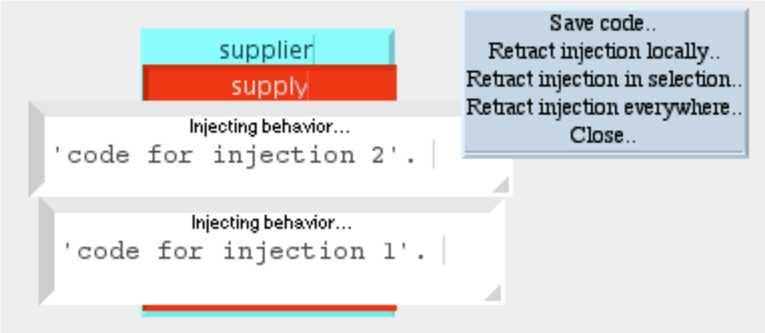


```
'original body'.
"begin injected code"
'code for injection 1'.
"end injected code"

"begin injected code"
'code for injection 2'.
"end injected code"
```

Figure 6.9: An operation's body with two code injections.

The menu items in figure 6.1, that relate to Code Injections are:

- **Show code injections**: to open a window for each previously added injection containing the code of that injection. Figure 6.9 illustrates two code injections and their menu.


supplier
supply
Injecting behavior...
Injecting behavior...
Save code..
Retract injection locally..
Retract injection in selection..
Retract injection everywhere..
Close..

```
'code for injection 2'.
```

```
'code for injection 1'.
```

Figure 6.10: An operation's body with two code injections.

- **Inject code**: to open an empty editor window. The injection code in the edited window can then be injected locally or in every selected operation. In figure 6.11 an injection can be added in the two selected operations.

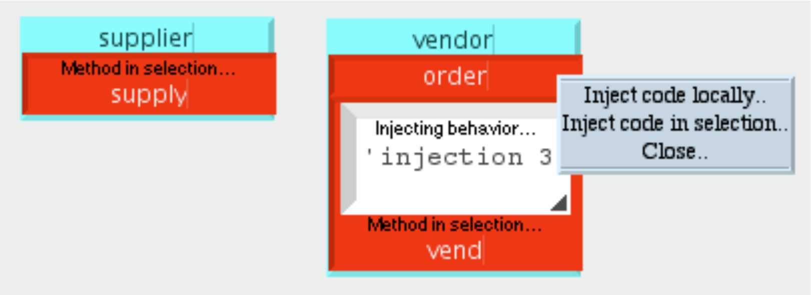


Figure 6.11: Code injection in selected operations.

- **Select method**: to select a number of operations.

6.3.3 Language Constructs for Code Injections

The implementation of Code Injections demands a mechanism to graphically select operations and mechanisms to inject and remove code locally or globally.

Language Mechanism 6.4 ***A Joinpoint Reification Mechanism*** *allows different operations in the data modelling view to be graphically selected, one after the other.*

Such a Joinpoint Reification Mechanism possibly combines a Method Reification Mechanism and a Method Absorption Mechanism with the View for Methods.

New injections can be added locally or globally (in all selected operations). The selection mechanism accumulates the data modelling view's operations onto which this code injection will be applied. It thus serves as a graphical pointcut language that gathers an extensionally defined set of joinpoints. This requires reification of methods in order for the selection mechanism to be able to accumulate them.

Language Mechanism 6.5 ***A Code Injection Mechanism*** *merges the original method body combined with all its injections in the data modelling view into one new method body in the implementation view.*

This mechanism can be based on the Bidirectional Connection (see definition 3.1), the Method Absorption and Reification Mechanisms, and the Addition and Removal Primitives of definition 3.12.

The original method body and all the injections are kept separately in the data modelling view, enabling them to be deleted one-by-one.

Language Mechanism 6.6 ***A Code Removal Mechanism*** *deletes previously added injections locally or globally (everywhere they were added at the time).*

Again, such a mechanism can be based on the Bidirectional Connection (see definition 3.1), the Method Absorption and Reification Mechanisms, and the State and Behaviour Addition and Removal Primitives of definition 3.12.

6.3.4 Presence of Code Injections Language Constructs In Programming Languages

In this section we discuss the possibilities to implement Code Injections in Smalltalk, Ruby and Self.

Code Injections in Smalltalk

A special extension of Smalltalk can function as a Code Injection Mechanism and a Code Removal Mechanism. *Method Wrappers* [19] support adding hidden behavior before or after a method without recompiling it, used for changing a method's behavior.

Multiple method wrappers can be nested, forming layers of added before and/or after code around the original method. These layers can be removed one by one dynamically. The current strategy of this approach is recompiling a method, which results in the method wrappers not being installed on the new method, but this strategy can be changed if necessary.

A side-effect of this strategy is that when adding or removing instance variables from a class, the method wrappers will no longer be installed since the entire class — hence also its methods — are recompiled.

Another shortcoming of method wrappers is that the decompiled source code of a method wrapper cannot be viewed. This is a result of the method wrapper referring to itself and thus going into an infinite loop when decompiled source code is being generated.

Code Injections in Ruby

A Code Injection Mechanism and a Code Removal Mechanism in Ruby can be constructed with nested singleton methods that function each as a method wrapper for the previous one. The problem here is that the method is exclusively wrapped for the run-time object containing the singleton methods, and for its copies. Therefore implementations exist that support an `override` construct for singleton methods: the wrapped method then becomes available for the super classes after redefinition.

A more robust method wrapping mechanism is implemented via the Cut construct [129]. A cut is used to encapsulate advice for a single class. Cuts are self-contained units similar to classes and have their own state, as well as private auxiliary methods. Although the Cut class is very similar to the Class class, it cannot be instantiated. Rather it is used solely as an "invisible overrider". Similar to Class the Cut can be defined *anonymously*, either through instantiation or as a special singleton. The anonymous definition can be especially convenient for internal (method) wrappings, as illustrated below where the method m1 is wrapped.

```
class A
  def m1; 9;
end


Cut.new(self) do
    def m1
      '<before code>' + super + '<after code>'
    end
  end
end
```

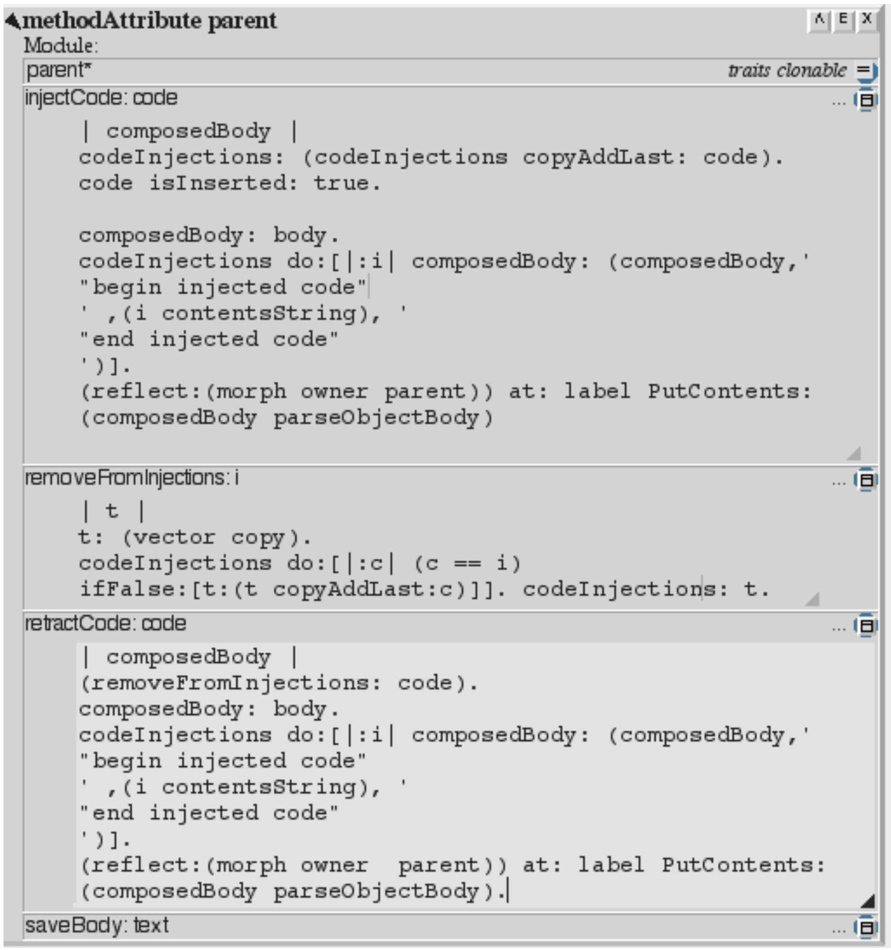
In order to be able to cut-across multiple classes a shared module can serve as a simple aspect by its inclusion in a cut for each class.

Code Injections in Self

A Code Injection Mechanism and a Code Removal Mechanism in Self, could be partially based on a combination of the parsing mechanism described above in order to keep the injections separated. Previously added injections can be deleted locally or globally with removeSlot: (everywhere they were added at the time). The changed method body is then parsed again into the method slot. In the following section we describe how Code Injections can be implemented in SelfSync.

6.3.5 Code Injections Implementation in Self

The behavior to manage code injections is implemented in the methodAttribute parent, as illustrated in figure 6.12.



methodAttribute parent

Module:

parent* — traits clonable

injectCode: code

```
| composedBody |
codeInjections: (codeInjections copyAddLast: code).
code isInserted: true.

composedBody: body.
codeInjections do:[|:i| composedBody: (composedBody,'
"begin injected code"
' ,(i contentsString), '
"end injected code"
')].
(reflect:(morph owner parent)) at: label PutContents:
(composedBody parseObjectBody)
```

removeFromInjections: i

```
 | t |
t: (vector copy).
codeInjections do:[|:c| (c == i)
ifFalse:[t:(t copyAddLast:c)]]. codeInjections: t.
```

retractCode: code

```
| composedBody |
(removeFromInjections: code).
composedBody: body.
codeInjections do:[|:i| composedBody: (composedBody,'
"begin injected code"
' ,(i contentsString), '
"end injected code"
')].
(reflect:(morph owner  parent)) at: label PutContents:
(composedBody parseObjectBody).
```

saveBody: text

Figure 6.12: The `methodAttribute` parent with code injection methods.

Adding Injections As mentioned before, the existing code injections of an operation are stored in a collection `codeInjections` in the `methodAttribute` prototype. The `injectCode:` method extends this collection with a new injection and composes a new body from the original one and the new injection. Next, this new body is added in the appropriate method slot in the implementation view.

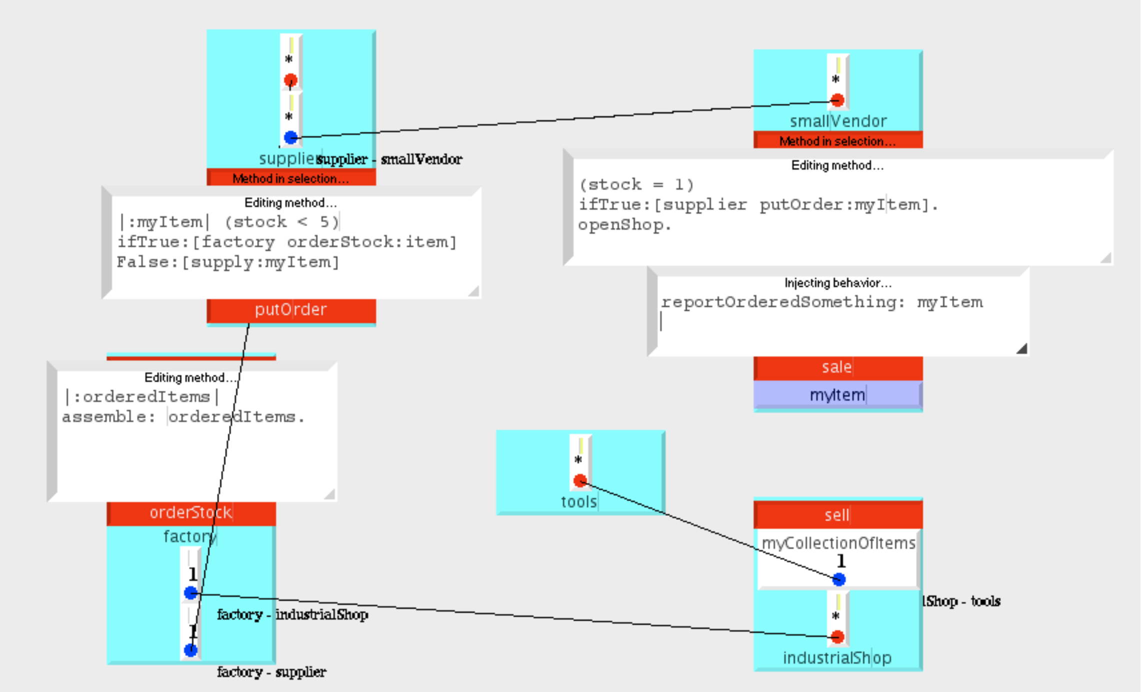
Removing Injections The reverse method `retractCode:` removes the injection in question from the code injections collection and recomposes the body from the original one and the remaining code injections.

Retracting injection code in a selection of operations is realised by calling the `retractCode` for all the methods that are currently member of the collection `selectedMethods`. This collection is stored in `traits methodAttributeMorphs`.

Currently, we do not support code injections in the implementation view. As a result the method synchronisation itself and managing all code injections separately are not compatible yet. We believe however that this optimisation requires minor changes to the Method Absorption and Reification mechanisms in Self.

6.3.6 Code Injection Example

In this section we present an example of a Code Injection. The setup of this example that contains four entities, is illustrated in figure 6.13. In this example, `supplier` is in a many-to-many relationship with `smallVendor`. The entity `factory` is in a one-to-many relationship with `supplier`, and in a one-to-many relationship with `industrialShop`.



```
|:myItem| (stock < 5)
ifTrue:[factory orderStock:item]
False:[supply:myItem]
```

```
(stock = 1)
ifTrue:[supplier putOrder:myItem].
openShop.
```

```
reportOrderedSomething: myItem
```

```
|:orderedItems|
assemble: orderedItems.
```

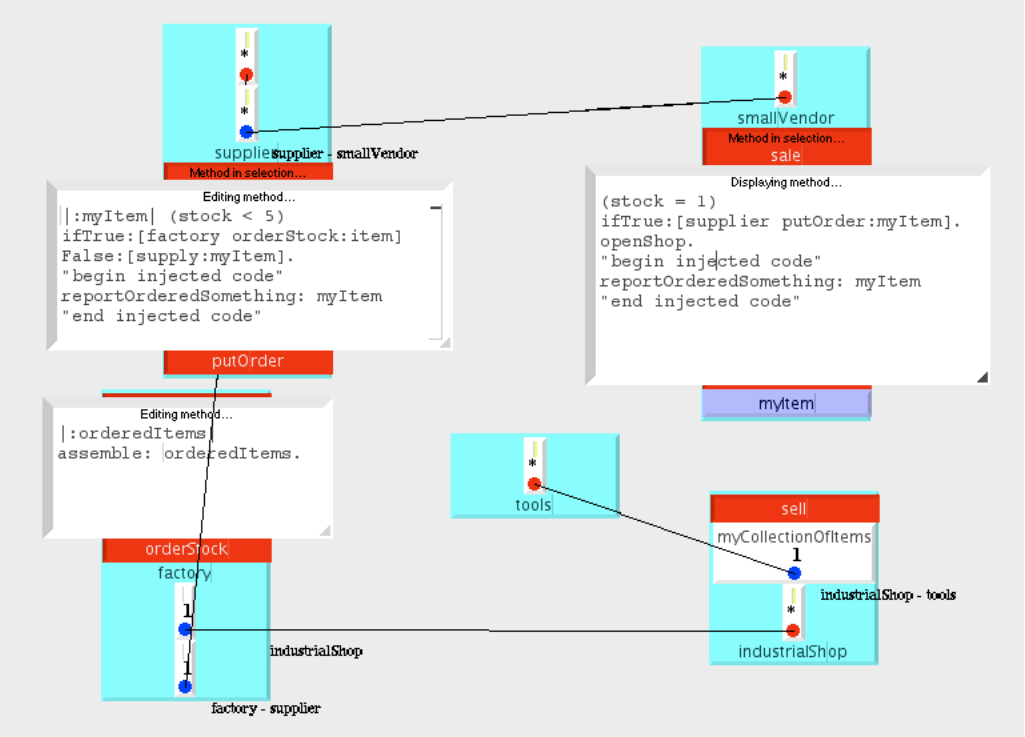
Figure 6.13: A Code Injection example (before the injection).

The idea is that a `smallVendor` orders the one specific item he sells, via the `supplier` that delegates this order to the `factory`. The `industrialShop` orders its collection of items to sell directly in the `factory`.

A possible code injection in this case, is logging code to report that an item has been ordered. We can inject this code anywhere there is some "ordering behaviour" present in the diagram.

In the example in figure 6.13, we select the `sale` and `putOrder` methods. and provide the injection code `reportOrderedItem: myItem`.

The situation after the injection is illustrated in figure 6.14.



```
|:myItem| (stock < 5)
ifTrue:[factory orderStock:item]
False:[supply:myItem].
"begin injected code"
reportOrderedSomething: myItem
"end injected code"
```

```
(stock = 1)
ifTrue:[supplier putOrder:myItem].
openShop.
"begin injected code"
reportOrderedSomething: myItem
"end injected code"
```

```
|:orderedItems|
assemble: orderedItems.
```

Figure 6.14: A Code Injection example (after the injection).

We need to remark that when injecting similar logging code into the behaviour of `industrialShop`, it is necessary to consider that it is an entire collection of items that is ordered instead of the one item.

6.4 Conclusion

We summarise this chapter with the following statements:

- AMS is an ARTE characteristic that applies synchronisation between operations in the data modeling view and methods in the implementation view.

- We defined the language constructs that are required to implement AMS and discussed their presence in C++, Java, Smalltalk, Ruby and Self.

- The EER view was extended with operations that correspond to method slots in the implementation view. Next, AMS was fitted in the ARTE approach.
- An AMS implementation was added to the SelSync ARTE tool.
- We have shown that it is feasible to evaluate operations in SelfSync models. In this way SelfSync EER models can be considered as Executable Models "in embryo". We provided an implementation scheme for an evaluator that evaluates an operation's code in the context of the entities they are part of.
- We have shown how AMS can be extended into Code Injections that are considered as a first step in the direction of Visual AOP. These Code Injections implement `before/after` constructs that are added to or removed from a number of scattered operations in the data modeling view.

Chapter 7

Validation of Our Approach

In this chapter we validate the ARTE approach. We realise a case study that is implemented in the SelfSync ARTE tool. The case study is based on the extended and mature TELEBIB domain model that is part of TELEBIB standard for the Belgian insurance sector (see section 7.1). We show that the ARTE approach is suitable to model an extended and mature domain model such as TELEBIB by modeling it in SelfSync, as explained in section 7.2. The Agile aspect of the ARTE approach is validated for the TELEBIB domain model in section 7.3. The implementation of TELEBIB in SelfSync benefits from the dynamic characteristic of ARTE. More precisely, the constraints that are automatically enforced when run-time programming are applicable to the TELEBIB domain model (see section 7.4).

We found that SelfSync allows to improve certain parts of the TELEBIB domain model such as roles. Moreover, TELEBIB in SelfSync functions as a highly generic domain model that is applicable in a wide range of insurance applications. This is partially caused by the fact that Interactive Prototyping can be used to reduce the extended TELEBIB domain model to a tailored application. There are many relationships in the TELEBIB domain model, that have an explicit cardinality. Also there exists a number of entities in the original domain model that are to be considered as dependents. Finally, TELEBIB includes 23 role entities and obviously benefits from role combination enforcement.

7.1 The TELEBIB Domain Model

7.1.1 The TELEBIB Standard

Since 1994 TELEBIB [148] is the Belgian insurance sector standard for electronic communication and exchange of data. TELEBIB's models are used to enable all users, standard developers and software providers to understand the structure and dynamics of the daily business in the Belgian insurance sector, independent of any technical solution. The models ensure that all interested parties have a common understanding of the sector data and the processes. The main components of the TELEBIB models are:

- *Conceptual Models*: simple models that represent knowledge on insurance contracts, claims, premium notification, accounts and assessment.

- *A Data Model*: a domain model that contains insurance contracts, claims, premium notification, accounts and assessment, represented as an Extended Entity-Relationship (EER) diagram.

- *A Process Model*: a model that represents the collaboration between the represented components of the data model.

7.1.2 The TELEBIB Domain Model

We will focus on the TELEBIB domain model. In its current state, the model covers the major business needs in the areas of policy administration, claims handling and accounting, and thus can be considered as a mature and realistic example of a domain model. The domain model contains 343 entities, 1048 attributes, and 233 relationships and generalisations.

The fragment in figure 7.1 illustrates, among others, that a *Group* has a *GroupName* attribute and can be specialised into a *Couple* or a *FactualGroup*. Also the relationship *FactualGroup-PartyRole* is depicted.

Entities and their attributes are represented by rectangles while relations between two entities are depicted by a diamond, containing the concatenation of the two entity names as a label. Generalization is represented by a white triangle, connected by a bold link to the most general entity. All entities and relationships of the TELEBIB domain model can be found in [148].

The TELEBIB domain model can be divided into the following clusters:

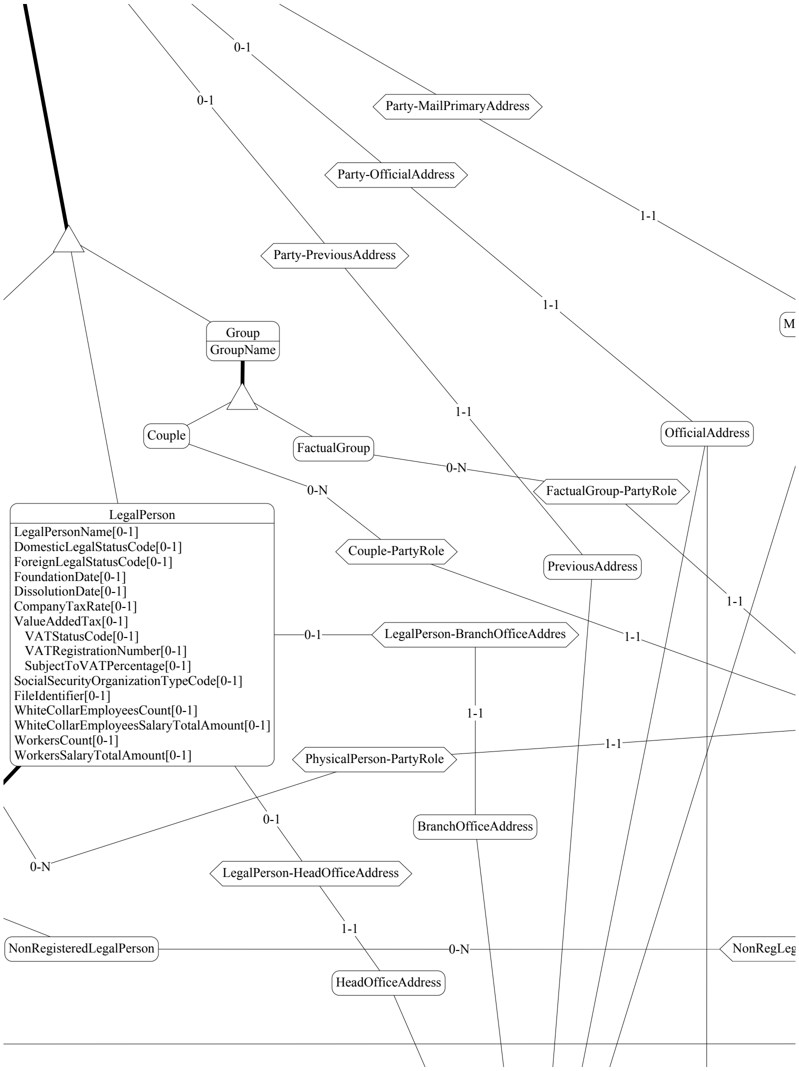


Figure 7.1: An excerpt of the TELEBIB domain model.

- **Parties, their corresponding roles, and party information**: e.g. a *LegalPerson* can fulfill the *PartyRole* of *Insured* and can have an *OfficeTelephoneNumber*. Other examples include:
  - a *Group*: (is-a *Party*) a collection of individuals that together have bearing on insurance, e.g. a group of employees, group of drivers.
  - *InsuranceProvider*: (is-a *PartyRole*) a party registered and controlled by the national Insurance Control Board.
- **Insureable objects**: e.g. an *Animal* is an object that can be insured. Other examples are *InsuranceObject*: something having concrete existence, living or non-living that forms an element

of or constitutes the subject matter of insurance. The object may be a single occurrence of an object or a group of objects. An insurance object can be a person or an activity. E.g. the generalization hierachy between *InsuranceObject*, *MaterialObject*, *Vehicle*, *RoadVehicle*, and *MotorizedRoadVehicle*.

- **Claims, contracts, and policies**: e.g. a *Claim* can be based on a *FireInsuranceContract*. This group also includes:
  - *Contract*: a binding agreement between two or more parties for the doing or not doing of certain things. A contract of insurance is embodied in a written document called the policy.
  - *Policy*: the printed legal document stating the terms of the insurance contract, that is issued to the policyholder by the company.
- **Incidents, accidents, damage, and losses**: e.g. a *FireInsuranceContract* insures against *MaterialDamage* caused by the incident *Fire*. Other elements of this group are:
  - *PhysicalInjury*: (is-a *Damage*) injury to the body as the result of an accident.
  - *WaterDamage*: (is-an *Incident*) damage caused by water.

There exist various relations in and between the 4 groups: e.g. *Insured-Contract*

7.1.3 The TELEBIB Domain Model in SelfSync

We modeled the orginal TELEBIB domain model entirely in SelfSync. A selection of the EER view on the TELEBIB model that corresponds to the excerpt in figure 7.1, is illustrated in figure 7.2.

7.2 Restructuring TELEBIB with Roles

In the original TELEBIB domain model, as it was modeled in SelfSync, a certain *Party* can fulfill one or more *PartyRoles* such as *Driver* or *PolicyHolder*. To represent this role relation the *Role Object Pattern* [38] was used. In this case, the object that can perform roles has a reference to a general role object that is subtyped for each role.

More precisely the following five entities are in a one-to-many relationship with *PartyRole*:

- RegisteredLegalPerson
- NonRegisteredLegalPerson

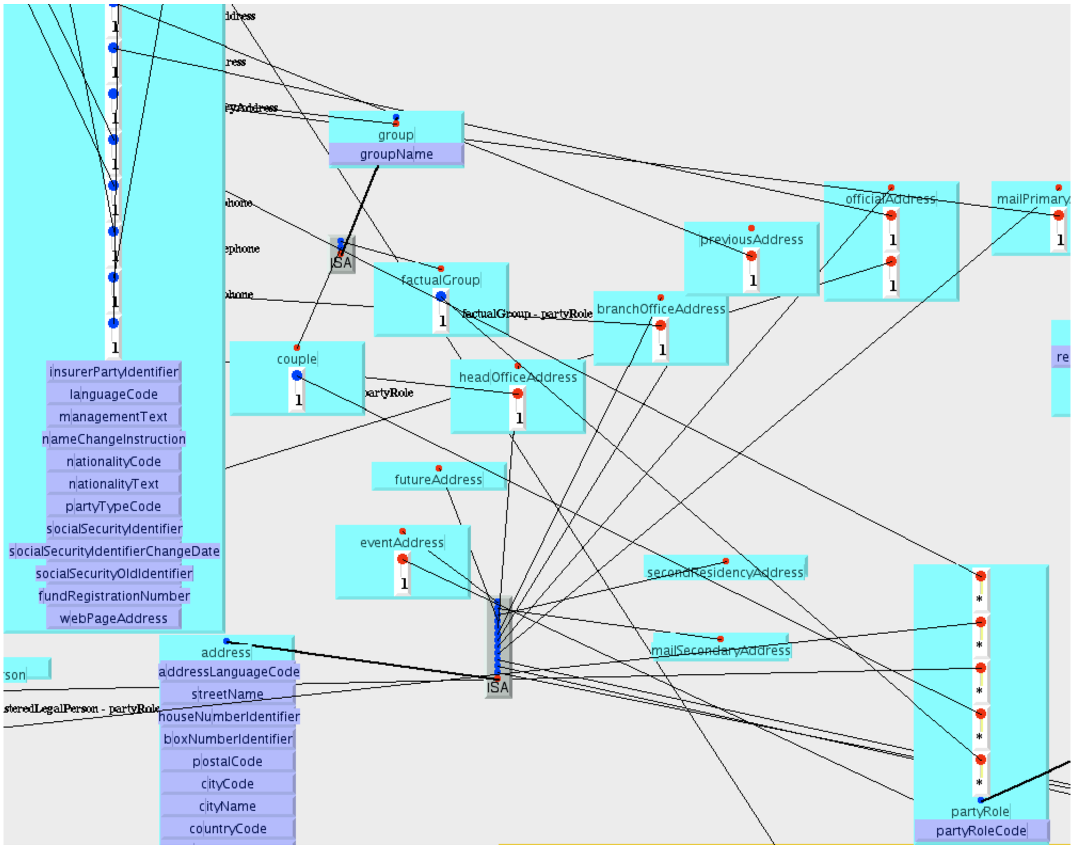


Figure 7.2: An excerpt of the TELEBIB domain model in SelfSync.

- Couple
- FactualGroup
- PhysicalPerson

PartyRole itself can be specialised into the following twenty three roles: *InsuranceProvider, Claimant, PolicyHolder, Repairer, Assessor, Hospital, InterveningAuthority, LeasingCompany, FinanceCompany, Creditor, InsuredPerson, FirstAidSupplier, Driver, Notifier, Requester, Beneficiary, OpponentThirdParty, Witness, DataEntryOperator, MedicalDoctor, VehicleRegistrationPlateHolder, Owner* and *Representative*.

An example is illustrated in figure 7.3. In this fragment, *FactualGroup* is in a one-to-many relationship with *PartyRole*, for which some of its specialised roles such as *Witness* and *Owner* can be found.

With the built-in role relation in SelfSync we can transform this part of the TELEBIB domain model with a real role relationship between the parties and the actual roles. Moreover it becomes

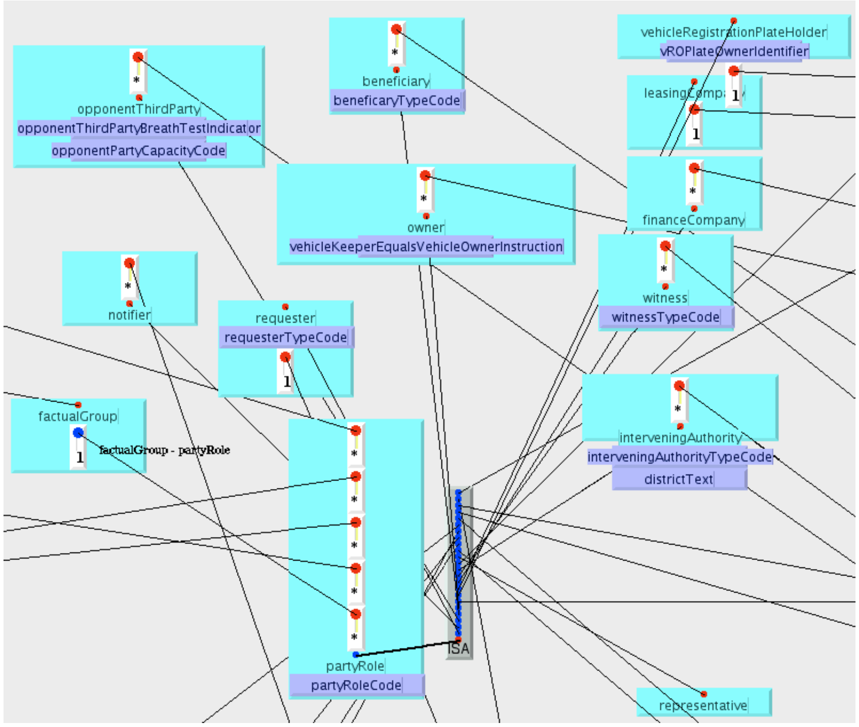


Figure 7.3: TELEBIB's partyRole entity in SelfSync.

possible to express certain illegal role combinations in the data model and to enforce them in the implementation. E.g. it is likely that a party cannot be a witness and an intervening authority at the same time. As a consequence *Witness* and *InterveningAuthority* are members of the same role branch. Similarly some roles are more adequate for certain parties. E.g. it is not likely that a physical person will perform the role of leasing company, as this role is more suitable for a factual group.

Based on the fragment in figure 7.3 one possibility to remodel this part of the TELEBIB domain model is shown in figure 7.4.

Another consequence of the expressive power of the EER modeling language in SelfSync, is that TELEBIB can also be extended with operations. The Advanced Method Synchronisation (AMS) (see chapter 6) in the ARTE approach synchronises the operations with methods in the code. Moreover, code injections can be added into or removed from a set of selected operations the domain model.

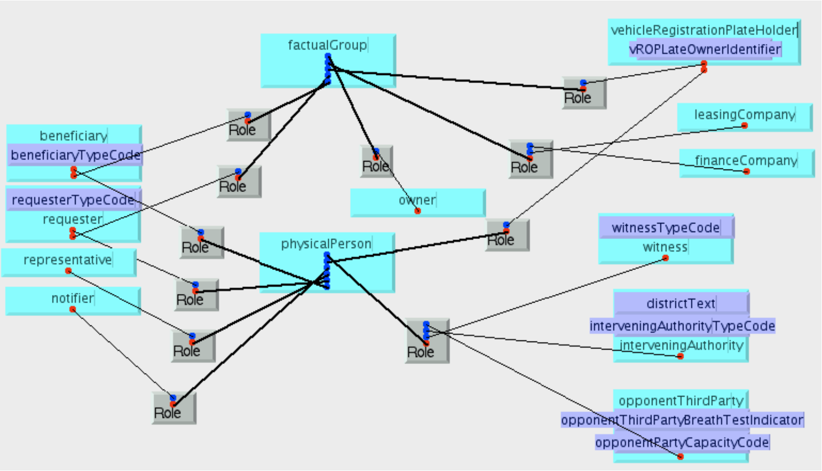


Figure 7.4: Fragment on TELEBIB's roles remodeled in SelfSync.

7.3 Agile Development

7.3.1 TELEBIB's Implementation

While modeling the TELEBIB domain model in SelfSync, a corresponding implementation was automatically generated. The resulting `telebib` namespace that contains the corresponding implementation objects is depicted in figures 7.5 to 7.8.

7.3.2 Extending the TELEBIB Implementation with Application Functionality

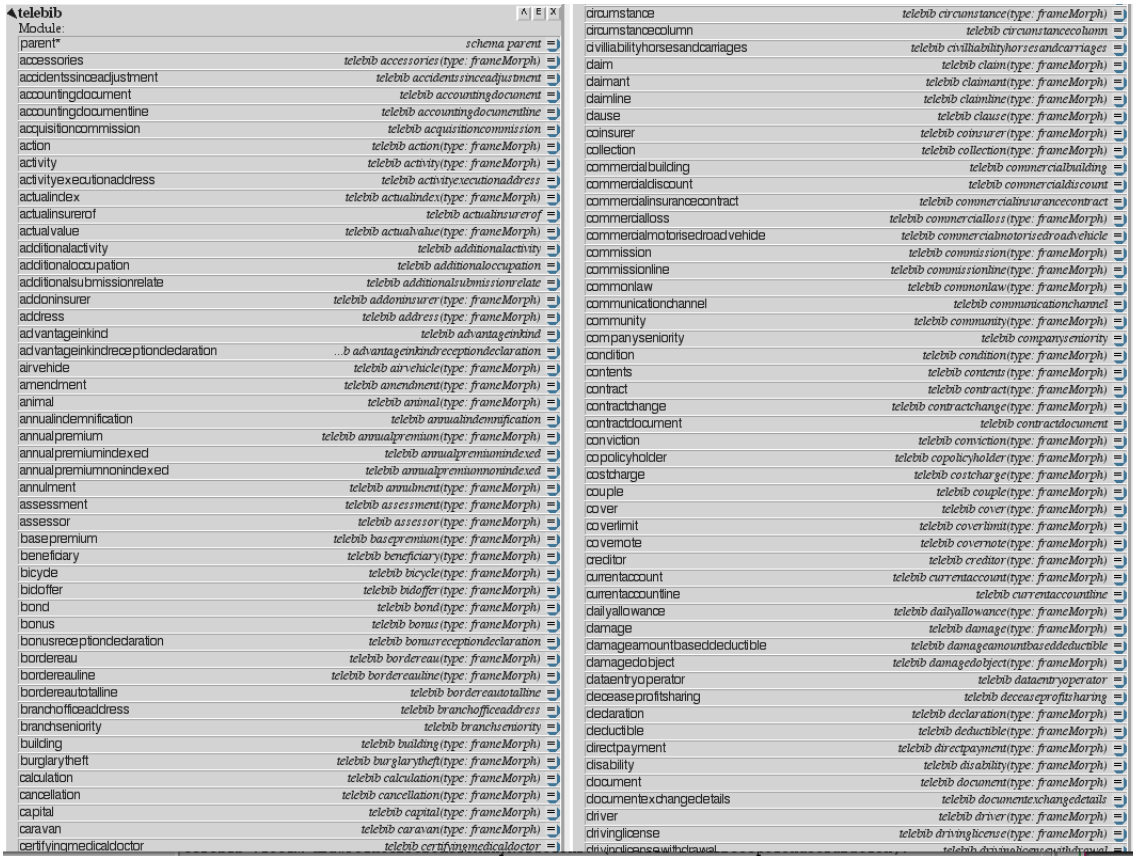
In this section we illustrate how functionality can be added to the resulting TELEBIB implementation for a particular application. Domain analysts can add functionality from inside the EER view, while designers and developers might prefer to change the implementation view directly.

Changes in the EER View

In the EER view, functionality can be added with the Advanced Method Synchronisation that was introduced in chapter 6. More precisely, the TELEBIB domain model can be extended with operations. Possibly, Self code can be added to these operations, from inside the EER view.

TELEBIB's process model[1] that is represented as a dependency graph, can be considered as an example of application functionality for the insurance domain. A fragment of this process model can be found in figure 7.9.

[1] Note that this model is merely an example: there does not exist a standardised process model for the Belgian insurance sector yet.



telebib

Module:

parent*	*schema parent*
accessories	*telebib accessories(type: frameMorph)*
accidentssinceadjustment	*telebib accidentssinceadjustment*
accountingdocument	*telebib accountingdocument*
accountingdocumentline	*telebib accountingdocumentline*
acquisitioncommission	*telebib acquisitioncommission*
action	*telebib action(type: frameMorph)*
activity	*telebib activity(type: frameMorph)*
activityexecutionaddress	*telebib activityexecutionaddress*
actualindex	*telebib actualindex(type: frameMorph)*
actualinsurerof	*telebib actualinsurerof*
actualvalue	*telebib actualvalue(type: frameMorph)*
additionalactivity	*telebib additionalactivity*
additionaloccupation	*telebib additionaloccupation*
additionalsubmissionrelate	*telebib additionalsubmissionrelate*
addoninsurer	*telebib addoninsurer(type: frameMorph)*
address	*telebib address(type: frameMorph)*
advantageinkind	*telebib advantageinkind*
advantageinkindreceptiondeclaration	*...b advantageinkindreceptiondeclaration*
airvehicle	*telebib airvehicle(type: frameMorph)*
amendment	*telebib amendment(type: frameMorph)*
animal	*telebib animal(type: frameMorph)*
annualindemnification	*telebib annualindemnification*
annualpremium	*telebib annualpremium(type: frameMorph)*
annualpremiumindexed	*telebib annualpremiumindexed*
annualpremiumnonindexed	*telebib annualpremiumnonindexed*
annulment	*telebib annulment(type: frameMorph)*
assessment	*telebib assessment(type: frameMorph)*
assessor	*telebib assessor(type: frameMorph)*
basepremium	*telebib basepremium(type: frameMorph)*
beneficiary	*telebib beneficiary(type: frameMorph)*
bicycle	*telebib bicycle(type: frameMorph)*
bidoffer	*telebib bidoffer(type: frameMorph)*
bond	*telebib bond(type: frameMorph)*
bonus	*telebib bonus(type: frameMorph)*
bonusreceptiondeclaration	*telebib bonusreceptiondeclaration*
bordereau	*telebib bordereau(type: frameMorph)*
bordereauline	*telebib bordereauline(type: frameMorph)*
bordereautotalline	*telebib bordereautotalline*
branchofficeaddress	*telebib branchofficeaddress*
branchseniority	*telebib branchseniority*
building	*telebib building(type: frameMorph)*
burglarytheft	*telebib burglarytheft(type: frameMorph)*
calculation	*telebib calculation(type: frameMorph)*
cancellation	*telebib cancellation(type: frameMorph)*
capital	*telebib capital(type: frameMorph)*
caravan	*telebib caravan(type: frameMorph)*
certifyingmedicaldoctor	*telebib certifyingmedicaldoctor*
circumstance	*telebib circumstance(type: frameMorph)*
circumstancecolumn	*telebib circumstancecolumn*
civilliabilityhorsesandcarriages	*telebib civilliabilityhorsesandcarriages*
claim	*telebib claim(type: frameMorph)*
claimant	*telebib claimant(type: frameMorph)*
claimline	*telebib claimline(type: frameMorph)*
clause	*telebib clause(type: frameMorph)*
coinsurer	*telebib coinsurer(type: frameMorph)*
collection	*telebib collection(type: frameMorph)*
commercialbuilding	*telebib commercialbuilding*
commercialdiscount	*telebib commercialdiscount*
commercialinsurancecontract	*telebib commercialinsurancecontract*
commercialloss	*telebib commercialloss(type: frameMorph)*
commercialmotorisedroadvehicle	*telebib commercialmotorisedroadvehicle*
commission	*telebib commission(type: frameMorph)*
commissionline	*telebib commissionline(type: frameMorph)*
commonlaw	*telebib commonlaw(type: frameMorph)*
communicationchannel	*telebib communicationchannel*
community	*telebib community(type: frameMorph)*
companyseniority	*telebib companyseniority*
condition	*telebib condition(type: frameMorph)*
contents	*telebib contents(type: frameMorph)*
contract	*telebib contract(type: frameMorph)*
contractchange	*telebib contractchange(type: frameMorph)*
contractdocument	*telebib contractdocument*
conviction	*telebib conviction(type: frameMorph)*
copolicyholder	*telebib copolicyholder(type: frameMorph)*
costcharge	*telebib costcharge(type: frameMorph)*
couple	*telebib couple(type: frameMorph)*
cover	*telebib cover(type: frameMorph)*
coverlimit	*telebib coverlimit(type: frameMorph)*
covernote	*telebib covernote(type: frameMorph)*
creditor	*telebib creditor(type: frameMorph)*
currentaccount	*telebib currentaccount(type: frameMorph)*
currentaccountline	*telebib currentaccountline*
dailyallowance	*telebib dailyallowance(type: frameMorph)*
damage	*telebib damage(type: frameMorph)*
damageamountbaseddeductible	*telebib damageamountbaseddeductible*
damagedobject	*telebib damagedobject(type: frameMorph)*
dataentryoperator	*telebib dataentryoperator*
deceaseprofitsharing	*telebib deceaseprofitsharing*
declaration	*telebib declaration(type: frameMorph)*
deductible	*telebib deductible(type: frameMorph)*
directpayment	*telebib directpayment(type: frameMorph)*
disability	*telebib disability(type: frameMorph)*
document	*telebib document(type: frameMorph)*
documentexchangedetails	*telebib documentexchangedetails*
driver	*telebib driver(type: frameMorph)*
drivinglicense	*telebib drivinglicense(type: frameMorph)*
drivinglicensewithdrawal	*telebib drivinglicensewithdrawal*

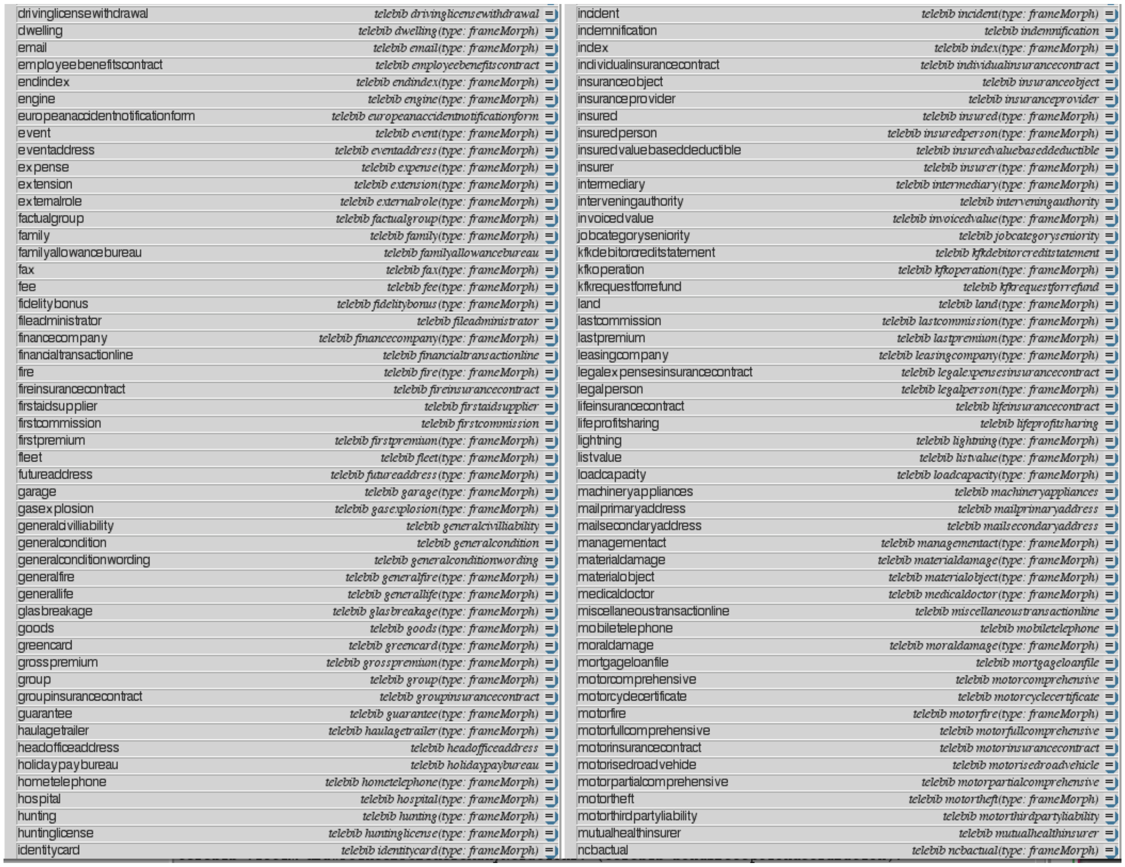
Figure 7.5: The `telebib` namespace in SelfSync (1/4).

In this dependency graph, the nodes with rounded edges are *action states* that are defined as methods, procedures, use cases or "triggers". The nodes with the square edges are entities of the domain model. One action state can call another one, require an entity, or have an entity as its outcome.

For example, the procedure `issueInvoice` has an outcome `premiumInvoice` and calls the procedure `processPremium`. The `processPremium` procedure requires access to the entity `contract`.

In SelfSync, we can model this by adding an operation `issueInvoice` in the entity `netPremiumNonIndexed` (see figure 7.10). Next we can edit the operation body as `contract processPremium`, as the premium that an insured has to pay, depends on the kind of insurance contract.

The operation `processPremium` in `contract` is then added manually or, in the case of an operation evaluator for AMS (see section 6.2), semi-automatically. In the body of this operation, `premium` can safely be used to access the appropriate premium object, and code to create a new



drivinglicensewithdrawal	telebib drivinglicensewithdrawal
dwelling	telebib dwelling(type: frameMorph)
email	telebib email(type: frameMorph)
employeebenefitscontract	telebib employeebenefitscontract
endindex	telebib endindex(type: frameMorph)
engine	telebib engine(type: frameMorph)
europeanaccidentnotificationform	telebib europeanaccidentnotificationform
event	telebib event(type: frameMorph)
eventaddress	telebib eventaddress(type: frameMorph)
expense	telebib expense(type: frameMorph)
extension	telebib extension(type: frameMorph)
externalrole	telebib externalrole(type: frameMorph)
factualgroup	telebib factualgroup(type: frameMorph)
family	telebib family(type: frameMorph)
familyallowancebureau	telebib familyallowancebureau
fax	telebib fax(type: frameMorph)
fee	telebib fee(type: frameMorph)
fidelitybonus	telebib fidelitybonus(type: frameMorph)
fileadministrator	telebib fileadministrator
financecompany	telebib financecompany(type: frameMorph)
financialtransactionline	telebib financialtransactionline
fire	telebib fire(type: frameMorph)
fireinsurancecontract	telebib fireinsurancecontract
firstaidsupplier	telebib firstaidsupplier
firstcommission	telebib firstcommission
firstpremium	telebib firstpremium(type: frameMorph)
fleet	telebib fleet(type: frameMorph)
futureaddress	telebib futureaddress(type: frameMorph)
garage	telebib garage(type: frameMorph)
gasexplosion	telebib gasexplosion(type: frameMorph)
generalcivilliability	telebib generalcivilliability
generalcondition	telebib generalcondition
generalconditionwording	telebib generalconditionwording
generalfire	telebib generalfire(type: frameMorph)
generallife	telebib generallife(type: frameMorph)
glasbreakage	telebib glasbreakage(type: frameMorph)
goods	telebib goods(type: frameMorph)
greencard	telebib greencard(type: frameMorph)
grosspremium	telebib grosspremium(type: frameMorph)
group	telebib group(type: frameMorph)
groupinsurancecontract	telebib groupinsurancecontract
guarantee	telebib guarantee(type: frameMorph)
haulagetrailer	telebib haulagetrailer(type: frameMorph)
headofficeaddress	telebib headofficeaddress
holidaypaybureau	telebib holidaypaybureau
hometelephone	telebib hometelephone(type: frameMorph)
hospital	telebib hospital(type: frameMorph)
hunting	telebib hunting(type: frameMorph)
huntinglicense	telebib huntinglicense(type: frameMorph)
identitycard	telebib identitycard(type: frameMorph)

incident	telebib incident(type: frameMorph)
indemnification	telebib indemnification
index	telebib index(type: frameMorph)
individualinsurancecontract	telebib individualinsurancecontract
insuranceobject	telebib insuranceobject
insuranceprovider	telebib insuranceprovider
insured	telebib insured(type: frameMorph)
insuredperson	telebib insuredperson(type: frameMorph)
insuredvaluebaseddeductible	telebib insuredvaluebaseddeductible
insurer	telebib insurer(type: frameMorph)
intermediary	telebib intermediary(type: frameMorph)
interveningauthority	telebib interveningauthority
invoicedvalue	telebib invoicedvalue(type: frameMorph)
jobcategoryseniority	telebib jobcategoryseniority
kfkdebitorcreditstatement	telebib kfkdebitorcreditstatement
kfkoperation	telebib kfkoperation(type: frameMorph)
kfkrequestforrefund	telebib kfkrequestforrefund
land	telebib land(type: frameMorph)
lastcommission	telebib lastcommission(type: frameMorph)
lastpremium	telebib lastpremium(type: frameMorph)
leasingcompany	telebib leasingcompany(type: frameMorph)
legalexpensesinsurancecontract	telebib legalexpensesinsurancecontract
legalperson	telebib legalperson(type: frameMorph)
lifeinsurancecontract	telebib lifeinsurancecontract
lifeprofitsharing	telebib lifeprofitsharing
lightning	telebib lightning(type: frameMorph)
listvalue	telebib listvalue(type: frameMorph)
loadcapacity	telebib loadcapacity(type: frameMorph)
machineryappliances	telebib machineryappliances
mailprimaryaddress	telebib mailprimaryaddress
mailsecondaryaddress	telebib mailsecondaryaddress
managementact	telebib managementact(type: frameMorph)
materialdamage	telebib materialdamage(type: frameMorph)
materialobject	telebib materialobject(type: frameMorph)
medicaldoctor	telebib medicaldoctor(type: frameMorph)
miscellaneoustransactionline	telebib miscellaneoustransactionline
mobiletelephone	telebib mobiletelephone
moraldamage	telebib moraldamage(type: frameMorph)
mortgageloanfile	telebib mortgageloanfile
motorcomprehensive	telebib motorcomprehensive
motorcydecertificate	telebib motorcyclecertificate
motorfire	telebib motorfire(type: frameMorph)
motorfullcomprehensive	telebib motorfullcomprehensive
motorinsurancecontract	telebib motorinsurancecontract
motorisedroadvehide	telebib motorisedroadvehicle
motorpartialcomprehensive	telebib motorpartialcomprehensive
motortheft	telebib motortheft(type: frameMorph)
motorthirdpartyliability	telebib motorthirdpartyliability
mutualhealthinsurer	telebib mutualhealthinsurer
ncbactual	telebib ncbactual(type: frameMorph)

Figure 7.6: The `telebib` namespace in SelfSync (2/4).

`premiumInvoice` object can be added.

Changes in the Implementation View

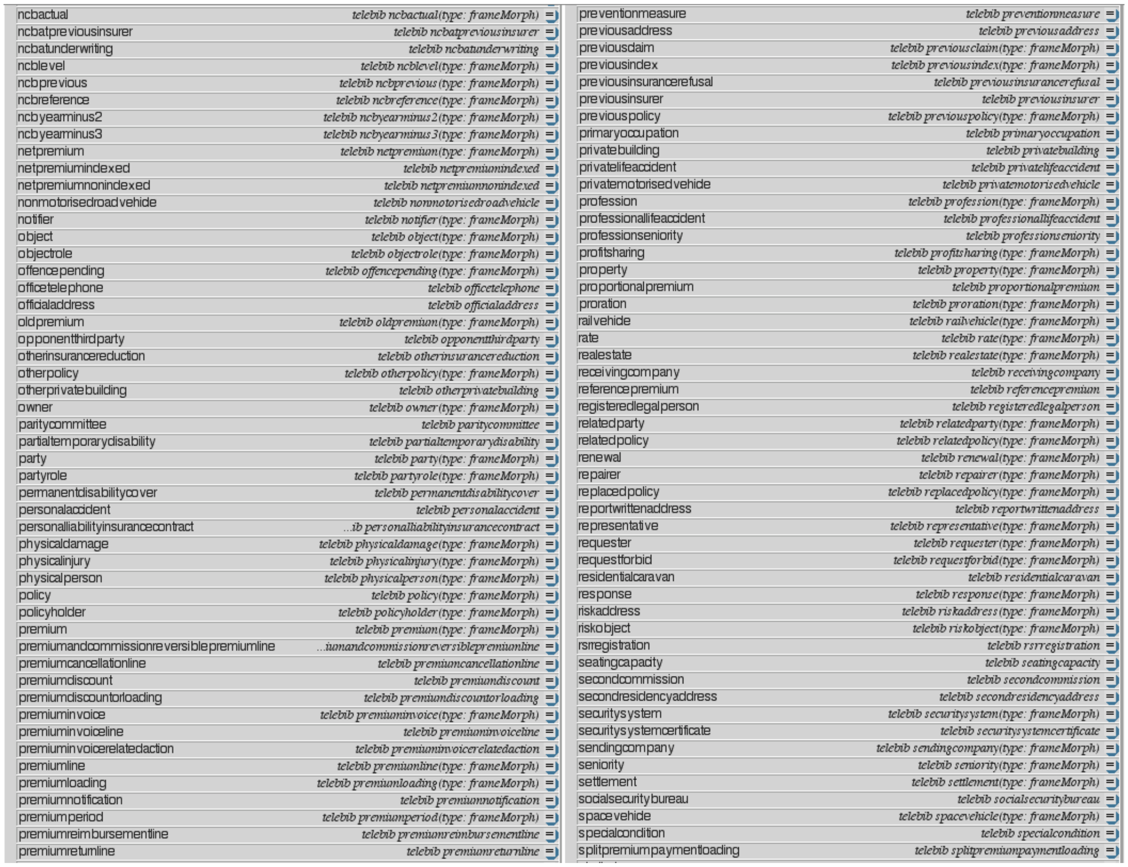
In this section we explain where in the code and in which way developers can add functionality in the TELEBIB implementation.

Using the Self object menu in SelfSync, implementation objects can be extended, with or without propagation to the domain model, as explained in section 5.3.3.

When propagation is desired, adding slots is always done in the prototype object while methods are to be added in the parent object.

Attributes in the data modelling view correspond to slots in the prototypes, that are by default clones of the `simpleAttribute` prototype. Therefore an assignment can be used in the implementation objects, such as `x:1` in order change to their "type" and initialise their value.

For operations in the data modeling view, that were not supplied with a body already in the



ncbactual	telebib ncbactual(type: frameMorph)
ncbatpreviousinsurer	telebib ncbatpreviousinsurer
ncbatunderwriting	telebib ncbatunderwriting
ncblevel	telebib ncblevel(type: frameMorph)
ncbprevious	telebib ncbprevious(type: frameMorph)
ncbreference	telebib ncbreference(type: frameMorph)
ncbyearminus2	telebib ncbyearminus2(type: frameMorph)
ncbyearminus3	telebib ncbyearminus3(type: frameMorph)
netpremium	telebib netpremium(type: frameMorph)
netpremiumindexed	telebib netpremiumindexed
netpremiumnonindexed	telebib netpremiumnonindexed
nonmotorisedroadvehicle	telebib nonmotorisedroadvehicle
notifier	telebib notifier(type: frameMorph)
object	telebib object(type: frameMorph)
objectrole	telebib objectrole(type: frameMorph)
offencepending	telebib offencepending(type: frameMorph)
officetelephone	telebib officetelephone
officialaddress	telebib officialaddress
oldpremium	telebib oldpremium(type: frameMorph)
opponentthirdparty	telebib opponentthirdparty
otherinsurancereduction	telebib otherinsurancereduction
otherpolicy	telebib otherpolicy(type: frameMorph)
otherprivatebuilding	telebib otherprivatebuilding
owner	telebib owner(type: frameMorph)
paritycommittee	telebib paritycommittee
partialtemporarydisability	telebib partialtemporarydisability
party	telebib party(type: frameMorph)
partyrole	telebib partyrole(type: frameMorph)
permanentdisabilitycover	telebib permanentdisabilitycover
personalaccident	telebib personalaccident
personalliabilityinsurancecontract	...ib personalliabilityinsurancecontract
physicaldamage	telebib physicaldamage(type: frameMorph)
physicalinjury	telebib physicalinjury(type: frameMorph)
physicalperson	telebib physicalperson(type: frameMorph)
policy	telebib policy(type: frameMorph)
policyholder	telebib policyholder(type: frameMorph)
premium	telebib premium(type: frameMorph)
premiumandcommissionreversiblepremiumline	...iumandcommissionreversiblepremiumline
premiumcancellationline	telebib premiumcancellationline
premiumdiscount	telebib premiumdiscount
premiumdiscountorloading	telebib premiumdiscountorloading
premiuminvoice	telebib premiuminvoice(type: frameMorph)
premiuminvoiceline	telebib premiuminvoiceline
premiuminvoicerelatedaction	telebib premiuminvoicerelatedaction
premiumline	telebib premiumline(type: frameMorph)
premiumloading	telebib premiumloading(type: frameMorph)
premiumnotification	telebib premiumnotification
premiumperiod	telebib premiumperiod(type: frameMorph)
premiumreimbursementline	telebib premiumreimbursementline
premiumreturnline	telebib premiumreturnline
preventionmeasure	telebib preventionmeasure
previousaddress	telebib previousaddress
previousclaim	telebib previousclaim(type: frameMorph)
previousindex	telebib previousindex(type: frameMorph)
previousinsurancerefusal	telebib previousinsurancerefusal
previousinsurer	telebib previousinsurer
previouspolicy	telebib previouspolicy(type: frameMorph)
primaryoccupation	telebib primaryoccupation
privatebuilding	telebib privatebuilding
privatelifeaccident	telebib privatelifeaccident
privatemotorisedvehicle	telebib privatemotorisedvehicle
profession	telebib profession(type: frameMorph)
professionallifeaccident	telebib professionallifeaccident
professionseniority	telebib professionseniority
profitsharing	telebib profitsharing(type: frameMorph)
property	telebib property(type: frameMorph)
proportionalpremium	telebib proportionalpremium
proration	telebib proration(type: frameMorph)
railvehicle	telebib railvehicle(type: frameMorph)
rate	telebib rate(type: frameMorph)
realestate	telebib realestate(type: frameMorph)
receivingcompany	telebib receivingcompany
referencepremium	telebib referencepremium
registeredlegalperson	telebib registeredlegalperson
relatedparty	telebib relatedparty(type: frameMorph)
relatedpolicy	telebib relatedpolicy(type: frameMorph)
renewal	telebib renewal(type: frameMorph)
repairer	telebib repairer(type: frameMorph)
replacedpolicy	telebib replacedpolicy(type: frameMorph)
reportwrittenaddress	telebib reportwrittenaddress
representative	telebib representative(type: frameMorph)
requester	telebib requester(type: frameMorph)
requestforbid	telebib requestforbid(type: frameMorph)
residentialcaravan	telebib residentialcaravan
response	telebib response(type: frameMorph)
riskaddress	telebib riskaddress(type: frameMorph)
riskobject	telebib riskobject(type: frameMorph)
rsrregistration	telebib rsrregistration
seatingcapacity	telebib seatingcapacity
secondcommission	telebib secondcommission
secondresidencyaddress	telebib secondresidencyaddress
securitysystem	telebib securitysystem(type: frameMorph)
securitysystemcertificate	telebib securitysystemcertificate
sendingcompany	telebib sendingcompany(type: frameMorph)
seniority	telebib seniority(type: frameMorph)
settlement	telebib settlement(type: frameMorph)
socialsecuritybureau	telebib socialsecuritybureau
spacevehicle	telebib spacevehicle(type: frameMorph)
specialcondition	telebib specialcondition
splitpremiumpaymentloading	telebib splitpremiumpaymentloading

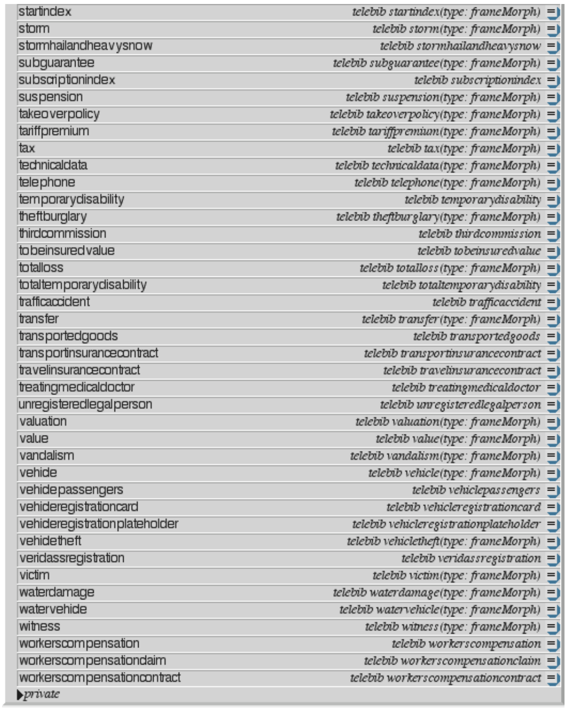
Figure 7.7: The `telebib` namespace in SelfSync (3/4).

EER view, it is clearly indicated in the parent object in the implementation view that code is still to be added for that method. The default body for such empty methods is the string `"Put custom code here."`.

Where to add new methods depends on the range of this new method. Adding a new method in the parent part of the implementation object affects the prototype and all existing and future clones. Adding the method in a run-time object's prototype will result in a singleton method for this one object.

Subclassing implementation objects happens via the Self menu that is invoked on the prototype part of implementation objects. Automatically the domain model is updated with a new specialised entity and a specialisation link.

When changing the contents of data slots in the implementation objects, SelfSync propagates new references between implementation objects to the domain model, as new links that are drawn automatically.



startindex	*telebib startindex(type: frameMorph)*
storm	*telebib storm(type: frameMorph)*
stormhailandheavysnow	*telebib stormhailandheavysnow*
subguarantee	*telebib subguarantee(type: frameMorph)*
subscriptionindex	*telebib subscriptionindex*
suspension	*telebib suspension(type: frameMorph)*
takeoverpolicy	*telebib takeoverpolicy(type: frameMorph)*
tariffpremium	*telebib tariffpremium(type: frameMorph)*
tax	*telebib tax(type: frameMorph)*
technicaldata	*telebib technicaldata(type: frameMorph)*
telephone	*telebib telephone(type: frameMorph)*
temporarydisability	*telebib temporarydisability*
theftburglary	*telebib theftburglary(type: frameMorph)*
thirdcommission	*telebib thirdcommission*
tobeinsuredvalue	*telebib tobeinsuredvalue*
totalloss	*telebib totalloss(type: frameMorph)*
totaltemporarydisability	*telebib totaltemporarydisability*
trafficaccident	*telebib trafficaccident*
transfer	*telebib transfer(type: frameMorph)*
transportedgoods	*telebib transportedgoods*
transportinsurancecontract	*telebib transportinsurancecontract*
travelinsurancecontract	*telebib travelinsurancecontract*
treatingmedicaldoctor	*telebib treatingmedicaldoctor*
unregisteredlegalperson	*telebib unregisteredlegalperson*
valuation	*telebib valuation(type: frameMorph)*
value	*telebib value(type: frameMorph)*
vandalism	*telebib vandalism(type: frameMorph)*
vehicle	*telebib vehicle(type: frameMorph)*
vehiclepassengers	*telebib vehiclepassengers*
vehicleregistrationcard	*telebib vehicleregistrationcard*
vehicleregistrationplateholder	*telebib vehicleregistrationplateholder*
vehicletheft	*telebib vehicletheft(type: frameMorph)*
veridassregistration	*telebib veridassregistration*
victim	*telebib victim(type: frameMorph)*
waterdamage	*telebib waterdamage(type: frameMorph)*
watervehicle	*telebib watervehicle(type: frameMorph)*
witness	*telebib witness(type: frameMorph)*
workerscompensation	*telebib workerscompensation*
workerscompensationclaim	*telebib workerscompensationclaim*
workerscompensationcontract	*telebib workerscompensationcontract*
▶*private*	

Figure 7.8: The `telebib` namespace in SelfSync (4/4).

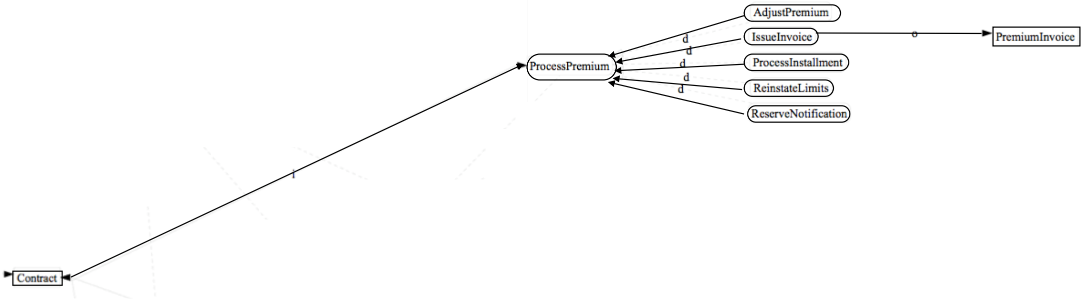


Figure 7.9: Fragment of the TELEBIB process model.

By default changes to run-time TELEBIB objects are not propagated to the domain model.

Creating run-time objects requires the execution of the `copy` operator and initiates an Interactive Prototyping process.

7.3.3 Interference

Due to the extent of the TELEBIB domain model, we have to consider the risk that the many different modeling constructs, via the automated synchronisation, result in an entangled implemen-

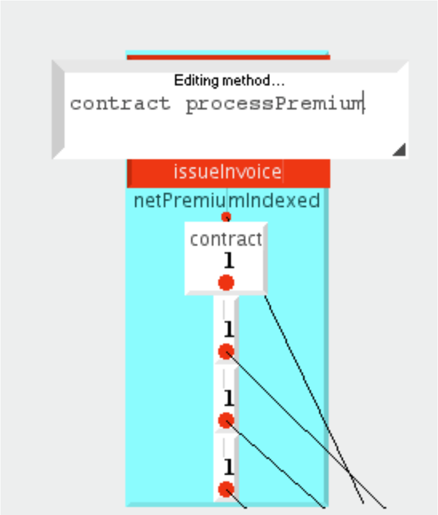


Figure 7.10: Adding application functionality to the TELEBIB domain model.

tation. For example, one specific entity in the EER view can, at the same time, be a specialisation of another entity, perform different roles, be a partner entity in relationships, and be part of an aggregation relationship.

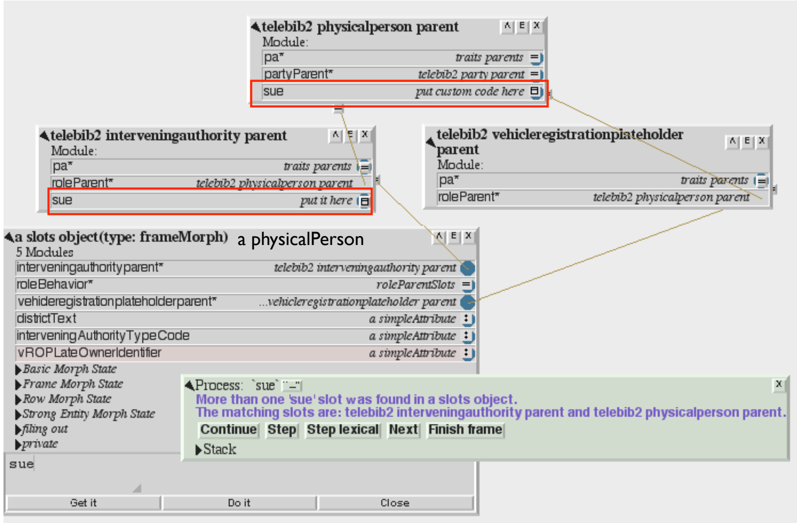
Each of these different modeling constructs is synchronised automatically to a "pattern" in the code that affects the same implementation object. Although we define clear mappings for the synchronisation between different views and use them consistently, there exists one case where these patterns might interfere, caused by multiple inheritance.

Consider the example with roles in figure 7.4. Suppose there is a run-time `physicalPerson` object that currently performs the roles of `vehicleRegistrationPlateHolder` and `intervening-Authority` as illustrated in figure 7.11.

Consider a method `sue` in the `physicalPerson` parent, that is overridden in the `intervening-Authority` parent. Sending the `sue` message to the run-time `physicalPerson` object will result in an ambiguity error. This is caused by the fact that there exist two "inheritance paths" via which a corresponding implementation is found. In this case it is clear however, that it is the most specialised, overriding `sue` method in the `interveningAuthority` parent that is to be invoked.

A possible optimisation for SelfSync is to calculate the distance between the receiver object and the one where the method is found, and execute the method that was found first. However, when for example two roles implement the same method, it remains unclear which one to select.

However, as ambiguity is a still unsolved drawback of multiple inheritance, and implement-



pa* traits parents
partyParent* telebib2 party parent
sue put custom code here

pa* traits parents
roleParent* telebib2 physicalperson parent
sue put it here

pa* traits parents
roleParent* telebib2 physicalperson parent

interveningauthorityparent* telebib2 interveningauthority parent
roleBehavior* roleParentSlots
vehideregistrationplateholderparent* ...vehicleregistrationplateholder parent
districtText a simpleAttribute
interveningAuthorityTypeCode a simpleAttribute
vROPLateOwnerIdentifier a simpleAttribute

sue

Process: `sue` '–'
More than one 'sue' slot was found in a slots object.
The matching slots are: telebib2 interveningauthority parent and telebib2 physicalperson parent.

Figure 7.11: A multiple inheritance diamond in SelfSync's population view.

ing roles gives rise to use multiple inheritance, this shortcoming of SelfSync can be considered inevitable.

7.3.4 Highly Generic Rapid Prototyping

Modeling an extended domain model such as TELEBIB demands a significant effort. SelfSync however provides a corresponding implementation and a semi-automatic instantiation process for free.

The domain model itself can be cloned graphically. Each of the implementation objects is then cloned automatically and can be installed in a new namespace.

Moreover, one and the same domain model is applicable in many cases. In its totality, it can be used for large applications. For smaller businesses the domain model can easily be reduced during Interactive Prototyping, by providing specific interactions for *variation points*. A variation point is an element of the data model, in which the different applications that are derived from this model differ.

An example of a variation point are the subtypes of *Party*. Consider for example an insurance business that only involves individuals, and not companies or groups. Then during Interactive Prototyping the option to specialise a *Party* into a *FactualGroup* is not to be selected. In order to reduce the complexity of an application, when the number of references for a certain relationship is asked during Interactive Prototyping, zero or a negative answer can be provided.

It is also possible to quickly apply some minor changes to the domain model, before creating a new working implementation. For example, we can replace some role combinations by others, as in different applications different "rules" to combine roles are valid. The code is immediately updated automatically. The Interactive Prototyping phase also immediately integrates the new role combination constraints. The role combination constraints in this case, are enforced during the entire lifetime of the new objects as well.

7.4 Run-Time Programming with Constraints

7.4.1 Multiplicity Constraint Enforcement in TELEBIB

The TELEBIB domain model consists of numerous relationships with a "one side". For example *Vehicle-MotorTheft* is a one-to-one relationship denoting that an insurance for motor theft covers one vehicle. For the implementation of each entity on a "one side" of a relationship, SelfSync will ensure that this uniqueness constraint is enforced at all times.

TELEBIB contains two relationships with an explicit cardinality larger than one. First, a *EuropeanAccidentNotificationForm* is in a one-to-two relationship[2] with *CircumstanceColumn*[3]. Second, a *CircumstanceColumn* is associated with at most six *Circumstances*. This setup is illustrated in figure 7.12. When new `telebib` run-time objects are created, the explicit cardinalities are enforced correctly at run-time.

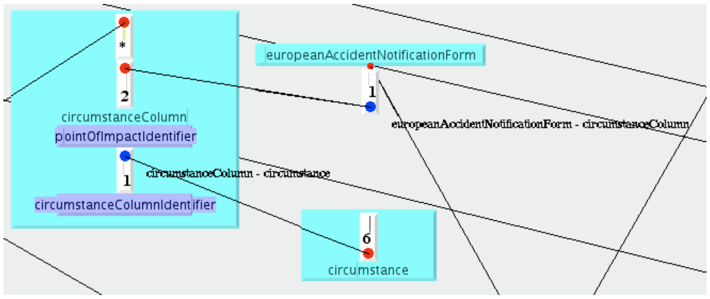

europeanAccidentNotificationForm
1
europeanAccidentNotificationForm - circumstanceColumn
*
2
circumstanceColumn
pointOfImpactIdentifier
1
circumstanceColumn - circumstance
circumstanceColumnIdentifier
6
circumstance

Figure 7.12: Fragment on TELEBIB's explicit cardinalities in SelfSync.

7.4.2 Object Dependency in TELEBIB

The original TELEBIB domain model does not differentiate between strong and weak entities. In its original form we modeled all attributes as simple attributes in SelfSync.

[2] This relationship could be replaced by an aggregation link in SelfSync

[3] A column on a European Accident Notification Form with predefined circumstances.

Nevertheless certain entities are good candidates to become dependent to other entities that contain a primary attribute. Since most attributes in the original TELEBIB model are optional (e.g. *OtherPolicyEndDate[0-1]*), we chose to select primary attributes from the attributes that are demanded. For example, *Profession* has one demanded attribute *ProfessionTypeCode*, that we can consider as a primary attribute. *Profession* is in a one-to-one relationship with *ActivityExecutionAddress* as illustrated in figure 7.13 Therefore the *ActivityExecutionAddress* entity is remodelled as a weak entity that is dependent to *Profession*. Note that *Victim* is also in a one-to-many relationship with *ActivityExecutionAddress*.

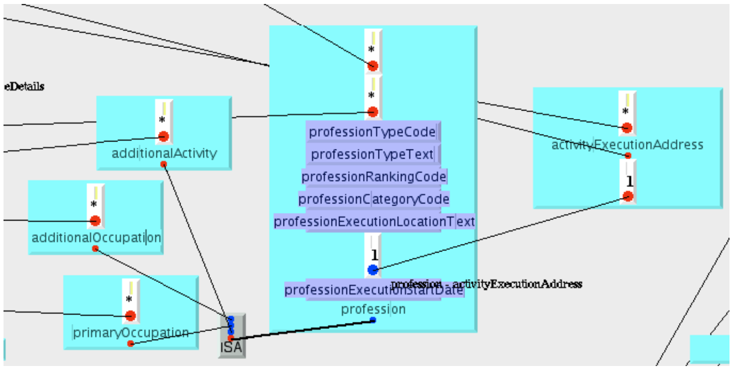


Figure 7.13: Fragment on a TELEBIB dependency in SelfSync.

As a consequence of the dependency relation, whenever a new run-time `activityExecutionAddress` object is created, it is required to reference at least one run-time `profession` object. When a run-time `profession` object is deleted all run-time `activityExecution` object are deleted as well, given that they are not referenced by a run-time `victim` object.

7.4.3 Role Combination Enforcement in TELEBIB

The mutual exclusion of the roles that are modeled in the example in figure 7.4 is also enforced at run-time. This ensures for example that a *Party* in TELEBIB can never be a *Witness* and an *OpponentThirdParty* at the same time. SelfSync allows for the enforcement of other combination rules that might be important for TELEBIB applications.

7.5 Conclusion

We can summarise this chapter as follows:

- TELEBIB is an extended and mature domain model that is represented as an EER diagram with 343 entities, 1048 attributes, and 233 relationships and generalisations.
- TELEBIB is modeled in SelfSync, resulting automatically in a corresponding implementation.
- SelfSync's expressive power allows to restructure parts of TELEBIB such as the roles of a *Party*.
- It is clear when and where extensions in the implementation view are to be realised.
- Interference problems in TELEBIB are mainly caused by multiple inheritance.
- TELEBIB in SelfSync is a highly generic domain model that is applicable in many cases. The Interactive Prototyping phase can be used to reduce the model in certain variation points. Moreover different kinds of application functionality can be added in the implementation objects.
- TELEBIB in SelfSync benefits from the constraint enforcement at run-time for multiplicities, dependencies and role combinations.

Chapter 8

Related Work

In this chapter we describe the related work for research domains that are discussed in this dissertation. For the domain of data modelling, we discuss a number of combinations between EER and objects and briefly discuss EER and UML modelling tools (see section 8.1). Note that a comparison between EER and UML was already presented in chapter 2. A discussion of related approaches for implementing roles was already presented in section 4.4.4. We describe the state-of-the-art RTE tools and focus on to which extent they implement the myARTE requirements (see section 8.2). Next we compare our myARTE approach to MVARE, an automatic MVC engineering approach that also considers multiple views on one underlying model (see section 8.3). In section 8.4 we describe an model-driven approach for data base evolution. Finally, in section 8.5 we discuss one of the first concrete examples in the domain of Executable Models.

8.1 Data Modelling

8.1.1 (Extended) Entity-Relationship Mappings

To map the (E)ER model to other domains, various mappings were defined to databases, information systems and objects.

There exists a 1-to-1 mapping between an ER diagram and a relational database schema. A lot of research has been done on the mapping of (E)ER diagrams onto databases [33, 89] and onto information systems [46, 77].

Since the late eighties, the promising combination between the (E)ER model and object-orientation [23, 83] was researched. The (E)ER model is usually mapped onto objects to be used in object-oriented databases, and not onto "real" programming objects. However, there is no difference between these objects at the design level. Various approaches and techniques exist for translating EER into object-oriented design and implementation:

- In [63] the gap between the ER and the OO model is bridged by introducing the category type and the possibility to define relations between types.
- Both [81] and [82] apply a set of mapping rules to transform EER diagrams into OO database schemas.
- In [79] EER schemas are transformed into OO database schemas by using a clustered form of the EER schema to reduce complexity of conceptual schemas.
- In [36] a default mapping between EER diagrams and OMT diagrams is provided, i.e. from entity objects to design objects. These guidelines are then used to create or re-engineer object-oriented databases, but are also suitable to develop implementation objects.
- [70] introduces evolutionary ER schemas: the original ER schema is mapped to a version derivation graph which is in its turn mapped onto an object-oriented data model.
- [45] provides a step-by-step description to map an EER diagram into an object model, in order to re-engineer an existing database into an object-oriented database.
- In [44] a formal transformation scheme is provided to map ER schemas onto an object-oriented specification language, to be used for object-oriented databases.
- In [56] a formal calculus to tranform ER schemas into object-oriented schemas is defined.
- The OOEER model [48] integrates different ER notations into an OOA notation and ensures consistency is maintained (interactively) between the ER view, the OOA view and a corresponding implementation view.

Since, in myARTE we consider mappings between entities and objects, fine-grained mappings such as the one between EER entities and OMT objects [36] are more suitable for our approach than rules that map entire EER diagrams to database schemas.

8.1.2 EER Tools

There exist a number of tools specifically developed for drawing EER diagrams such as Visio [151], Dia [130] and Gershwin [133]. Development environments for UML often also support ER diagrams, e.g. the Together Round-Trip Engineering environment [150] (see section 2.2). These applications do not support optimal transformations between EER diagrams and a corresponding implementation.

Recently however, the promising Core Data application [127, 128] became part of the Mac OSX development framework. In Core Data, an application is modeled in ER diagrams, while as much as possible corresponding code and the interface is generated automatically.

8.1.3 UML Tools

Since UML is the defacto modelling language for software design, there exists a vast body of UML tools. Examples are Together Designer [149] that is part of the Together Round-Trip Engineering environment [150], the open source project ArgoUML [125] and MagicDraw [138]. For more tools we refer to [41].

For the myARTE approach, it is more feasible to use UML tools with a direct connection to a programming environment, e.g. Together, than to work in stand-alone modeling tools, such as MagicDraw.

8.2 Advanced Round-Trip Engineering Tools

In this section we present the Together RTE tool [150] and the Naked Objects approach [86], and briefly discuss other tools that provide (partial) support for one or more myARTE characteristics. Next, we describe to which extent Together and Naked Objects support myARTE (see section 8.2.2).

8.2.1 Tool Support

In this section we discuss one of the leaders of the state-of-the-art in RTE that includes application such as Rational XDE [144], Borland Together [150], and FUJABA [132]. We also discuss tools that implement synchronisation techniques, that are relevant in the context of advanced RTE.

Together One of the leaders in the RTE domain is Borland Together Control Center [150]. Together supports modelling, stand-alone forward or backward engineering for entire UML and ER diagrams, and provides full RTE for UML class diagrams.

The synchronization mechanism between UML class diagrams and implementation is realized by the *LiveSource* technology and is based on the Model-View-Controller (MVC) pattern [42] as illustrated in figure 8.1. This implies that a change initiated in a view is not actually performed in

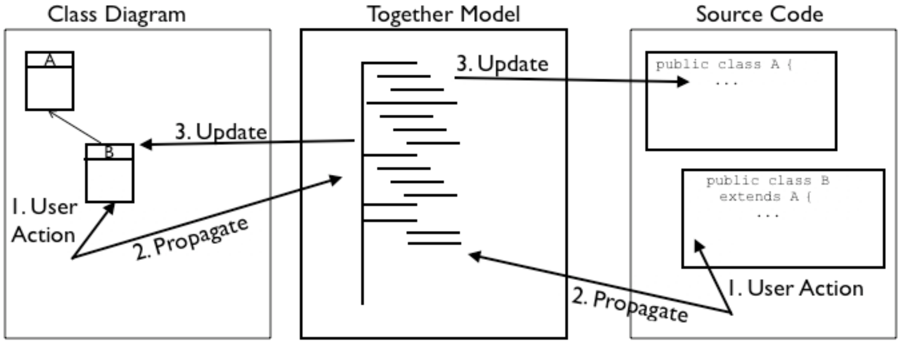


Figure 8.1: Architecture of the Together [150] RTE application.

the view, but in the underlying implementation element, which results in the relevant views being automatically updated. More specifically, the implementation view (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.

Similar to Together, most RTE tools apply model transformations in their synchronization mechanisms. In the case of Together only the specific part of the view that has been changed is tranformed into the other views. Less advanced tools tranform the entire changed view.

Naked Objects The *Naked Objects* [86, 87, 139] approach is relevant for optimal transformations and supports a very limited form of run-time RTE programming, as explained in section 8.2.2. Similar to Together it applies the Model-View-Controller (MVC) architecture illustrated in figure 8.2.

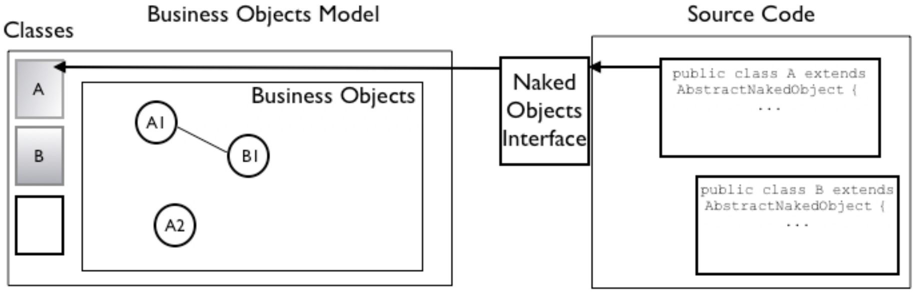


Figure 8.2: Architecture of the Naked Objecst [86] RTE application.

In this case, building a business system consists solely of defining the domain business objects in Java, that are made directly visible to, and manipulatable 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. This happens dynamically without any code generation. All user actions such as creating new business objects, specifying their attributes, adding associations between them, or invoking methods on them, take place at the level of the object business model.

Other Tools Other tools that implement multiplicity constraint enforcement (see section 2.5.3) or object dependency (see section 2.5.3) between implementation objects based on an abstract data model, can be found in the domain of object-relational (O/R) mappers. These tools generate an implementation from a data model and possibly support synchronization of both models. The state-of-the-art includes LLBLGen [137], a commercial O/R mapper that generates a data layer in C++ or Visual Basic code from Entity-Relationship diagrams (see section 2.7, NIAM [122] or ORM [146] database schema's. This generated (.NET) code is compiler-ready and can, being compiled by the appropriate compiler, be used immediately by other applications.

8.2.2 ARTE Support in Together and Naked Objects

Compared to other existing tools, Together and the Naked Objects approach are two of the most suitable tools that could partially support myARTE. Therefore, in the following section we investigate to which extent they actually implement the myARTE characteristics.

Optimal transformations Together's LiveSource provides forward RTE support when evolving the following data modelling elements: classes, attributes, operations, relations and specializations. Similarly, it provides backward RTE support when evolving the following object-oriented implementation elements: classes, attributes, methods, references between classes and inheritance between classes. The transformations are minimal in both directions and happen per attribute.

Naked Objects are restricted to backward RTE. The transformations happen similarly to those in Together, i.e. through MVC, but an additional compilation step of the changed Java code is necessary. There is also no support for synchronizing evolving references and inheritance in the Java code.

Run-time object generations Neither Together nor Naked Objects supports full run-time RTE support where the evolution of attributes, operations, relations, specializations, cardinalities and dependencies are reflected in the run-time objects. The main reason is that a statically typed implementation language such as Java results in too little flexibility to change the data model (or even the source code directly) and to synchronise the corresponding run-time objects, as explained in detail in chapter 3.

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.

Multiplicity constraint enforcement Together's LiveSource only translates cardinalities into comments and puts them just before the variable declaration in the class definition. The only other support offered for cardinalities is not inherent to LiveSource but a consequence of the fact that Together supports the technology of Enterprise Java Beans (EJB) [131], the component model for J2EE [135]. 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. A new container-managed relationship is automatically created. This is realized by (1) adding a container-managed field in both entity beans, together with an abstract getter and setter and (2) adding the relationship declaratively in the XML deployment descriptor. The properties such as relation name and multiplicities are added as comments in the entity bean's source. When an entity bean is deployed, the deployment descriptor is parsed and code is generated to implement the underlying classes.

To the best of our knowledge, the actual enforcement of these cardinalities is limited, and merely a direct consequence of the static typing that is provided by Java. In particular, an attribute cannot contain an object of another type than declared with the attribute. This opposed to our approach, where any kind of run-time object can be referenced in a particular slot. We merely count the references to a particular constrained "type" at run-time, and generate a warning if the multiplicity is not respected.

Naked Objects do not support multiplicity constraint enforcement.

Object dependency Similarly to the enforcing of dependencies using EJB in Together, a `cascade-delete` tag is added to the XML relationship descriptor: when the entity bean is deleted, all its dependents it is in a relationship with, are deleted as well. That part of the enforcement that ensures that a weak object at creation time always references an object it can be dependent to, is not supported.

Naked Objects do not support object dependency.

Method synchronisation There is no support for any kind of method synchronisation in Together or Naked Objects since the contents of methods in the implementation view are not visible in the data modelling views.

Role modelling Role modelling is not supported in Together or Naked Objects since the data modelling language they use in the data modelling view does not include a modelling concept for roles in the sense of behaviour that can be performed temporarily.

8.2.3 Existing RTE Research

Other related work in RTE, is mostly concerned with characterizing RTE rather than providing concrete tool support. In [6], 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 [6] approach advocates the automatic derivation of this inverse transformation function based on the original transformation function. However, this research focuses more on the formal characterisation of RTE than on the implementation.

In [95] RTE is connected to inconsistency handling. The author 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. This idea justifies that part of our myARTE approach where, for some mappings between the data modelling view and the implementation view, we annotate implementation objects with reified information in “private” slots.

8.3 Automatic Model-View-Controller Engineering (MVARE)

[68] advocates an RTE approach where one model as core repository is considered, on which multiple views are defined that each show certain aspects of this model. They propose automatic model view controller engineering (MVARE), a method for keeping the model and its views consistent. In particular, in this method transformations from the model to each of the views should be defined manually, and the inverse transformations should be calculated automatically. A con-

crete implementation of the MVARE method is presented, CODEX, based on double pushout graph transformations. Codex identifies different states and events in the model and views, and describes the way changes to one of the views are propagated to the model and consequently to the other views in terms of these states and events. Codex implements an infrastructure for managing this, by giving the elements of model and views different roles, tags and colours. This infrastructure, together with a restriction on the matching of the transformation rules, allows for the automatic inversion of transformation rules. However, this infrastructure — and in particular the tags, which are created for each element involved in each rule application — introduce a considerable memory overhead. If memory is limited, the tags can be discarded but as a result the transformation rules will be irreversible.

myARTE and the implementation SelfSync also consider a common model and two views, an EER view and an implementation view. Self manages the consistency between the model, i.e. the Self code, and the implementation view, which consists of Self outliners. Hence, changes to the outliners — the implementation view — are propagated to the model. SelfSync adds a second view, the EER view. Changes to the EER view are also first propagated to the model, after which the outliners are automatically updated. For practical purposes, changes to the outliners are directly propagated to the model and the EER view at the same time. The transformations that keep the views and model consistent are implemented in SelfSync, hence no inverse transformations can be calculated automatically.

However, SelfSync also support constraint enforcement, more specifically of multiplicities, dependencies and role combinations. Although it would be possible to create a run-time view in Codex, enforcing the aforementioned constraints is a task that cannot be realised with the current transformations in Codex. Constraint enforcement is not represented in terms of changes to a view that are propagated to the model and that are updated in the other views. It rather involves a change to a view that triggers an action in the same view, based on knowledge in another view. Moreover as the graph formalism is not suitable for counting, it becomes hard to enforce multiplicitiy constraints with the current Codex transformations.

8.4 Model-Based Data Migration

[16] reports on an approach that integrates model-driven software development, database access and data migration in order to let schema evolution affect the population objects in a database.

A UML model of a database and a corresponding implementation is considered by an extension of the RTE tool Together. If a reorganization of the database is required and the UML model is changed, the entire old database is cloned. An *upgrader generator* compares the new model with the old model and detects changes. This generator constructs SQL code that changes the cloned database based on the new model. Next, the data of the old database is to be migrated to the new database. As a consequence of cloning the database, for all classes that are not affected by the model evolution step, data migration is already completed. For each of the other classes the upgrader generator creates an update class. All the update classes together form the upgrader API, i.e. a data migration program with customizable hooks. This means that sometimes the developer has to add data migration code himself.

In our approach the changes to a model are incrementally synchronised in the implementation objects. Since the derived run-time objects share the parents of these implementation objects, they are immediately synchronised with the model. Since this approach applies Together, a more detailed comparison of the myARTE characteristics is provided above.

8.5 An Executable Model Example

In this section we discuss the Extreme Models approach [14] that is a concrete example of an Executable Model. In this case, UML diagrams are made executable, cfr. figure 8.3.

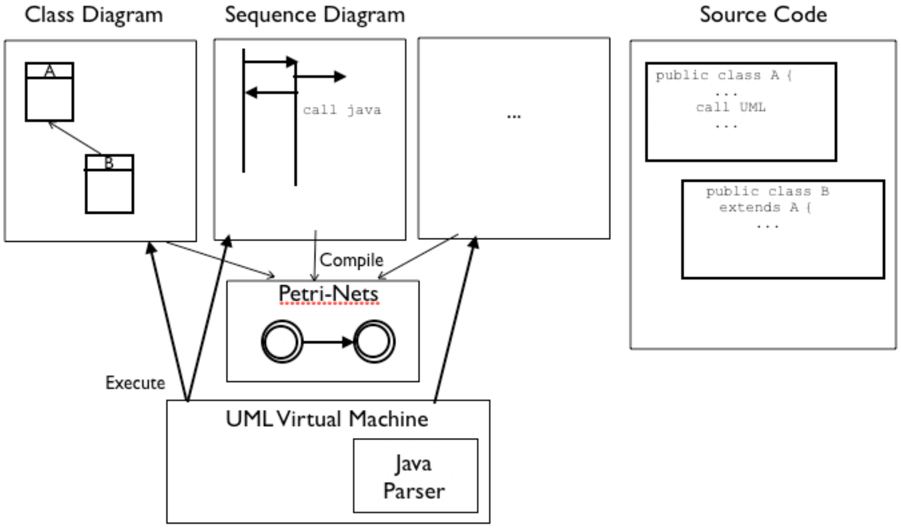


Figure 8.3: The eXtreme models [14] approach.

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. To run a simulation (testing), the needed diagram instances are created, that are translated into the corresponding Petri-Net instances. While the Petri-Net instances run simultaneously, they interact via method calls and events.

The interaction language is Java, interpreted by the UML Virtual machine. Since the diagrams are annotated in Java, they can send messages to real Java objects. In the other direction, diagrams are represented by Java wrapper objects that can be called from Java source code. These two mechanisms minimize the distinction between modeled and coded objects.

An object can be modeled first, while Java code is added later. It is executable in both cases. The Java implementation code is considered another view on the model being constructed, that can be parsed and executed by the UML Virtual Machine. Changes to one specific model are directly reflected in all other views.

Chapter 9

Conclusion

In this chapter we present the conclusions of this dissertation. We summarise the work and contributions realised in this dissertation, followed by a description of future research directions.

9.1 Summary and Contributions

In this dissertation we propose Advanced Round-Trip Engineering as an answer to the limited automated support for Agile Model-Driven Development.

Contribution 1: ARTE extends traditional RTE by considering an analysis view, by supporting an Agile Development cycle, and by integrating run-time objects in the RTE process. As a response to ARTE requirements, we propose the following characteristics for ARTE. First, ARTE considers a data modelling view that enables advanced Role Modelling. Second, ARTE implements Optimal Transformations that are minimal per node and change-preserving. Third, ARTE considers Run-time Object Generations as a part of the RTE process, and constrains them with multiplicities and dependency relationships defined in the data modelling view.

Contribution 2: Based on these three ARTE characteristics, we define a set of language characteristics that are required to implement ARTE tools, and whose presence is proportional to the extent in which languages have dynamic typing, reflective capabilities and flexible object grouping mechanisms.

Contribution 3: We provide a detailed and structured analysis of the above language characteristics in representative object-oriented programming languages C++, Java, Smalltalk, Ruby and Self.
From this analysis we observe that dynamically-typed prototype-based programming languages

with a powerful reflection mechanism are most suitable, if they combine the high flexibility of "everything is an object" with the efficiency of explicit behaviour sharing. Statically-typed class-based languages, such as C++ and Java, in general lack a "live" connection between classes and instances and also reflective power. Due to the dynamic character of Smalltalk and its reflective capabilities, it is in theory possible to simulate the Role Modelling features of ARTE, however, this introduces an enormous management overhead for the generation of new classes and making existing objects instances of these classes and back. Ruby takes the power of Smalltalk one step further with singleton methods and singleton classes that can function as prototypes. However, Ruby still lacks a straightforward separation of state and behaviour so as to implement Role Modelling for Warped Inheritance Hierarchies. Finally, it is Self that combines the high flexibility of "everything is an object" with the efficiency of traits objects to share behaviour, and implements in this way all language characteristics that are required to build an ARTE tool.

Contribution 4: We embed our ARTE instance in current Agile practices.

Our ARTE instance implements an Agile Development cycle that consists of two phases. During the Active Modelling phase that can be fully automated, there is a continuous and incremental synchronisation between the EER view and the implementation view. At any point in time during Active Modelling, the second phase, Interactive Prototyping, can be activated. During this phase, population objects are interactively created and constrained based on the structure of the EER view.

Contribution 5: As a proof-of-concept, an ARTE tool called SelfSync, is presented, that is entirely written in Self and extends it with an ARTE environment.

The SelfSync UI includes a data modelling view that employs a graphical drawing editor for EER diagrams, and is built with the Morphic framework. Entities and implementation objects are represented with outliners that are in fact two views on one and the same underlying Self object. Internally, SelfSync objects are represented by prototypes that combine the state of these two views and with a double traits inheritance hierarchy that also illustrates the duality of the two views.

SelfSync's Active Modelling phase is fully automated and applies Optimal Transformations between the different views. In the EER view, SelfSync employs a dynamically built entity menu whose actions result in annotations and synchronisation in the implementation view. Changes in the implementation view are realised via the Self menu, via the Self UI, and literally via the code.

Run-time Object Generations are created during a semi-automated Interactive Prototyping phase.

The associated behaviour is implemented in a new cloning method for SelfSync objects, that generates interactions depending on their annotations. SelfSync implements Multiplicity Constraint Enforcement and Object Dependency on the Run-time Generations, with reflective constructs that mainly compare implementation objects' references with their annotated objects. Furthermore, this ARTE tool allows for advanced Role Modelling, by implementing general role behaviour, based on the warped hierarchy implementation scheme, and a mechanism to enforce role combination constraints.

Contribution 6: We provide a general solution to implement roles that are modelled in the data modelling view, with the warped hierarchies implementation scheme.

This specific mapping for roles deals with the synchronisation between roles in the EER view, their annotations in the implementation view and their dynamic behaviour and the role combination constraints in the population view. The warped hierarchies implementation scheme merges the viewpoints that roles can be seen as both subtypes and supertypes, and that roles are instances that are to be adjoined to the objects that perform them. This solution enables run-time objects to dynamically start and stop performing roles. Moreover, multiple roles can be performed simultaneously, taking into account that some roles are mutually exclusive.

Contribution 7: Based on Optimal Transformations and on the fact that Self does not distinguish between data and method slots, we extend our ARTE approach with Advanced Method Synchronisation (AMS). This extension allows us to explore two possible future uses of ARTE: Executable Models and Visual AOP.

We consider operations in the data modelling view, that have their corresponding methods in the implementation view. Methods and their bodies are visually rendered in the data modelling view and are synchronized with the implementation and population views. We define the language constructs that are required to implement AMS and investigate their presence in C++, Java, Smalltalk, Ruby and Self. An AMS implementation was added to the SelfSync ARTE tool: the EER view was extended with operations that correspond to method slots in the implementation view, and AMS was fitted in the ARTE approach.

We show that it is feasible to evaluate operations in SelfSync models. In this way, SelfSync EER models can be considered as Executable Models "in embryo". We provide an implementation scheme for an Entity Checker that evaluates an operation's code in the context of the entities it is part of. Moreover, we illustrate how AMS can be extended into Code Injections that are considered as a first step in the direction of Visual AOP. These Code Injections implement `before/after` constructs that are added to or removed from a number of scattered operations in the data modelling view.

Contribution 8: We validate the scalability, the expressiveness and the usability of the SelfSync tool with the extensive and mature TELEBIB domain model that defines the standard for the Belgian insurance business. This domain model is represented as an EER diagram with 343 entities, 1048 attributes, and 233 relationships and generalisations. SelfSync's Role Modelling concept allows us to restructure parts of the TELEBIB domain model, such as the different roles of persons and objects, that are involved in the insurance business, in a more expressive and concise way. The TELEBIB implementation is continuously synchronised with the domain model and can be extended with application functionality, either in the model or the implementation. Since the SelfSync tool generates many implementation objects, interference occurs caused by mutliple inheritance. This shortcoming of SelfSync can be considered inevitable but we describe a possible optimisation in section 9.2.1. Moreover, TELEBIB in SelfSync can be seen as a highly generic domain model that is customisable for many concrete applications.

9.2 Future Research

In this section we describe possible optimisations and possible future research directions for the work in this dissertation. First, we deal with optimisations to the mappings that are applied for Optimal Transformations (see section 9.2.1). In section 9.2.2 we propose a number of extensions for the SelfSync tool, while in section 9.2.3 includes a discussion on the portability of our ARTE instance. Finally we describe how SelfSync can be combined with a relational database (see section 9.2.4).

Note that two extensions to our ARTE instance are already discussed in chapter 6, where we envision that AMS might one day lead to an approach for creating Executable Models, and to support for Visual AOP.

9.2.1 Advanced Mappings for Optimal Transformations

In this section we describe how the mappings that are applied in the Optimal Transformations in our ARTE instance for EER and Self, can be improved.

In section 4.3 we define a set of rather naive mappings between the EER view and the implementation view. For example, a relationship with a multiplicity > 1 in an entity, automatically maps to a data slot in the corresponding implementation object, that contains a vector. The mappings that involve relationships can be optimised by interactively providing the user with a choice between different implementation patterns. These patterns can for example be based on the work described in [84], that focusses on implementing different kinds of conceptual relationships.

Another optimisation deals with the default creation of a prototype-traits pair in the implementation view, for each entity in the EER view. Due to the convention that attributes in an entity are mapped to data slots in the prototype, while operations in the entity map to method slots in the traits object, we can postpone the creation of a prototype or a traits object until it is actually required to hold a data or a method slot. More precisely, if a new entity is created, we await the initial creation of an attribute or an operation in this entity, before creating a new prototype or a traits object.

In section 7.3.3 we mentioned the problems with ambiguous methods caused by multiple inheritance. A related optimisation is to support an ad-hoc disambiguation when adding operations in the EER view. As these operations are automatically added in the traits objects in the implementation view, it is possible to immediately detect an ambiguity, with the help of reflective constructs. We can then generate an interaction to let the user decide which method has to be selected in the implementation and run-time objects that correspond to the entity where the ambiguous operation is added. A simple solution to realise such a prioritised method could for example use an aliasing mechanism that automatically renames the operation and the corresponding method.

We can let the user decide interactively whether adding attributes in an entity has to be propagated to the run-time objects that correspond to this entity. As mentioned in section 4.6, on the one hand, it intuitively is expected that these changes in an entity view are propagated to the already existing run-time objects. On the other hand, in order to respect a prototype-based style, the above changes should not be propagated to existing run-time population objects that are in fact shallow copies of the implementation objects.

Finally, as mentioned in section 4.6, we can consider mappings from run-time objects to the other views. When a large number of (scattered) population objects created from the same prototype are (manually) changed in the same way, this change could be synchronised in the prototype and thus also in the corresponding entity in the EER view (or vice versa), or, alternatively a new protoype could be copied down from the original one in the implementation view resulting in a new specialised entity in the EER view (or vice versa). Therefore, we can use reflection to detect similar changes in a group of run-time objects and interactively let the user decide on which action to take.

9.2.2 SelfSync 2: Optimising Tool Support

Since SelfSync is a research prototype that was developed to support our ARTE instance, it can be optimised in several ways. Besides the improved mappings described in the previous section, we can provide SelfSync with "full RTE support". Currently this is not supported in SelfSync since we are convinced that it has to be possible to annotate implementation objects and to extend them with application logic, without synchronising the EER view. Full RTE support implies that developers cannot realise any change action in the implementation view, that is not propagated to the EER view, unless this is explicitly mentioned. Currently, in the EER view we synchronise adding data and methods slots in, respectively, prototypes or traits objects, extension via copy-down, and slot references to other implementation objects. However, adding for example a method slot in a prototype (instead of a traits object) is not synchronised in the EER view. Also, establishing slot references to objects that are not part of the RTE process is not synchronised. An optimisation would consider these and similar synchronisations from the implementation view to the EER view.

Another extension to SelfSync enables to "plug in" domain-specific patterns in the RTE process. For example, in [37] a set of analysis patterns for modelling, among others, organisational structures and accounting, are provided. Each of these patterns can be provided with a mapping to a corresponding implementation in Self. When such patterns are plugged into SelfSync, these patterns become available in the entity menu in the EER view: next to entities and attributes, etc., one then can also add an entire pattern, such as for example an Accounting Transaction, to the EER view, that automatically results in the provided synchronisation in the implementation view.

In the other direction, it can become possible to detect recurrent configurations in the implementation objects and define an analysis pattern for them. For example, if a significant number of implementation objects are manually extended with a data slot `name`, SelfSync can interactively suggest to create a new pattern `namedEntity` in the EER view, that contains an entity with an attribute `name`.

9.2.3 Portability of the EER-Self ARTE Instance

In this section we discuss how ARTE can be realised with other data modelling languages such as UML, and with other implementation languages.

We deliberately decided to define a concrete ARTE approach for EER and Self in chapter 4, as opposed to a more generic but abstract description. Nevertheless, the analysis of the data modelling languages EER and UML we provide, and the aforementioned analysis of language characteristics of programming languages, make it possible to reconstruct the ARTE approach in other languages.

The fact that we extended the ER model with specialistations, operations, categorisations and aggregations, combined with the notion that for static data modelling, UML and EER are functionally equivalent, results in a straightforward transition from EER to UML in the data modelling view. Note that the Role Modelling concept in our EER model corresponds to dynamic classifications in UML. Moreover, there exists various works that deal with mappings between EER and UML and in general, it is accepted that when UML class diagrams are used, the transformation from an (E)ER to a UML representation is feasible and easy to perform [33].

In order to realise our ARTE instance with another implementation language that can be used in the implementation and population views, the language requirements we defined in chapter 3 can be used as a kind of cookbook, while our analysis of the presence of these language requirements in object-oriented programming languages can be seen as a feasibility study. For the studied languages we can immediately see which ARTE characteristics can or cannot be supported, as well as the amount of effort that is required to implement the ARTE characteristics. For other implementation languages that were not discussed, we suggest to identify the available meta-programming support, the kind of typing and the mechanisms to share behaviour.

9.2.4 Population Objects in a Relational Database Schema

In this section we discuss why it makes sense to link our ARTE instance to a relational database, and in which way database schemas can benefit from ARTE characteristics.

An EER diagram in the data modelling view can be represented as a relational database schema. A strong entity is mapped to a schema with the same attributes, while a weak entity becomes a table that includes a column for the primary key of the strong entity it depends on. A many-to-many relationship is represented as a schema with attributes for the primary keys of the two participating entities, and the attributes of the relationship[1]. For more mappings between EER and a relational

[1] Note that we do not support attributes in relationships in our current EER model.

database we refer to [33].

Since population objects are concrete instances of the modelled entities, a population object corresponds to a row in a relational database schema. In this way, when we would link a relational database to SelfSync, the Interactive Prototyping phase would become a process to populate databases with run-time objects.

Run-time Object Generations result in schema updates. In this context, it becomes especially useful to propagate changes to a prototype in the implementation view. Adding a new attribute in an entity in the EER view results in updating the corresponding schema with the new attribute, since each of the rows in the schema is then automatically extended with the new attribute containing a default value.

Multiplicity Constraint Enforcement takes it one step further and enables for example detecting the elements in a database that do not longer respect the multiplicity constraints of the database schema. Consider for example a many-to-2 relationship between a `person` entity and an `account` entity. As mentioned before, such a relationship is mapped to a schema that includes the entities' primary keys as its attributes, for example, `person_account=(name,accountNr)`. This representation allows for an optimisation of our Multiplicity Constraint Enforcement: after a multiplicity change, only those population objects (database rows) that contain the attributes in the left (constrained) column of the relation schema are to be checked.

Object Dependency ensures that when a row in the database is deleted, all entries that depend on the former row's primary key value, are deleted as well.

The normalisation of a SelfSync database can be taken into account as follows:

- **No repeating groups**: due to reflection in Self we can determine which population objects of the same Run-time Object Generation reference the same value in the same slots. In extremum, we can suggest to the user to (automatically) create a new object containing the repeating slots, that is shared via copy-down by all appropriate population objects.

- **Columns are homogenous**: A column consist of the values for the same data slot in the population objects of a Run-time Object Generation. During the synchronisation of an attribute in an entity to a data slot in a prototype, we can let the data slot reference a default object. With the help of assignment shadowing for example, we can enforce that the argument of an assignment of this data slot, is always cloned from the same prototype as the default object.

- **Each column is assigned a distinct name**: By default, Self disallows homonymous slots in objects. As a result, entities cannot have attributes with the same name.

- **Duplicate rows are not allowed**: If the modelled entities in SelfSync are enforced to include a primary attribute that moreover has a unique value for each entity, we can avoid repeating groups in the database schema.

Appendix A

The SelfSync Implementation

A.1 Implementation of the `checkSlotContents` Method

```
| check. finished. names. num. other. relLabel1. relLabel2. snames. total |
names: (vector copy).
total: (vector copy).
finished: (vector copy).
"names of entities in schema"
(reflect:currentSchema) do:[|:m| names: (names copyAddLast:( m name))].

"names of slots in entity"
snames: (vector copy).
(reflect:self) do:[|:s|
 ((((s category == 'private') || (s category == 'filing out') ||
(s category == 'Strong Entity Morph State') ||
(s category == 'Row Morph State') ||
(s category == 'Frame Morph State') ||
(s category == 'Basic Morph State') ||
(s isParent)) || (s isMethod)|| (s isAssignment)))
 ifFalse:[snames: (snames
        copyAddLast:((reflect:(s contents reflectee))
creatorSlotHint storedName))]].

"which slots in entity contain other entities of the schema"
(reflect:self) do:[|:s| num:0.
((((s category == 'private') || (s category == 'filing out') ||
(s category == 'Strong Entity Morph State') ||
(s category == 'Row Morph State') ||
(s category == 'Frame Morph State') ||
(s category == 'Basic Morph State') ||
(s isParent)) || (s isMethod)||
 (s isAssignment)))
ifFalse:[

(names includes: (s contents creatorSlotHint storedName)) ifTrue:[
(finished includes:((reflect:(s contents reflectee))
                         creatorSlotHint storedName))
ifFalse:[

"check in vectors for this entity"
```

```
(reflect:self) do:[|:s2|
((s2 contents creatorSlotHint storedName) = 'vector')
ifTrue:[(s2 contents reflectee) do:[|:el|
(((reflect:(s contents reflectee)) creatorSlotHint storedName) =
((reflect:el) creatorSlotHint storedName)) ifTrue:[num: (num + 1)]]]].

"#occurences in slots"
num: (num + (snames occurrencesOf:(s contents creatorSlotHint storedName))).

other: (s contents reflectee).
relLabel1: (labelMorph copy).
relLabel2:(labelMorph copy).
relLabel1 myLabel: ((self label),'_',(other label)).
relLabel2 myLabel: ((other label),'_',(self label)).
check: true.
finished: (finished copyAddLast: (s contents creatorSlotHint storedName)).

"check whether there exist already a link between the two entities"
(self morphs) do:[|:m|(m morphTypeName == 'relationFrameMorph')

ifTrue:[((m relationName = (relLabel1 myLabel))  ||
(m relationName = (relLabel2 myLabel)))
ifTrue:[check
ifTrue:[
"update existing cardinalities"
m morphs do:[|:m|(m morphTypeName == 'pointerTailMorph')
ifTrue:[(m headMorph rawOwner setCardinality: num)]].

m morphs do:[|:m|(m morphTypeName == 'pointerHeadMorph')
ifTrue:[(m tailMorph rawOwner setCardinality: num)]].

check: false]]]].

"if not, draw a link"
check ifTrue:[num > 1 ifFalse:[
(relationshipMorph copyIntoWorld: (self world)
    FromMorph: self Offset: (self pointerPoint)
    ToMorph: other Offset: (other pointerPoint)) addMorph: relLabel1.]
True:[(oneToManyRelationshipMorph copyIntoWorld:
   (self world) Cardinality: num
    FromMorph: self Offset: (self pointerPoint)
    ToMorph: other Offset: (other pointerPoint)) addMorph: relLabel1.]
]]]

False:[

"look at level 1 in vectors"

((s contents creatorSlotHint storedName) = 'vector')
ifTrue:[(s contents reflectee) do:[|:el|

(finished includes: ((reflect:el) creatorSlotHint storedName)) ifFalse:[
num: 0. (names includes:
((reflect:el) creatorSlotHint storedName)) ifTrue:
[num: (num + ((s contents reflectee) occurrencesOf: el)).

total: (total copyAddLast: ((reflect:el) creatorSlotHint storedName)).
total: (total copyAddLast: num).

other: el.
relLabel1: (labelMorph copy).
relLabel2:(labelMorph copy).
relLabel1 myLabel: ((self label),'_',(other label)).
relLabel2 myLabel: ((other label),'_',(self label)).
check: true.
finished: (finished copyAddLast: ((reflect:el) creatorSlotHint storedName)).

"check whether there exist already a link between the two entities"
(self morphs) do:[|:m|(m morphTypeName == 'relationFrameMorph')

ifTrue:[((m relationName = (relLabel1 myLabel))
|| (m relationName = (relLabel2 myLabel)))
ifTrue:[check
ifTrue:[
"update existing cardinalities"
m morphs do:[|:m|(m morphTypeName == 'pointerTailMorph')
ifTrue:[(m headMorph rawOwner setCardinality: num)]].

m morphs do:[|:m|(m morphTypeName == 'pointerHeadMorph')
ifTrue:[(m tailMorph rawOwner setCardinality: num)]].

check: false]]]].

"if not, draw a link"
check ifTrue:[num > 1 ifFalse:[
(relationshipMorph copyIntoWorld: (self world)
    FromMorph: self Offset: (self pointerPoint)
    ToMorph: other Offset: (other pointerPoint)) addMorph: relLabel1.]
True:[(oneToManyRelationshipMorph copyIntoWorld: (self world)
    Cardinality: num
    FromMorph: self Offset: (self pointerPoint)
    ToMorph: other Offset: (other pointerPoint)) addMorph: relLabel1.]

]].
]]]]]].
^finished
```

A.2 The Implementation of the copy Method

A.2.1 ResolveAggregates

```
| aggs. children. childrenNames. clonesChildren. coll. newAssoc.
numOfAssoc. selChild. slotName. slotName2 |

aggs:(vector copy).
aggregationSets do:[|:s| children:(vector copy).
childrenNames:(vector copy).
children: s.
                 children do:[|:c|

    numOfAssoc:
    (userQuery askString:'Let this new ',
(((new label)uncapitalizeAll) capitalize),'
object contain how many ',
(((c label) uncapitalizeAll)),' aggregates?') asInteger.

numOfAssoc do:[
coll: ((reflect:new) at:('collection_',(c label))) contents reflectee.
(reflect:new) at:('collection_',(c label)) PutContents:
(reflect:(coll copyAddLast: c))]
]].
new aggregationSets: (vector copy).
```

A.2.2 ResolveCategorisations

```
| assocs. catPar. categorizationNames. categorizationParents.
children. childrenNames. clonesChildren. newAssoc. numOfAssoc.
roleNames. rolePar. roleParents. selChild. slotName. slotName2 |

((categorizations size) == 0) ifFalse:[
categorizationParents: categorizations.
categorizationNames: (vector copy).
categorizationParents do:[|:p| categorizationNames:
(categorizationNames copyAddLast: (p label))].

"checking whether I was not categorized before,
 if so by default keep the parent"
(reflect: self) copyDowns do:[|:cd| (categorizationParents includes:
(cd immutableParentMirror reflectee))
ifTrue:[userQuery report:'This object is already categorized into',
((cd immutableParentMirror reflectee label)
 uncapitalizeAll).^new]].
catPar: userQuery askMultipleChoice: 'This new ',
((self label) uncapitalizeAll),
 ' object can be categorized in one of the following parents: '
                          Choices: categorizationNames
                          Results: categorizationParents.
(reflect: catPar) createSubclassIn: new.
(reflect:new) at: (((catPar label),'Parent') uncapitalizeAll)
PutContents: (reflect: (catPar parent)).
((reflect:new) at: (((catPar label),'Parent')uncapitalizeAll))
isParent: true.].
```

A.2.3 ResolveSpecialisations

```
| children. childrenNames. clonesChildren. newAssoc.
numOfAssoc. selChild. slotName. slotName2 |

clonesChildren:(vector copy).
specializationSets do:[|:s| children:(vector copy).
childrenNames:(vector copy). children: s.
                 children do:[|:c| childrenNames:
(childrenNames copyAddLast: (c label uncapitalizeAll))].
                 selChild: userQuery
 askMultipleChoice:'This new ',((self label) uncapitalizeAll),
' object can be specialized in one of the following children:
'Choices: (childrenNames copyAddLast:'None')
Results: (children copyAddLast:nil).
           clonesChildren:
          (clonesChildren copyAddLast: (selChild clone)).].
```

```
clonesChildren do:[|:child| (child isNil) ifFalse:
[((reflect:child) removeSubclassFrom:
((reflect:child) copyDowns at: 0) immutableParentMirror reflectee).
(reflect: new) at: 'as',((child label) uncapitalizeAll capitalize)
 PutContents: (reflect:child).

"createSubclassIn: child."
(reflect: (child parent)) at: ((label uncapitalizeAll),'Parent')
PutContents: (reflect:(self parent)).
((reflect: (child parent)) at: ((label uncapitalizeAll),'Parent'))
isParent: true.
(reflect: (child parent)) parentsDo:[|:p|
((p name) == 'parent')  ifTrue:[(reflect: (child parent))
 removeSlot: (p name)]].
]].
new specializationSets: (vector copy).
```

A.2.4 ResolveRoles

```
| roleNames. rolePar. roleParents. selChild. slotName. slotName2 |

"#################################################"
"constraints at creation time by roles"
"#################################################"
((roleSets size) == 0) ifFalse:[roleParents: roleSets.
roleParents do:[|:pset| roleNames: (vector copy).
pset do:[|:p| roleNames: (roleNames copyAddLast: (p label))].

"checking whether I was not performing this role before,
 if so by default keep the parent"
(reflect: new) copyDowns do:[|:cd|
 (roleParents includes:(cd immutableParentMirror reflectee))
ifTrue:[userQuery report:'This object is already performing the role of',
((cd immutableParentMirror reflectee label) uncapitalizeAll).^new]].

rolePar: userQuery askMultipleChoice: 'This new ',
((self label) uncapitalizeAll),
 ' object can perform the role of one of the following objects '
                          Choices: (roleNames copyAddLast: 'None')
                          Results: (pset copyAddLast: nil).

(rolePar isNil) ifFalse:[(reflect: rolePar) createSubclassIn: new.
new currentJobs: (new currentJobs copyAddLast: (rolePar description)).
(reflect:new) removeSlot: 'parent' IfFail: ''.
(reflect:new) at: (((rolePar label),'Parent') uncapitalizeAll)
PutContents: (reflect: (rolePar parent)).
((reflect:new) at: (((rolePar label),'Parent')uncapitalizeAll))
isParent: true.]]].
```

A.2.5 ResolveOneConstraints

```
| assocs. catPar. categorizationNames. categorizationParents. children.
 childrenNames. clonesChildren. newAssoc. numOfAssoc. roleNames.
rolePar. roleParents. s. selChild. slotName. slotName2 |

constraintByOne do:[|:la | s: ((((reflect: currentSchema) at:
 (la uncapitalizeAll)) contents) reflectee).
s isEntity ifTrue:[slotName: la.

(reflect:new) do:[|:t| ((t name matchesPattern:
 ’roleInRelation_’,(new label),’_’,(s  label)) ||
(t name matchesPattern: ’roleInRelation_’,(s label),’_’,(new label)))
ifTrue:[slotName:(t contents reflectee).
(reflect:new) removeSlot: (t name).]].

(userQuery askYesNo:’Establishing exclusive link in this new ’,
(((new label)uncapitalizeAll) capitalize),’ object to a(n) ’,
(s label uncapitalizeAll capitalize),’ object?’)

ifTrue:[
assocs:(s clone).

(reflect:assocs) do:[|:t| ((t name matchesPattern:
(’*’,’relation_’,(new label),’_’,(assocs label))) ||
(t name matchesPattern: (’*’,’relation_’,(assocs label),’_’,(new label))))
ifTrue:[(reflect:assocs) removeSlot: (t name).]].

(reflect:new) do:[|:t| ((t name matchesPattern:
(’*’,’relation_’,(new label),’_’,(assocs label))) ||
(t name matchesPattern: (’*’,’relation_’,(assocs label),’_’,(new label))))
ifTrue:[(reflect:new) removeSlot: (t name).]].

slotName2: (new label uncapitalizeAll).

(reflect:assocs) do:[|:t| ((t name matchesPattern:
’roleInRelation_’,(new label),’_’,(s label)) ||
(t name matchesPattern: ’roleInRelation_’,(s label),’_’,(new label)))
ifTrue:[slotName2:(t contents reflectee).
(reflect:assocs) removeSlot: (t name).]].

(assocs constraintByOne includes: (new label)) ifTrue:[
assocs constraintByOne: (assocs constraintByOne remove: (new label)).
(reflect:assocs) at: slotName2 PutContents:(reflect:new)].

(assocs constraintByN flatIncludes: (new label)) ifTrue:[
(reflect:assocs) at: (slotName2,’Collection’)
PutContents: (reflect:(vector copyAddLast:new)).
(assocs constraintByN) do:[|:v| (v includes:(new label))
 ifTrue:[assocs constraintByN: (assocs constraintByN remove: v)]].
]
.
(reflect:new) at: slotName PutContents: (reflect:(assocs copy)).
"(reflect:new) removeSlot:(s label)."
]

False:[
   isStrongEntity
    ifFalse:[
        (entityOwner == s)
          ifTrue:[userQuery report:’This new ’,
(((new label)uncapitalizeAll) capitalize),’ object is dependent to this ’,
(s label uncapitalizeAll capitalize),’ object:
establish at least one association!’.
```

```
             ^(self copy)]
          False:["no reference but reservate slot"(reflect:new)
at: slotName
PutContents:(reflect: nil).(reflect:new) removeSlot:(s label)]]
     True: [(reflect:new) at: slotName
PutContents:(reflect: nil).(reflect:new) removeSlot:(s label)]]]].
new.
```

A.2.6 ResolveNConstraints

```
| ans. assocList. assocs. catPar. categorizationParents. children.
childrenNames. clonesChildren. constr. newAssoc. numOfAssoc.
 roleNames. rolePar. roleParents. s. selChild. slotName. slotName2 |

constraintByN do:[|:la | s: ((((reflect: currentSchema) at:
 (la at: 0)) contents) reflectee).
s isEntity ifTrue:[slotName: (la at:0).

(reflect:new) do:[|:t| ((t name matchesPattern:
’roleInRelation_’,(new label),’_’,(la at:0)) ||
(t name matchesPattern: ’roleInRelation_’,(la at:0),’_’,(new label)))
ifTrue:[slotName:(t contents reflectee).
 (reflect:new) removeSlot: (t name).]].

(la at:1) > 0 ifTrue:[
ans:
(userQuery askString:’Establish how many links in this new ’,
(((new label)uncapitalizeAll) capitalize),’ object to a(n) ’,
(s label uncapitalizeAll capitalize),’ object?
(maximum arity is ’, ((la at: 1) asString), ’)’) ]
False:[
ans:
(userQuery askString:’Establish how many links in this new ’,
(((new label)uncapitalizeAll) capitalize),’ object to a(n) ’,
(s label uncapitalizeAll capitalize),’ object?’)
].

(ans asInteger > 0) ifTrue:[

assocs:(s clone).

(reflect:assocs) do:[|:t| ((t name matchesPattern:
 (’*’,’relation_’,(new label),’_’,(assocs label))) ||
(t name matchesPattern: (’*’,’relation_’,(assocs label),’_’,(new label))))
ifTrue:[(reflect:assocs) removeSlot: (t name).]].

(reflect:new) do:[|:t| ((t name matchesPattern:
 (’*’,’relation_’,(new label),’_’,(assocs label))) ||
(t name matchesPattern: (’*’,’relation_’,(assocs label),’_’,(new label))))
ifTrue:[(reflect:new) removeSlot: (t name).]].

slotName2: (new label uncapitalizeAll).

(reflect:assocs) do:[|:t| ((t name matchesPattern:
 ’roleInRelation_’,(new label),’_’,(assocs label)) ||
(t name matchesPattern: ’roleInRelation_’,(assocs label),’_’,(new label)))
ifTrue:[slotName2:(t contents reflectee).
```

```
 (reflect:assocs) removeSlot: (t name).]].


(assocs constraintByN flatIncludes: (new label)) ifTrue:[

(reflect:assocs) at: (slotName2,'Collection')
PutContents: (reflect:(vector copyAddLast:new)).
(assocs constraintByN) do:[|:v| (v includes:(new label))
ifTrue:[assocs constraintByN: (assocs constraintByN remove: v)]].
]
False:[
(reflect:assocs) at: slotName2 PutContents:(reflect:new).
(assocs constraintByOne: (assocs constraintByOne remove: (new label))).
].
assocList: (vector copy).
ans asInteger do:[assocList: (assocList copyAddLast: (assocs copy))].
(reflect:new) at: (slotName,'Collection') PutContents: (reflect: assocList).
]

False:[
   isStrongEntity
    ifFalse:[
        (entityOwner == s)
          ifTrue:[userQuery report:'This new ',
(((new label)uncapitalizeAll) capitalize),' object is dependent to this ',
(s label uncapitalizeAll capitalize),' object:
establish at least one association!'.
             ^copy]
          False:[(reflect:new) at: slotName PutContents:(reflect: nil).
(reflect:new) removeSlot:(s label)]]
     True: [(reflect:new) at: slotName PutContents:(reflect: nil).
             (reflect:new) removeSlot:(s label)]]]].
new.
```

A.3 Constraint Enforcement

A.3.1 Multiplicities: the `checkConstraints` method

```
| names1. names2 |
isPopulationMember ifTrue:[
 names1: (vector copy). "of the ones I am pointing to"
 names2: (vector copy). "of the ones that point to me"

checkingConstraints ifTrue:[

"For each 1-to-(1/n) relation I can only point to this one exclusive partner"
(reflect:self) do:[|:s|
 ((((s category == 'private') ||
(s category == 'Basic Morph State') ||
(s category == 'Frame Morph State') ||
(s category == 'Row Morph State') ||
(s category == 'Strong Entity Morph State') ||
(s category == 'Weak Entity Morph State') ||
(s category == 'filing out') ||
 (s isParent)) || (s isMethod)) ||(s isAssignment))
 ifFalse:[
names1: (names1 copyAddLast:
((reflect:(s contents reflectee)) creatorSlotHint storedName)).
"look in vectors"
((reflect:(s contents reflectee)) creatorSlotHint storedName) = ’vector’
ifTrue:[
(s contents reflectee)
do:[|:el| names1: (names1 copyAddLast:
((reflect:el) creatorSlotHint storedName))]
]].


constr1 do:[|:c|((names1 occurrencesOf: c) asInteger > 1) ifTrue:

[userQuery show:’Constraint of 1-to-1 relation between the
’,(self label uncapitalizeAll capitalize),’ and ’,(c capitalize),
’ objects has been violated!
 I am pointing to more than one ’,(c capitalize),’ object...’ ForSecs:1]]].

"For each 1-to-(1/n) only one exclusive partner can point to me"
names2: (vector copy).
(browse referencesOfReflectee: (reflect:self)) do:[|:r| (r isFake)
 ifFalse:[(names2:
names2 copyAddLast:(r mirror creatorSlotHint storedName))]].

constr1 do:[|:c|((names2 occurrencesOf: c) asInteger > 1) ifTrue:
[userQuery show:’Constraint of 1-to-1 relation between the
’,(self label uncapitalizeAll capitalize),’ and ’,(c capitalize),
’ objects has been violated!
More than one ’ ,(c capitalize),’ object is pointing at me...’ ForSecs:1]].
"--------------------------"

"For each (1/n)-to-N relation I can point to at most N partner"


constrN do:[|:c|((names1 occurrencesOf:
(c at: 0)) asInteger > ((c at:1) asInteger)) ifTrue:
[userQuery show:’Constraint of x-to-’,((c at: 1) asString),
’ relation between the
’,(self label uncapitalizeAll capitalize),’ and ’,((c at:0) capitalize),
’ objects has been violated!
 I am pointing to more than ’,((c at: 1) asString),’ ’,
((c at: 0) capitalize),’ objects...’ ForSecs:1]]].

"For each (1/n)-to-N relation at most N partners can point to me"

constrN do:[|:c|((names2 occurrencesOf: (c at: 0))
asInteger > ((c at:1) asInteger)) ifTrue:
[userQuery show:’Constraint of x-to-’,((c at: 1) asString),
’ relation between the
’,(self label uncapitalizeAll capitalize),’ and ’,((c at: 0) capitalize),
’ objects has been violated!
More than ’,((c at: 1) asString),’ ’ ,((c at: 0) capitalize),
’ objects are pointing at me...’ ForSecs:1]].
]
```

A.3.2 Dependencies: the `deleteMe` method

```
dependents do:[|:d|
  (reflect:self) do:[|:s|
    ((((s category == 'private')  ||
(s category == 'filing out')||
(s category == 'Weak Morph State')||
(s category == 'Strong Entity Morph State')||
(s category == 'Basic Morph State')||
(s category == 'Row Morph State')||
(s category == 'Frame Morph State')
 || (s isParent)) || (s isMethod)) ||(s isAssignment))
      ifFalse:[(((reflect:(s contents reflectee))
            creatorSlotHint name) == 'vector')
              ifTrue:[(s contents reflectee) do:[|:e|
                       (((reflect:d) creatorSlotHint name) ==
                       ((reflect:e) creatorSlotHint name))
                        ifTrue:[e deleteMe] False:['']]]
               False:[(((reflect:(d )) creatorSlotHint name) ==
                       ((reflect:(s contents reflectee))
creatorSlotHint name))
                        ifTrue:[(s contents reflectee) deleteMe]]]]]]
False:[(entityOwner isNil) ifFalse:[(entityOwner) dependents:
 ((entityOwner dependents) remove: (self label))]].

(reflect:self) do:[|:s|
    ((s category == 'private')  ||
(s category == 'filing out')||
(s category == 'Weak Morph State')||
(s category == 'Strong Entity Morph State')||
(s category == 'Basic Morph State')||
(s category == 'Row Morph State')||
(s category == 'Frame Morph State'))
 ifFalse:[(reflect: self) removeSlot: (s name)]].
```

Appendix B

Glossary

B.1 Reflection Terminology

Taken from [18].

- *Introspection* is the ability of a program to examine its own structure.
- *Self-modification* is the ability of a program to change its own structure.

B.2 Prototype-Based Programming and Self Terminology

Taken mostly from the *The SELF 4.1 Programmers Reference Manual* [145].

- A *slot* is a name-value pair. The value of a slot is often called its contents.
- An *object* is composed of a (possibly empty) set of slots and, optionally, a series of expressions called code.
- A data object is an object without code.
- A *data slot* is a slot holding a data object.
- An *assignment slot* is a slot containing the assignment primitive.
- An *assignable data slot* is a data slot for which there is a corresponding assignment slot whose name consists of the data slots name followed by a colon. When an assignment slot is evaluated its argument is stored in the corresponding data slot.
- An *ordinary method* (or simply method) is an object with code and is stored as the contents of a slot. The methods name (also called its selector) is the name of the slot in which it is stored.
- A *unary message* is a message consisting of a single identifier sent to a receiver. A *binary message* is a message consisting of an operator and a single argument sent to a receiver. A *keyword message* is a message consisting of one or more identifiers with trailing colons, each followed by an argument, sent to a receiver.
- *Unary, binary, and keyword slots* are slots with selectors that match unary, binary, and keyword messages, respectively.
- *Message lookup* is the process by which objects determine how to respond to a message (which slot to evaluate), by searching objects for slots matching the message.
- *Inheritance* is the mechanism by which message lookup searches objects for slots when the receivers slots are exhausted.
- An objects *parent slots* contain objects that it inherits from.
- *Dynamic inheritance* is the modification of object behavior by setting an assignable parent slot.

- *Cloning* is the primitive operation returning an exact shallow copy (a clone) of an object, i.e. a new object containing exactly the same slots and code as the original object.
- A *prototype* is an object that is used as a template from which new objects are cloned.
- A *traits object* is a parent object containing shared behavior, playing a role somewhat similar to a class in a class-based system. Any SELF implementation is required to provide traits objects for integers, floats, strings, and blocks (i.e. one object which is the parent of all integers, another object for floats, etc.).
- A *copy-down* of a prototype is a concatenation mechanism for state inheritance that copies (some of) the slots of the prototype into a new object, ensuring that changes (adding/removing slots) to the prototype are propagated to all copied-down children.

Bibliography

[1] G. K. A. Agrawal and F. Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2003.

[2] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. *Agile software development methods: Review and Analysis*. VTT, 2002.

[3] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. *Lecture Notes in Computer Science*, 1919:21+, 2000.

[4] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[5] S. Ambler. *The Object Primer: Agile Model Driven Development with UML 2*. Cambridge University Press, 2004.

[6] U. Assman. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82, 2003.

[7] S. Bayer and J. Highsmith. Radical software development. *American Programmer*, 7:35–42, 1994.

[8] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[9] K. Beck. Simple smalltalk testing: With patterns. *Smalltalk Report*, 1999.

[10] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM Press.

[11] B. A. Bichelmeyer. Rapid-prototyping. In *In Ann Kovalchick and Kara Dawson (Eds.), Educational Technology: An Encyclopedia.*, New York, NY, USA, 1990. New York: ABC-CLIO.

[12] G. Blaschek. Type-safe object-oriented programming with prototypes - the concepts of omega. *Structured Programming*, 12(4):217–226, 1991.

[13] C. Bock and J. Odell. A more complete model of relations and their implementation: Roles. *JOOP*, 11(2):51–54, 1998.

[14] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. *Extreme Programming Examined*, pages 175–189, 2001.

[15] G. Booch. *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[16] B. Bordbar, D. Draheim, M. Horn, I. Schulz, and G. Weber. Integrated model-based software development, data access, and data migration. In *MoDELS*, pages 382–396, 2005.

[17] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[18] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[19] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 396–417, London, UK, 1998. Springer-Verlag.

[20] E. Breton and J. B&#233;zivin. Towards an understanding of model executability. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 70–80, New York, NY, USA, 2001. ACM Press.

[21] J. M. Carroll. The evolution of human-computer interaction. In *Annual Review of Psychology, Volume 48, Palo Alto, CA: Annual Reviews*, pages 501–522. 1997.

[22] S. Ceri and J. Widom. Deriving production rules for constraint maintainance. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 566–577, 1990.

[23] P. Chen. Er versus oo. *Proceedings of the 11th Internaltional Conference on Entity-Relationship Approach*, pages 153–156, 1992.

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

[25] P. P. Chen. A preliminary framework for entity-relationship models. In P. P. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis, Proceedings of the Second International Conference on the Entity-Relationship Approach (ER'81), Washington, DC, USA, October 12-14, 1981*, pages 19–28. North-Holland, 1981.

[26] P. Coad and E. Yourdon. *Object-oriented analysis*. Yourdon Press, Upper Saddle River, NJ, USA, 1990.

[27] A. Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[28] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, 1989.

[29] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[30] P. Desfray. *Object engineering: the fourth dimension*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[31] C. Dony, J. Malenfant, and D. Bardou. Classifying Prototype-based Programming Languages. In J. Noble, A. Taivalsaari, and I. Moore, editors, *Prototype-Based Object-Oriented Programming: Concepts, Languages and Applications*, chapter 2, pages 17–45. Springer, Feb. 1999.

[32] A. Duncan and U. Hoelzle. Adding contracts to java with handshake. Technical Report TRCS98-32, 9, 1998.

[33] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley World Student Series, 3 edition, 1994.

[34] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: an extension to the entity-relationship model. *Data Knowl. Eng.*, 1(1):75–116, 1985.

[35] R. E. Filman, M. Haupt, and R. H. (eds.). Proceedings of the 2005 Dynamic Aspects Workshop. Technical Report RIACS Technical Report No. 05.01, RIACS, 2005.

[36] J. Fong. Mapping extended entity relationship model to object modeling technique. *SIGMOD Record*, 24(3):18–22, 1995.

[37] M. Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[38] M. Fowler. Dealing with roles. Technical report, Department of Computer Science, Washington University, 1997.

[39] M. Fowler. Is there such a thing as object-oriented analysis? *Distributed Computing Magazine*, pages 40–41, Sept. 1999.

[40] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[41] M. Fowler and K. Scott. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2000.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[43] R. L. Glass and A. Hunt. *Software Conflict 2.0: The art and science of software engineering*. Developer.*Books, 2006.

[44] 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.

[45] M. Gogolla, A. K. Huge, and B. Randt. Stepwise re-enginieering and development of object-oriented database schemata. In *DEXA Workshop*, pages 943–948, 1998.

[46] M. Gogolla, B. Meyer, and G. Westerman. Drafting extended entityrelationship schemas with queer, 1991.

[47] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[48] J. C. Grundy and J. R. Venable. Providing integrated support for multiple development notations. In *Conference on Advanced Information Systems Engineering*, pages 255–268, 1995.

[49] J. Gustavsson. A classification of unanticipated runtime software changes in java. In *Proceedings of Software Maintenance 2003, IEEE Conference on Software Maintenance*, pages 4–12. IEEE, IEEE, 2003.

[50] J. Gustavsson and U. Assmann. A classification of runtime software changes. In *Proceedings of the First International Workshop on Unanticipated Software Evolution*, 2002.

[51] D. Hay. UML misses the boat. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[52] D. C. Hay. here is no object-oriented analysis. *Data to Knowledge Newsletter*, 27(1), february 1999.

[53] D. C. Hay. *Data Model Patterns: Conventions of Thought*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[54] D. C. Hay. *Requirements Analysis Architecture*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[55] A. Henriksson and H. Larsson. A definition of round-trip engineering. Technical report, Linkopings Universitet, Sweden, 2003.

[56] 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.

[57] J. A. Highsmith. *Adaptive software development: a collaborative approach to managing complex systems*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[58] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[59] A. Jodlowski, P. Habela, J. Plodzien, and K. Subieta. Dynamic object roles – adjusting the notion for flexible modeling. In *IDEAS*, pages 449–456, 2004.

[60] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[61] S. Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, unknown 2002.

[62] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 220–242. Springer-Verlag, 1997.

[63] M. F. Kilian. Bridging the gap between o-o and e-r. In T. J. Teorey, editor, *Proceedings of the 10th International Conference on Entity-Relationship Approach (ER'91), 23-25 October, 1991, San Mateo, California, USA*, pages 445–458. ER Institute, 1991.

[64] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, April 2003.

[65] R. Kramer. icontract - the java design by contract tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[66] D. Kroenke. *Database Concepts*. Prentice Hall, 2000.

[67] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[68] H. Larsson and K. Burbeck. Codex – an automatic model view controller engineering system. Technical report, 2003.

[69] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21-11, pages 214–223, New York, NY, 1986. ACM Press.

[70] C.-T. Liu, S.-K. Chang, and P. K. Chrysanthis. Database schema evolution using ever diagrams. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 123–132, New York, NY, USA, 1994. ACM Press.

[71] P. Maes. *Computational Reflection. PhD Thesis, Vrije Universiteit Brussel*. 1987.

[72] J. Maloney. An introduction to morphic: The squeak user interface framework. *Squeak: Open Personal Computing and Multimedia, chapter 2, pages 39–68*, 2002.

[73] C. B. Medeiros and P. Pfeffer. Object integrity using rules. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 219–230, London, UK, 1991. Springer-Verlag.

[74] S. Mellor. Agile mda. *The MDA Journal: Model Driven Architecture Straight from the Masters*, 2004.

[75] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson.

[76] S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[77] B. Meyer, G. D. Westerman, and M. Gogolla. Drafting er and oo schemas in prototyping environments.

[78] M. Minsky. A framework for representing knowledge. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 1–25. de Gruyter, Berlin, 1980.

[79] R. Missaoui, J.-M. Gagnon, and R. Godin. Mapping an extended entity-relationship schema into a schema of complex objects. In *Object-Oriented and Entity-Relationship Modelling*, pages 204–215, 1995.

[80] F. G. Mossé. Modeling roles - a practical series of analysis patterns. *Journal of Object Technology*, 1(4):27–37, 2002.

[81] J. Nachouki, M. P. Chastang, and H. Briand. From entity-relationship diagram to an object-oriented database. In T. J. Teorey, editor, *Proceedings of the 10th International Conference on Entity-Relationship Approach (ER'91), 23-25 October, 1991, San Mateo, California, USA*, pages 459–481. ER Institute, 1991.

[82] J. S. Narasimham B., Navathe S. On mapping er and relational models into oo schemas. In V. K. R.A. Elmasri and B. Thalheim, editors, *Proceedings of the 12th International Conference on Entity-Relationship Approach (ER'93), Arlington, Texas, USA*, pages 402–413. ER Institute, 1993.

[83] 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.

[84] J. Noble. Some patterns for relationships. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*. Prentice-Hall, 1996.

[85] B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 676–681, Washington, DC, USA, 2003. IEEE Computer Society.

[86] R. Pawson and R. Matthews. Naked objects: a technique for designing more expressive systems. *ACM SIGPLAN Notices*, 36(12):61–67, Dec. 2001.

[87] M. R. Pawson R. *Naked Objects, First Edition*. John Willey & Sons Ltd, 2004.

[88] F. Pennaneac'h, J.-M. Jézéquel, J. Malenfant, and G. Sunyé. Uml reflections. In *Reflection*, pages 210–230, 2001.

[89] G. J. Ramakrishnan R. *Database Management Systems, Third Edition.* McGraw-Hill, 2003.

[90] W. Reenskaug, Trygve and Lehne. *Working with objects. The OOram software engineering method*. Manning, Prentice Hall, 1996.

[91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[92] J. E. Rumbaugh. Omt: The object model. *JOOP*, 7(8):21–27, 1995.

[93] 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.

[94] E. Sciore. Object specialization. *ACM Trans. Inf. Syst.*, 7(2):103–122, 1989.

[95] 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.

[96] S. Sharoff. Philosophy and cognitive science. *Stanford Electronic Humanities Review*, 4(2), 1995.

[97] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, 1988.

[98] W. R. Smith. Newtonscript : prototypes on the palm. *Prototype-Based Object-Oriented Programming: Concepts, Languages and Applications*, pages 17–45, 1999.

[99] M. Snoeck and G. Dedene. Generalization/specialization and role in object oriented conceptual modeling. *Data Knowl. Eng.*, 19(2):171–195, 1996.

[100] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowledge Engineering*, 35(1):83–106, 2000.

[101] F. Steimann. A radical revision of UML's role concept. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 194–209. Springer, 2000.

[102] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested mixin-methods in agora. *Lecture Notes in Computer Science*, 707:197–??, 1993.

[103] P. Steyaert and W. D. Meuter. A marriage of class- and object-based inheritance without unwanted children. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 127–144, London, UK, 1995. Springer-Verlag.

[104] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel. Using uml action semantics for model execution and transformation. *Inf. Syst.*, 27(6):445–457, 2002.

[105] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[106] A. Taivalsaari. On the notion of inheritance. 28(3):438–479, Sept. 1996.

[107] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv.*, 18(2):197–222, 1986.

[108] P. Thagard. Cognitive science. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2004.

[109] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.

[110] D. Thomas, D. Hansson, L. Breedt, M. Clark, T. Fuchs, and A. Schwarz. *Agile Web Development with Rails (The Facets of Ruby Series)*. Pragmatic Bookshelf, July 2005.

[111] A. Uhl and S. W. Ambler. Point/counterpoint: Model driven architecture is ready for prime time / agile model driven development is good enough. *IEEE Software*, 20(5):70–73, 2003.

[112] 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.

[113] E. Van Paesschen, W. De Meuter, and M. D'Hondt. Selfsync: a dynamic round-trip engineering environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 190–191, New York, NY, USA, 2005. ACM Press.

[114] E. Van Paesschen, W. De 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).

[115] E. Van Paesschen, W. De 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 (to appear).

[116] E. Van Paesschen, W. De 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.

[117] E. Van Paesschen and M. D'Hondt. Selfsync: A dynamic round-trip engineering environment. In *MoDELS Satellite Events*, pages 347–352, 2005.

[118] E. Van Paesschen, M. D'Hondt, and W. De Meuter. Rapid prototyping of extended entity relationship models. In *ISIM 2005, Hradec Nad Moravici, Czech Republic, April 2005, Proceedings*, pages 194–209. MARQ, 2005.

[119] B. Verheecke and R. V. D. Straeten. Specifying and implementing the operational use of constraints in object-oriented applications.

[120] M. Vlter and T. Stahl. *Model-Driven Software Development : Technology, Engineering, Management*. John Wiley & Sons, June 2006.

[121] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, 2002.

[122] J. Wintraecken. *The NIAM Information Analysis Method: Theory and Practice.* Kluwer, The Netherlands, 1990.

[123] Agile manifesto: http://www.agilemanifesto.org.

[124] Arcstyler: http://www.arcstyler.com/.

[125] Argouml: http://argouml.tigris.org/.

[126] Atom: A tool for multi-paradigm modeling, http://atom3.cs.mcgill.ca.

[127] Core data: http://developer.apple.com/macosx/coredata.html.

[128] Core data tutorial: http://cocoadevcentral.com/articles/000085.php.

[129] Cut-based aop in ruby: http://www.rcrchive.net/rcr/show/321

[130] Dia: http://www.gnome.org/projects/dia/.

[131] Entity java beans: java.sun.com/products/ejb/.

[132] Fujaba: http://wwwcs.uni-paderborn.de/cs/fujaba/.

[133] Gershwin: http://www.csse.monash.edu.au/courseware/cse2132/gwin30495.exe.

[134] Jamda: The java model driven architecture 0.2: http://sourceforge.net/projects/jamda/.

[135] Java 2 enterprise edition: http://java.sun.com/javaee/index.jsp.

[136] jboss: http://www.jboss.org.

[137] Llblgen: http://www.llblgen.com/.

[138] Magicdraw: http://www.magicdraw.com/.

[139] Naked objects framework: http://www.nakedobjects.org.

[140] Nucleus: http://www.mentor.com/products/embedded_software/nucleus_modeling/.

[141] Object management group: the model-driven architecture: http://www.omg.com/mda/.

[142] Ocl: http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[143] Optimalj website: http://www.compuware.com/products/optimalj/.

[144] Rational: http://www-306.ibm.com/software/awdtools/developer/rosexde/.

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

[146] Simpleorm: http://www.simpleorm.org/.

[147] Spring application framework: http://www.springframework.org/.

[148] Telebib: http://www.telebib2.org/test/.

[149] Together designer: http://www.borland.com/us/products/together/index.html.

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

[151] Visio: http://www.microsoft.com/office/visio/prodinfo/default.mspx.

[152] Xmi: http://www.omg.org/technology/documents/formal/xmi.htm.

[153] P. Ziemann and M. Gogolla. Validating ocl specifications with the use tool: An example based on the bart case study. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.