Component-Based Model-Driven Software Development

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Medieninf. Jendrik Johannes
geboren am 26. März 1981 in Uelzen

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Prof. Dr. Richard Paige
(University of York, UK)

Tag der Verteidigung: Dresden, den 15. Dezember 2010

Dresden im Januar 2011

Abstract

Model-driven software development (MDSD) and component-based software development are both paradigms for reducing complexity and for increasing abstraction and reuse in software development. In this thesis, we aim at combining the advantages of each by introducing methods from component-based development into MDSD. In MDSD, all artefacts that describe a software system are regarded as *models* of the system and are treated as the central development artefacts. To obtain a system implementation from such models, they are transformed and integrated until implementation code can be generated from them. Models in MDSD can have very different forms: they can be documents, diagrams, or textual specifications defined in different modelling languages. Integrating these models of different formats and abstraction in a consistent way is a central challenge in MDSD.

We propose to tackle this challenge by explicitly separating the tasks of *defining* model components and *composing* model components, which is also known as distinguishing programming-in-the-small and programming-in-the-large[1]. That is, we promote a separation of models into models for modelling-in-the-small (models that are components) and models for modelling-in-the-large (models that describe compositions of model components). To perform such component-based modelling, we introduce two *architectural styles* for developing systems with component-based MDSD (CB-MDSD).

For CB-MDSD, we require a universal composition technique that can handle models defined in arbitrary modelling languages. A technique that can handle arbitrary textual languages is *universal invasive software composition*[2] for code fragment composition. We extend this technique to *universal invasive software composition for graph fragments* (U-ISC/Graph) which can handle arbitrary models, including graphical and textual ones, as components. Such components are called *graph fragments*, because we treat each model as a typed graph and support reuse of partial models.

To put the composition technique into practice, we developed the tool REUSEWARE that implements U-ISC/Graph. The tool is based on the Eclipse Modelling Framework[3] and can therefore be integrated into existing MDSD development environments based on the framework.

To evaluate the applicability of CB-MDSD, we realised for each of our two architectural styles a model-driven architecture with REUSEWARE. The first style, which we name ModelSoC, is based on the component-based development paradigm of *multi-dimensional separation of concerns*[4]. The architecture we realised with that style shows how a system that involves multiple modelling languages can be developed with CB-MDSD. The second style, which we name ModelHiC, is based on *hierarchical composition.* With this style, we developed abstraction and reuse support for a large modelling language for telecommunication networks that implements the Common Information Model[5] industry standard.

[1]DeRemer, F., Kron, H.: Programming-in-the-large versus programming-in-the-small. (1975)

[2]Henriksson, J.: A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. PhD thesis, Technische Universität Dresden. (2009)

[3]Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. (2009)

[4]Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. (2000)

[5]DMTF: Common Information Model Standards. `http://www.dmtf.org/standards/cim`. (2010)

Acknowledgements

I thank my supervisor Uwe Aßmann for giving me the opportunity to work on this interesting and challenging topic and for all the advices he gave me. My thanks go to the whole software technology group at TU Dresden, who provided a working atmosphere in which it was fun working on this thesis and on many other interesting research challenges. Furthermore, I thank all my colleagues from around Europe with whom I had the pleasure to work together in the Modelpex project. Without the project and the input I got from many of them, this thesis would not have been possible.

Although I had to go the final steps of finishing this thesis alone, many people supported me along the way to whom I like to extend a special thanks. First of all, there are Jakob Henriksson and Ilie Şavga who introduced me to the work on invasive software composition back in 2005 when I was still an undergraduate student and with whom I started to work on a composition tool which eventually became the Reuseware tool. Without Jakob, who supervised me as a student and soon became a good friend, this work would probably have never started. Steffen Zschaler supervised me during the first two years of my work. He taught me a lot about research, project management, writing publications, and a thousand other things a researcher should know about. Without him, I would have had a much harder start.

When I started working on my thesis in winter 2006, I was accompanied by colleagues who started shortly before or after me. It turned out that research and hacking were not the only interests we had in common and soon we became good friends. In 2007, Florian Heidenreich, Steffen, and me worked intensively on transferring invasive software composition to modelling. The endless discussions we had about (meta)modelling, slots, hooks, and other scary things were a lot of fun and laid the foundations for this thesis. Another impulse for my work was given in 2008, when I tried to separate out the grammar-processing part from Reuseware into a separate tool. Sven Karol volunteered to participate in that and did a massive improvement of my really ugly code. Then, Mirko Seifert took over and added a lot of improvements. At the same time, Christian Wende and Florian also got interested in using the tool and contributed to it. Finally, we coined the tool EMFText and released it in the beginning of 2009. We wrote several publications together based on the work on EMFText which gave us a common ground to work on. Without this work as foundation, this thesis would be missing quite some pages; and without our meetings I would be missing quite some fun in my life. Mirko also supported me at the end of Modelplex which gave me time to concentrate on writing.

There are many more people in our group who supported me in one way or the other. Jan Polowinski, who is a good friend since we studied Medieninformatik together, often showed me that it is also important to relax from time to time (unfortunately, I too seldom followed this advice). Sebastian Cech joined Modelplex halfway through and took over a huge amount

This research has been co-funded by the European Commission within the 6th Framework Programme (FP) project Modelplex contract number 034081 (cf. `http://www.modelplex.org`).

of work from me which allowed me to concentrate on the model composition work. Matthias Schmidt was supervised by me for his diploma and joined the Modelplex team after that. We had many interesting discussions from which I highly profited. Matthias and Karsten Gaul, whom I supervised during his diploma and also before that, both contributed additional features to the Reuseware tooling which helped me in my experimentations. Rosi Pjater, whom I shared a office with, endured my tempers over the years and always found some motivating words for me. Andreas Bartho, Henrik Lochmann, Birgit Grammel, Konrad Voigt, Julia Schröter, Simone Röttger, Katja Siegemund, Christoff Bürger, and Claas Wilke all helped here and there over the time by giving advice and motivation. In my last weeks of writing, Sebastian Richly and Sebastian Götz took over some work, which I was supposed to do, without complaining, for which I am thankful. Katrin Heber always helped me with administrative issues; in particular with issues concerning Reisekostenabrechnungen.

Half of my research life was taking place in the Modelplex project, where I had the pleasure to work with many interesting people from different backgrounds. In the beginning of the project, I had intensive discussions with Marcos Didonet Del Fabro, Hugo Brunelière, Jean Bézevin, Dimitris Kolovos, and Steven Völkel about model compositions, model weaving, and aspect-oriented modelling (and the meaning of all these terms), which helped me to find a direction for my work. Later, I worked with Dimitris, Richard Paige, and Steffen on using model composition for abstraction, which inspired me to the work on composition program extraction which was a missing piece in my work until then. Miguel Fernández and his colleagues at Telefónica gave me the opportunity to work on a real problem and some real material which fueled my work. Without that, and their kind support, I would have never come that far. It was always a pleasure to discuss and share experiences with Mathias Fritzsche, who was also writing his thesis in the context of Modelplex in a similar time frame as me.

I would like to extend a special thanks to all the people who, often on short notice, commented on parts of my thesis: Birgit, Julia, Ilie, Christian, Christoff, Mirko, and Jan; and a very special thanks to Annika, Karsten, and Sven who took the time to read and comment on a complete draft of the document.

Finally, but not lastly, I want to thank my family: my parents Ursula and Wilhelm, my sister Julia and her fiancé Marc, my sister Jana and her husband Cornelius, and my grandmother Oma Helga. They all always encourage and supported me. The biggest thank goes to my love Annika who not only encouraged and supported me even in situations that for me seemed to be past hope, but who loved me despite all my little and big flaws and my occasional workaholic behaviour – I love you.

Jendrik Johannes
Dresden, September 2010

Publications

The central topic of this thesis is partially based on the following peer-reviewed publications. In co-authored publications, the author contributed a large part and in particular the parts this thesis is based on.

- Jendrik Johannes, and Uwe Aßmann. *Concern-based (de)composition of Model-Driven Software Development Processes.* In Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), volume 6395 of LNCS, pages 47–62. Springer, October 2010. *(conference paper)*
- Jendrik Johannes, and Miguel A. Fernández. *Adding Abstraction and Reuse to a Network Modelling Tool using the Reuseware Composition Framework.* In Proceedings of 6th European Conference on Modelling Foundations and Applications (ECMFA 2010), volume 6138 of LNCS, pages 132–143. Springer, June 2010. *(conference paper)*
- Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. *On Language-Independent Model Modularisation.* In Transactions on Aspect-Oriented Software Development VI, volume 5560 of LNCS, pages 39–82. Springer, October 2009. *(journal article)*
- Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. *Extending Grammars and Metamodels for Reuse – The Reuseware Approach.* In IET Software Special Issue on Language Engineering, volume 2(3) of IET Software, pages 165–184. IET, June 2008. *(journal article)*
- Jendrik Johannes. *Controlling Model-Driven Software Development through Composition Systems.* In Proceedings of the 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE 2009), pages 240–253. Tampere University of Technology, August 2009 *(workshop paper)*
- Jendrik Johannes. *Developing a Model Composition Framework with Fujaba – An Experience Report.* In Proceedings of the 7th International Fujaba Days, pages 1–4. Technische Universiteit Eindhoven, November 2009. *(workshop paper)*
- Florian Heidenreich, Jendrik Johannes, and Steffen Zschaler. *Aspect-Orientation for Your Language of Choice.* In Proceedings of the 11th International Workshop on Aspect-Oriented Modeling (AOM@MoDELS 2007). www.aspect-modeling.org, October 2007. *(workshop paper)*

The following peer-reviewed publications, to which the author contributed a large part, cover research that was required as prerequisite for the central topic of this thesis.

– Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert and Christian Wende. *Derivation and Refinement of Textual Syntax for Models.* In Proceedings of 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), volume 5562 of LNCS, pages 114–129. Springer, June 2009. *(conference paper)*
– Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende. *Closing the Gap between Modelling and Java.* In Proceedings of 2nd International Conference on Software Language Engineering (SLE 2009), volume 5969 of LNCS, pages 374–383. Springer, March 2010. *(conference paper)*

The following peer-reviewed publications, to which the author contributed a large part, cover research that builds upon the results of this thesis but is not covered in the central topic.

– Jendrik Johannes, Steffen Zschaler, Miguel A. Fernández, Antonio Castillo, Dimitrios S. Kolovos and Richard F. Paige. *Abstracting Complex Languages through Transformation and Composition.* In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), volume 5795 of LNCS, pages 546–550. Springer, October 2009. *(conference paper)*
– Jendrik Johannes, Roland Samlaus and Mirko Seifert. *Round-Trip Support for Invasive Software Composition Systems.* In Proceedings of International Conference on Software Composition 2009 (SC 2009), volume 5634 of LNCS, pages 90–106. Springer, June 2009. *(conference paper)*
– Matthias Schmidt, Jan Polowinski, Jendrik Johannes and Miguel A. Fernández. *An Integrated Facet-based Library for Arbitrary Software Components.* In Proceedings of 6th European Conference on Modelling Foundations and Applications (ECMFA 2010), volume 6138 of LNCS, pages 261–276. Springer, June 2010. *(conference paper)*
– Jendrik Johannes and Karsten Gaul. *Towards a Generic Layout Composition Framework for Domain Specific Models.* In Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM 2009), pages 113–118. HSE Print, October 2009 *(workshop paper)*

Contents

List of Figures

List of Listings

1 Introduction

Model-driven software development (MDSD) is a software development paradigm that leverages *models* as the premier artefacts of software development. Models are specifications that describe systems at different levels of abstraction, including abstractions of the domain of the customer for whom a software system is developed. By this, complexity is reduced and separation of concerns is improved. While system design with models is a common practice in software engineering, models are often discarded in the implementation phase of the system, which leads to inconsistencies between system design and implementation. In MDSD, however, models have precedence over implementation code: They are input to transformation, generation, and composition engines, which generate system implementations from models and therewith automate the transfer from system design to system implementation.

This principle of *model-driven*, as opposed to code-centric, development was promoted through the Model Driven Architecture (MDA) [OMG01] introduced by the Object Management Group (OMG) [OMG10a] in 2001. The utilisation of a model-driven architecture promises advances in different critical fields of software system engineering [OMG01]:

> "This is the promise of Model Driven Architecture: to allow definition of machine-readable application and data models which allow long-term flexibility of implementation [. . .], integration [. . .], maintenance[. . .], testing and simulation [. . .]."
>
> – *Object Management Group. MDA Guide [OMG01]*

For instance [OMG01], a model-driven architecture allows for generating different implementations from a common set of models and thus new platform technologies (e.g., new programming languages) can be targeted with existing designs. Furthermore, since the design of systems, and not only their implementations, are available in machine-readable form, system integration can be automated and maintenance becomes simpler. Design models can then, similar to code, be tested and validated against requirements.

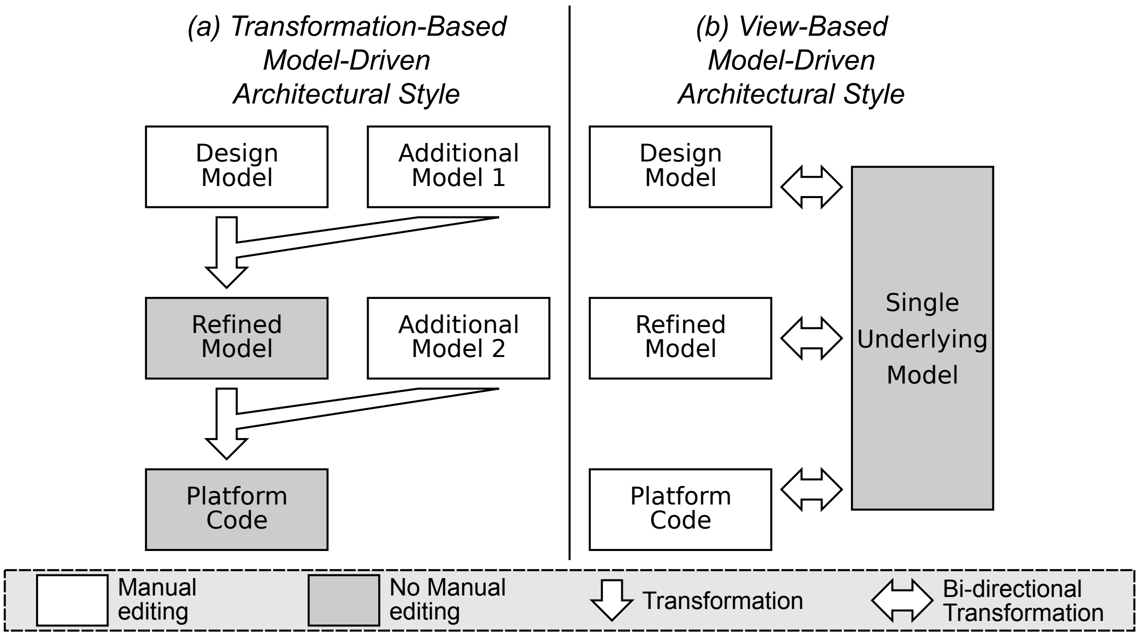


Figure 1.1: Two existing architectural styles for MDSD: (a) transformation-based style (inspired by [OMG01, Figure 2.2]), (b) view-based style (inspired by [AS08, Figure 1]).

A model-driven architecture is a special type of software architecture [SG96] that follows the MDSD paradigm by adhering to the following two key concepts. First, a model-driven architecture offers developers machine-readable modelling languages with abstractions appropriate for the domain of the system under construction. Second, a model-driven architecture utilises refinement techniques that automatically extract information from models created with such languages and transfer this information into the implementation of the modelled system.

The OMG initially proposed the Unified Modelling Language (UML) [OMG10b] as standard language for model-driven architectures. It turned out however, that one standard language is not sufficient to model arbitrary systems for various customer domains and that domain-specific languages [vDKV00] are needed within model-driven architectures [KBJV06]. A domain specific-language used in a model-driven architecture is hence called *domain-specific modelling language (DSML)* [Kle08]. To support the usage of DSMLs in model-driven architectures as alternatives or in addition to UML, the OMG standardised the way to define DSML language concepts in the Meta Object Facility (MOF) standard [OMG06a]—such DSML definitions are called *metamodels*.[1]

The automatic refinement technique used by a model-driven architecture dominates the general structure of the architecture. The choice of refinement technique can thus be regarded as choice of *architectural style* [SG96]. The architectural style proposed by the OMG in the MDA guide [OMG01] defines a transformation-based style illustrated in Figure 1.1a. In this style,

[1]For simplicity, we refer to any language as DSML for which a MOF specification of the language structure exist. This includes large modelling languages as UML, but also programming languages, such as Java, if their structure is defined with MOF.

models are automatically transformed into refined models, which are eventually transformed into an implementation. However, additional information is required for each automated refinement, which has to be provided through additional models (Figure 1.1a). Although this style of model-driven architecture defined by the OMG is often referred to as *the* (one and only) Model-Driven Architecture, it can be argued that any architecture that follows the MDSD paradigm is *a* model-driven architecture. Then, other architectural styles are possible. One such style is proposed by Atkinson et. al in [AS08] for the KobrA modelling method [ABB^{+}01]. This *view-based* model-driven architectural style, illustrated in Figure 1.1b, uses a single underlying model that holds all information about the system and treats the models developers work with as views on the underlying model—including the implementation code. Bi-directional transformations are responsible for creating models from and reintegrating models into the single underlying model (Figure 1.1b). This shows that different architectural styles for model-driven architectures are possible and, in certain situations, more suitable.

The shown architectural styles have drawbacks which makes them difficult to use in practical software development; we discuss this in the following. We believe that many problems are caused by a lack of *component support* in MDSD. Although, components are a central concept in software architectures in general [SG96]. In this thesis, we therefore develop a composition technique for MDSD and styles for component-based model-driven architectures.

1.1 Problem: Consistency and Information Tracing in MDSD

"A central tenet of MD[S]D is that there are multiple representations of artefacts inherent in a software development process, representing different views of or levels of abstraction on the same concepts. [. . .] The basic problem is that the introduction of multiple, interrelated representations implies the issue of assuring their mutual consistency—a very difficult problem."

– *Hailpern, Tarr.*
Model-driven development: The good, the bad, and the ugly [HT06, p. 456]

Hailpern and Tarr essentially state that redundancy is inherent in MDSD because similar information is defined, or transferred between, multiple artefacts and that therefore consistency between these artefacts needs to be assured which is "a very difficult problem". In fact, consistency problems occur during two activities, illustrated in Figure 1.2: (1) during the set up of a model-driven architecture in terms of defining DSMLs and transformations, which is performed by a *process architect*, and (2) during the development of a system with the model-driven architecture, which is performed by a *system designer.*

In the first case, the process architect has to integrate the DSMLs and transformations such that all information can be defined in separate models, as suggested in Figure 1.1a, and that manual refinement of generated models or code is not necessary. This is a task which is often not successful because it is difficult to grasp which features from the target platform should be abstracted and which not. An example is that developers often experience the need to manually refine generated code since it is not possible to define each information in a separate model. In such a case, the architecture should be refined. However, such evolutions of model-driven architectures are not well supported by the transformation-based style.

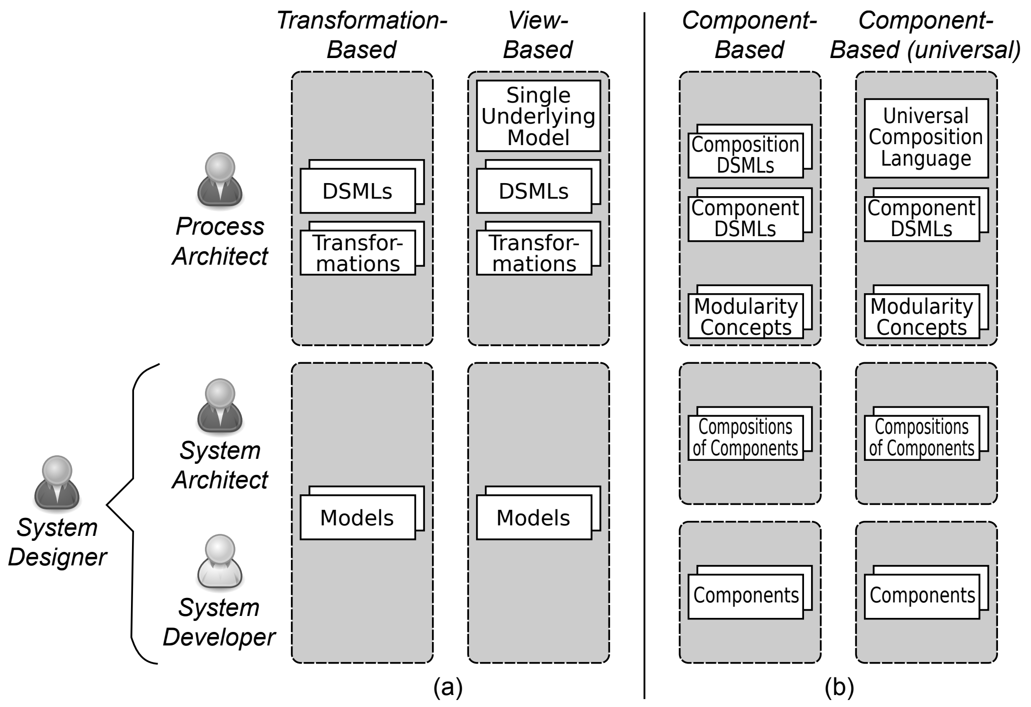


Figure 1.2: Roles and their responsibilities in developing and using model-driven architectures of different styles.

In the second case, the system designer has to cope with the problem of tracing information in the complex scenario where multiple transformations replicate and scatter similar information. For examples, developers often modify generated code when they find an error by debugging the running system, not because it is impossible to change the information in a model, but because they do no know where this model is located. Locating a model that is responsible for a piece of generated code is especially difficult if the code was generated by a chain of transformations that extracted information from different places in different models.

Since in both cases there is knowledge missing about where which information is needed and where which information is combined, we see the root problem in an insufficient separation of concerns and in particular in an insufficient separation between *components* and *component composition*, which should be a fundamental principle in each software architecture. This separation of component implementations and component composition is also known as distinguishing between *programming-in-the-small* and *programming-in-the-large*, one of the foundations of software architecture, introduced by DeRemer and Kron [DK75] who state:

> "We argue that structuring a large collection of modules [i.e., components] to form a system is an essentially distinct and different intellectual activity from that of constructing the individual modules [i.e., components]. That is, we distinguish programming-in-the-large from programming-in-the-small."
>
> – *DeRemer, Kron. Programming-in-the-Large versus Programming-in-the-Small [DK75, p. 1]*

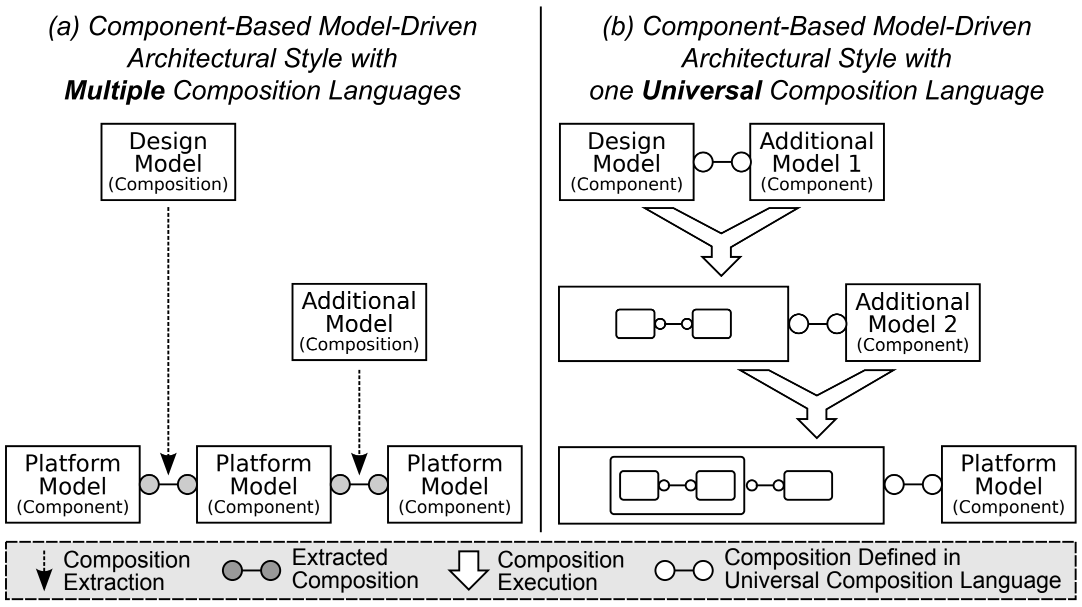

(a) Component-Based Model-Driven Architectural Style with Multiple Composition Languages
Design Model (Composition)
Additional Model (Composition)
Platform Model (Component)
Platform Model (Component)
Platform Model (Component)
(b) Component-Based Model-Driven Architectural Style with one Universal Composition Language
Design Model (Component)
Additional Model 1 (Component)
Additional Model 2 (Component)
Platform Model (Component)
Composition Extraction
Extracted Composition
Composition Execution
Composition Defined in Universal Composition Language

Figure 1.3: Proposals for a component-based model-driven architectural styles: (a) with multiple DSMLs and (b) with one DSML and a universal composition language.

Bézevin et. al argue that distinguishing these two development activities is also important for MDSD and introduced the terms *modelling-in-the-large* and *modelling-in-the-small* [BJRV05]. They do, however, not discuss the separation of modelling-in-the-large and modelling-in-the-small in model-driven architectures directly. We believe that this separation requires more attention to solve the consistency and traceability issues.

A language for programming-in-the-large in software architecture is called architectural description language, which is a special type of *composition language* [Aßm03]. In contrast, a language for programming-in-the-small, which is used to implement components, is a *component language*. In model-driven architectures, however, there is so far no separation of composition and component languages. In the transformation-based (Figure 1.1a), and also the view-based (Figure 1.1b) style, there is only a separation between models and transformations, but no separation between models that are components (defined in component languages) and models that are compositions of components (defined in composition languages).

To illustrate this, let us again consider the roles different developers play when they work with a model-driven architecture. In addition to the two roles of process architect and system designer discussed above, Figure 1.2 splits the system designer role further into *system architect* and *system developer*. The system architect corresponds to the classical software architect of a non-model-driven architecture. He or she is responsible for modelling the system's structure by defining composition of components, which we call *composition programs*, in composition languages. The system developer is responsible for defining components with a component language. Since there is no distinction between composition and component languages, the system architect and system developer roles are not easy to separate.

We believe that these two roles should be separated and that we should distinguish between DSMLs that are component languages and DSMLs that are composition languages. In such a *component-based* architectural style, the process architect would decide which DSMLs are component and which are composition languages as suggested in Figure 1.2b. Instead of defining transformations, he or she defines modularity concepts through which composition and component languages relate (Figure 1.2b). System architects are then responsible for defining the architecture of a system by using the composition language DSMLs and system developers use the component language DSMLs to model or implement components.

Our suggestion for two component-based model-driven architectural styles are sketched in Figure 1.3. Both styles require a *universal composition language* to represent the complete system structure. This composition language can either be used as unifying format for all composition language DSMLs (Figure 1.2b left; Figure 1.3a) or directly as single universal composition language (Figure 1.2b right; Figure 1.3b). The former case allows for customisation, while the latter allows for reuse of the composition language.

Our hypothesis is that a combination of MDSD and component-based development to design model-driven architectures with the suggested styles improves the separation of concerns in MDSD and therewith the management of consistency and information tracing. In this thesis, we therefore develop a universal composition technique for models and a tool based on that for component-based MDSD (CB-MDSD). CB-MDSD combines the advantages of MDSD—domain abstraction in DSMLs and automated refinement—and component-based development—separation of concerns and reuse of components. We use CB-MDSD to define concrete component-based architectural styles for MDSD and use these styles to implement model-driven architectures through which we evaluate our hypothesis.

1.2 Thesis Contributions and Structure

The overall contribution of this thesis is *component-based model-driven software development* (CB-MDSD) illustrated in Figure 1.4. CB-MDSD is founded on a universal composition technique for models (Figure 1.4 left), a tool implementing this technique (Figure 1.4 middle), and two component-based architectural styles for MDSD (Figure 1.4 right). The first style, called ModelSoC[2], transfers the concept of *multi-dimensional separation of concerns* [OT00] to MDSD and can be used to implement component-based model-driven architectures that, similar to traditional transformation-based architectures, allow the inclusion of arbitrary many DSMLs as composition as well as component languages. The second style, called ModelHiC[3], is based on *hierarchical composition* and restricts an architecture to include a single predefined composition language—the Universal Composition Language (UCL). In cases where this restriction is acceptable, setting up an architecture with ModelHiC requires less effort as doing so with ModelSoC.

To illustrate the aforementioned problem and our solution, we first introduce a motivating example in Chapter 2 which we use throughout the thesis. The remaining structure of the document is oriented at our three major contributions, which correspond to the three columns in Figure 1.4. They are summarised in the following.

[2]Component-based model-driven architectures with multi-dimensional separation of concerns
[3]Component-based model-driven architectures with hierarchical composition

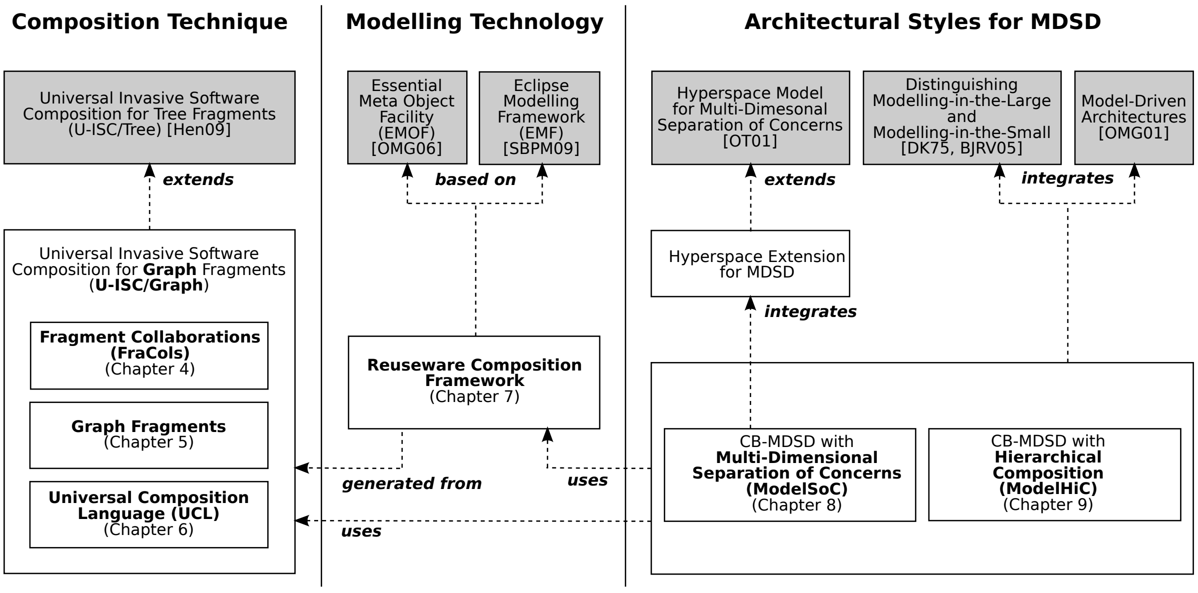

Composition Technique
Modelling Technology
Architectural Styles for MDSD
Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree) [Hen09]
extends
Universal Invasive Software Composition for Graph Fragments (U-ISC/Graph)
Fragment Collaborations (FraCols) (Chapter 4)
Graph Fragments (Chapter 5)
Universal Composition Language (UCL) (Chapter 6)
Essential Meta Object Facility (EMOF) [OMG06]
Eclipse Modelling Framework (EMF) [SBPM09]
based on
Reuseware Composition Framework (Chapter 7)
generated from
uses
uses
Hyperspace Model for Multi-Dimesonal Separation of Concerns [OT01]
Distinguishing Modelling-in-the-Large and Modelling-in-the-Small [DK75, BJRV05]
Model-Driven Architectures [OMG01]
extends
integrates
Hyperspace Extension for MDSD
integrates
CB-MDSD with Multi-Dimensional Separation of Concerns (ModelSoC) (Chapter 8)
CB-MDSD with Hierarchical Composition (ModelHiC) (Chapter 9)

Figure 1.4: Overview of CB-MDSD and our contributions.

1.2.1 Contribution 1 (C1): Composition Technique for Typed Graphs (Part I)

In CB-MDSD, models are composed at development time to obtain new integrated models and implementations of the modelled system. Therefore, a static composition technique is needed that can compose models by merging them. Since models that are defined in a DSML specified with MOF are typed graphs, we introduce a composition technique that treats typed graphs as components.

Our composition technique is an extension of Universal Invasive Software Composition (U-ISC) introduced by Henriksson [Hen09]. With U-ISC, one can design component support for textual (domain-specific) languages that are described by context-free grammars [Cho56]. Components in U-ISC, called *fragments*, are trees obtained by parsing textual artefacts. Composition is performed by merging these trees. We base our work on U-ISC, because we require a method to define component support for domain-specific languages which U-ISC is. However, U-ISC is limited to textual languages and artefacts that are represented as trees. It can thus not be applied on arbitrary, possibly graphical, DSMLs with models that are represented as graphs. We therefore extend U-ISC to work with graphs and call this new approach Universal Invasive Software Composition for Graph Fragments (U-ISC/Graph) and refer to Henriksson's approach [Hen09] as Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree). U-ISC/Tree is introduced in more detail in Chapter 3. The complete semantics of U-ISC/Graph are defined with Story Driven Modelling (SDM) [FNTZ00] in Appendix A.

U-ISC/Graph is defined in Chapters 4–6 of Part I. Each of these chapters presents a major advance over U-ISC/Tree, which are: In Chapters 4 we introduce a method to explicitly define modularity concepts for fragments called *fragment collaborations* (FraCols). U-ISC/Tree does not include such a method, but proposes the introduction of it as future work [Hen09, Section 8.1]. Chapters 5 introduces support for typed graphs as fragments and with that support for languages that are defined with EMOF[4]. In Chapters 6 we define a universal composition language (UCL) that is independent of component languages and modularity concepts. U-ISC/Tree does not define a universal composition language. It only provides an abstract base language [Hen09, Section 2.2.2] that needs to be integrated with each new component language.

1.2.2 Contribution 2 (C2): Composition Framework for Models (Chapter 7)

While Part I defines U-ISC/Graph as a graph composition technique, Part II is concerned with utilising U-ISC/Graph in practical MDSD. It starts with Chapter 7 in which we present the Reuseware Composition Framework[5] (short REUSEWARE)—a tool that integrates U-ISC/Graph into the Eclipse Modelling Framework (EMF) [SBPM09], which is an EMOF-based MDSD environment. REUSEWARE realises the concepts of U-ISC/Graph (Contribution 1) which includes tooling to specify modularity concepts as FraCols and to map them to DSMLs defined in EMOF. Furthermore, it contains tooling to specify composition programs in UCL.

REUSEWARE was itself developed in a model-driven way. For this, the specifications of U-ISC/Graph, which are defined using EMOF and SDM, are used as input to code generators provided by EMF and the Fujaba toolsuite [NNZ00, GBD07]. The integration in EMF allows us to reuse existing EMF-based tools to create DSMLs and models with these DSMLs. Thus, with REUSEWARE, CB-MDSD, with U-ISC/Graph as underlying composition technique, can be used in practice.

1.2.3 Contribution 3 (C3): Component-Based Model-Driven Architectures (Chapters 8/9)

Finally, we present two architectural styles for MDSD—ModelSoC and ModelHiC. Both styles are introduced and evaluated. In the case of ModelSoC (Chapter 8), we explain how multiple modularity concepts, which are often required when multiple DSMLs are combined, can be handled in parallel by U-ISC/Graph. With this we realise the *hyperspace model* for *multi-dimensional separation of concerns* [OT00] and extend it for the specifics of MDSD. We revisit and extend the example from Chapter 2 and show that a component-based model-driven architecture with multiple languages can be implemented with CB-MDSD that resolves the consistency problems of a corresponding transformation-based architecture.

We used ModelHiC (Chapter 9) to develop a modelling environment for a large telecommunication DSML. For this, we present an iterative development process to find suitable abstractions in CB-MDSD. This work, carried out in collaboration with telecommunication experts at Telefónica, shows how a new model-driven architecture with hierarchical composition can be set up by using CB-MDSD with less specification effort as by using transformations to achieve the same abstraction capabilities.

[4]Essential MOF [OMG06a]; the MOF core concepts
[5]`http://www.reuseware.org`

Part I

Universal Invasive Software Composition for Graph Fragments (U-ISC/Graph)

2 Motivating Examples and Requirements

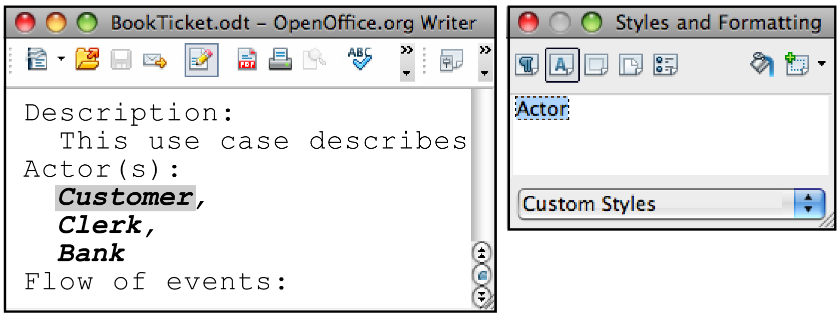
This chapter demonstrates CB-MDSD by examples. From these examples, requirements for our composition technique U-ISC/Graph are derived.

To illustrate the goals of our work, this chapter introduces two examples of component-based MDSD (CB-MDSD). Using these examples, we illustrate the central concepts of CB-MDSD and derive requirements we need to address to realise CB-MDSD in practice.

The first example—introduced in Sections 2.1—is a model-driven architecture that uses multiple DSMLs for the development of a reservation system in which customers can book tickets and perform other related activities. It is inspired by an example from Roussev and Wu [RW07]. Five DSMLs[1] are used: textual use case descriptions (defined in OpenDocument format [OAS07]), UML use case diagrams [OMG10b] annotated with invariants (as introduced in [RW07]), UML class diagrams [OMG10b], a DSML for defining dataflow (not used in this chapter; will be introduced in Chapter 8), and Java. This example illustrates the ideas of ModelSoC which we define in Chapter 8.

The second example—introduced in Section 2.2—centres around a single object-oriented DSML. We use two variants of the examples, one with UML as DSML and one with Java as DSML. The example illustrates the usage of a universal composition languages in CB-MDSD, which is the central idea of ModelHiC discussed in Chapter 9. Both examples are used throughout this thesis for demonstration purposes and the first example is extended in Chapter 8 to evaluate ModelSoC.

[1]We refer to all software languages that are defined by an EMOF metamodel as DSMLs. In this case this includes UML, OpenDocument, Java, and two DSMLs we defined for the purpose of this example. More details about of how these languages are defined with EMOF are discussed in Chapter 8.



```
Description:
  This use case describes
Actor(s):
  Customer,
  Clerk,
  Bank
Flow of events:
```

Figure 2.1: *BookTicket* use case description.

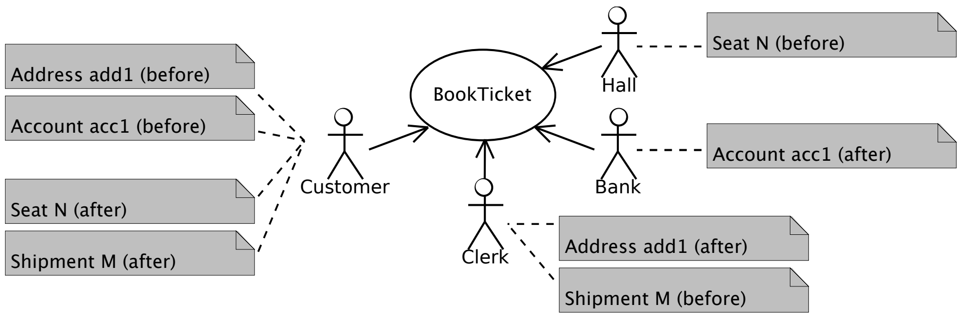


Figure 2.2: *BookTicket* use case diagram with annotated value added invariants.

2.1 CB-MDSD with Multiple DSMLs

In this model-driven architecture, development starts with textual use case descriptions. For each use case, a document in OpenDocument format is created that includes a description of the use case and enumerates the actors participating in the use case (cf. [RW07] for more details on the document format). The description of the use case *BookTicket* is shown in Figure 2.1. While it is hard to process arbitrary free text in the description automatically, the document clearly states which actors participate in the use case by using a dedicated custom style to mark actors in the text. This information must be extracted from the document to be integrated into other models—for instance into a UML use case diagram as shown in Figure 2.2.

This representation as UML use case diagram is needed, because the example refines the use cases by annotating use case diagrams with *value added invariants* as introduced by Roussev and Wu [RW07]. This means, that we define for each actor *business values* it holds before and after the execution of the use case. The total number of business values in a use case needs to be invariant (i.e., a value that exists before use case execution needs still to be there after execution and a value cannot appear out of nowhere).

In the *Book Ticket* use case for instance (Figure 2.2), the Customer has an Account and an Address before execution of the use case and a Seat and a Shipment after execution. The Address is passed to the Clerk, while the Account is passed to the Bank. The Shipment is received from the Clerk. However, no actor owns the Seat before use case execution. This is a violation of the invariant, the developer has to correct. According to [RW07], these cases

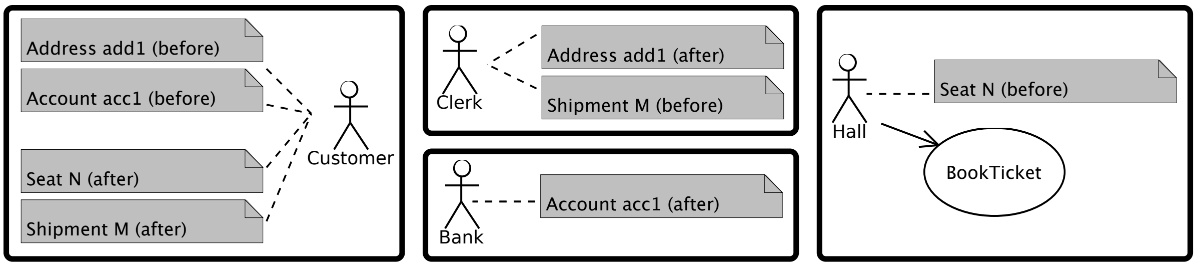


Figure 2.3: Information needed for an automatic refinement from *BookTicket* use case description (cf. Figure 2.1) to annotated use case diagram (cf. Figure 2.2).

may occur if there are actors not visible from outside the system (e.g., a passive storage that cannot be observed as acting entity from the outside). For such cases, they propose to add an additional actor to the UML use case diagram (but not to the textual use case descriptions). In this case, the actor Hall is added, which owns the Seat before use case execution (Figure 2.2 top right), which was not present in the textual use case description (Figure 2.1).

In a simple transformation-based solution, one can transform the textual use case description into a use case diagram without annotations and then refine it manually. However, this forbids future modifications of the textual use case descriptions, since re-executing the transformation would override the manual refinements in the UML use case diagrams. To allow for this, the generated and manually added information needs to be separated. Figure 2.3 shows the parts of the use case diagram for *BookTicket* that were added manually. For each actor, the additional information can be defined in a separate model. In the case of the new actor Hall (Figure 2.3 right), we also need to state that it is an actor participating in the *BookTicket* use case, since this information is not available in the textual use case description (Figure 2.1) which only knows about the actors Customer, Clerk, and Bank. A model transformation can then be defined that takes the textual use case description (Figure 2.1) and the additional information (Figure 2.3) as input and produces an integrated use case diagram (Figure 2.2).

While such transformations can be defined with current model transformation technology, the situation described above is a pattern one would have to apply thoroughly and consistently to all transformations in a model-driven architecture, which would lead to recurring effort in transformation design. Therefore, our suggestion for CB-MDSD manifests this pattern using concepts of component-based design. The main requirements for this are illustrated on the example in the following.

2.1.1 Switching Formats

The first issue to consider is the automatic transformation of information into different formats. The information about actors participating in a use case, for example, is required in all formats of our model-driven architecture (i.e., it needs to be represented in all DSMLs). To automatically transfer information from one format into another, two steps are required: (1) extraction of information from the model in the original format and (2) composition of information into an integrated model of the new format. A drawback of transformation-based architectures (cf. Figure 1.1a) is that model transformations in general do not distinguish these two steps which

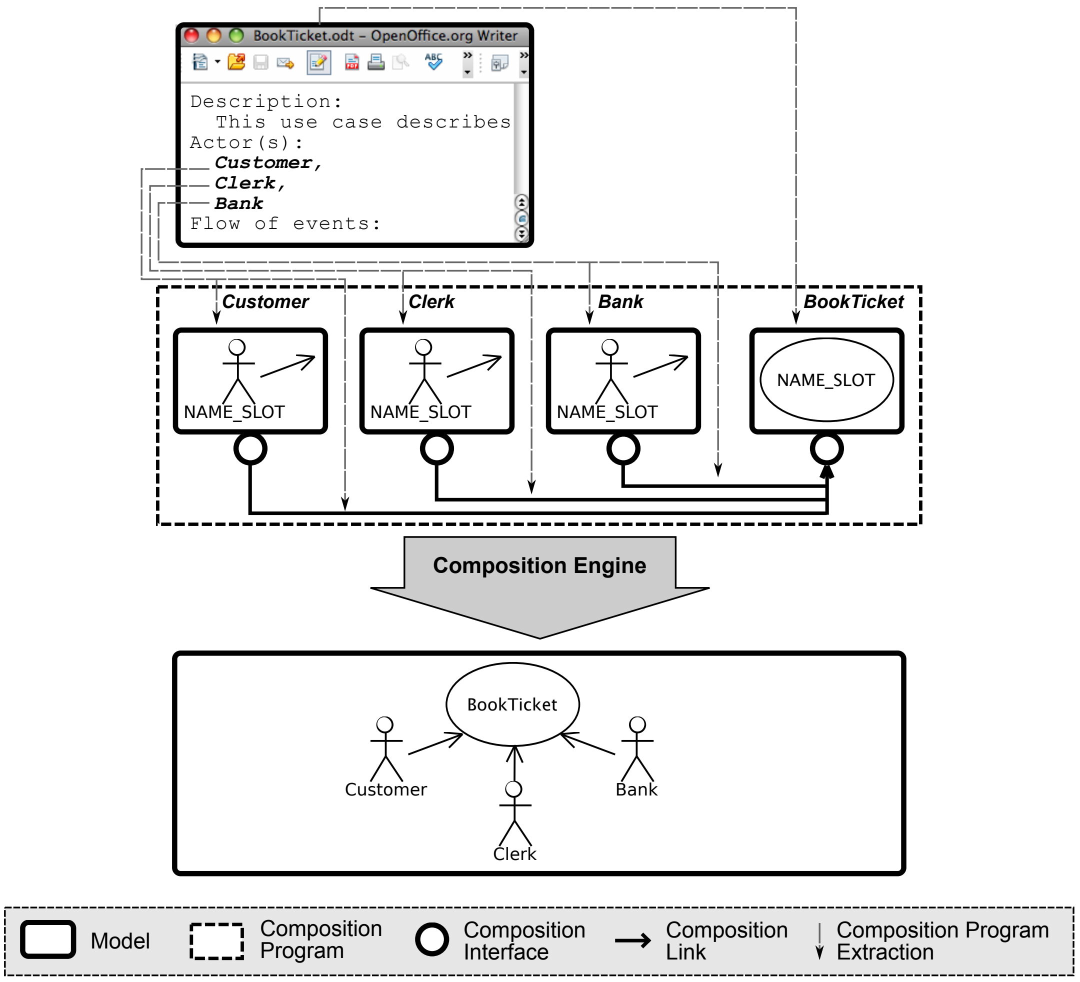
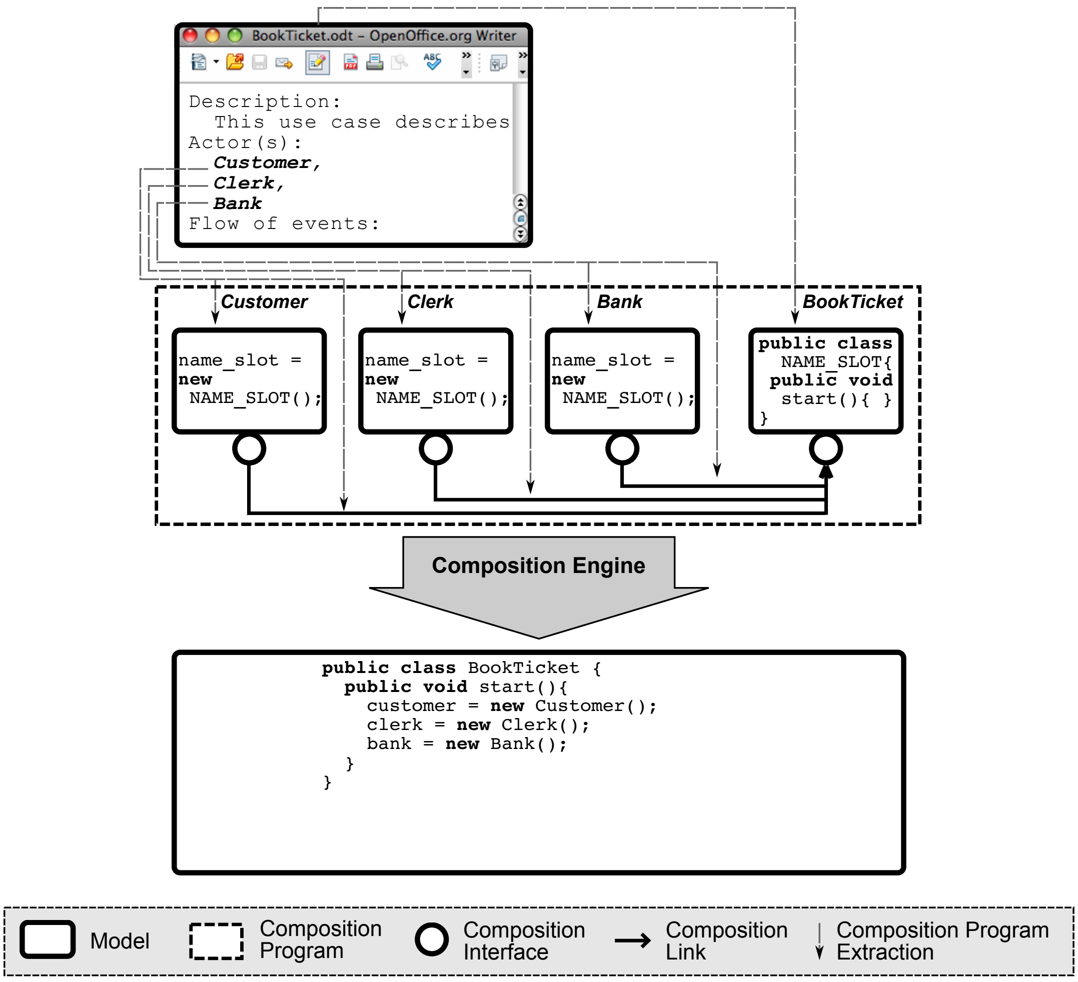


Figure 2.4: OpenDocument as composition and UML use cases as component language.

leads to situations where similar information is extracted several times. For example, the information about actors, which is extracted from the textual use case descriptions, would have to be extracted again from the use case diagrams if these diagrams would be used as the base for further refinement (e.g., to generate Java code from them). Another problem originating from this is tracing of information. When a particular information about the system has to be changed, one needs to know, where the information is defined. For example, if one discovers in the Java code that something about the actor Hall should be changed, it is difficult to trace whether Hall was introduced in a textual use case description or a UML use case diagram, because the information has been transformed several times.

To improve this situation, we suggest to treat the transformation of models as composition of model components. Figure 2.4 illustrates this for generating a use case diagram from a textual use case description. Here, the source DSML (in this case OpenDocument) is treated as a *composition language* and the target DSML (in this case UML use cases) is treated as a *component language*. From the source model (in this case the textual use case description) a



```
Description:
  This use case describes
Actor(s):
  Customer,
  Clerk,
  Bank
Flow of events:
```

```
name_slot =
new
 NAME_SLOT();
```

```
name_slot =
new
 NAME_SLOT();
```

```
name_slot =
new
 NAME_SLOT();
```

```
public class
  NAME_SLOT{
 public void
  start(){ }
}
```

```
public class BookTicket {
  public void start(){
    customer = new Customer();
    clerk = new Clerk();
    bank = new Bank();
  }
}
```

Figure 2.5: UML use cases exchanged for Java as component language.

composition program is extracted. A composition program describes a system by identifying, linking, and parameterising components. In this case, the components are fragments of UML use case diagrams (single Actors and single UseCase) that are configured with the names of the actors (`Customer`, `Clerk`, `Bank`) and the name of the use case (`BookTicket`) which are extracted from the textual use case description. Given the composition program and the components, a composition engine produces the integrated target model (in this case a UML use case diagram).

We now have a distinction between (1) extraction of information—which is the extraction of a composition program from the source model—and (2) composition of information—which is the composition of the target model performed by a composition engine interpreting the composition program on model components defined in the target language. This composition accesses internals of the model components through composition interfaces, which have two purposes. First, they have to give access to internals of the models for modification. For instance, the Actor component contains a placeholder for the name (`NAME_SLOT`), which is replaced by the name with which the component is parameterised in the composition program. Second, the composition interfaces should hide details of the component language (UML use case in Figure 2.4) such that the language can be exchanged, which allows us to reuse the extracted composition program to compose different types of models. This is illustrated in Figure 2.5, where the UML use case fragments have been exchanged for Java fragments. While the composition performed by the composition engine is different (instead of merging UML Actors and UseCases, Java Methods and Statements are merged), the composition interfaces remain the same. The observation above lead us to the following requirements for CB-MDSD:

- **Requirement 1 (Arbitrary Component Languages)** We need to treat arbitrary models as components and, therefore, should support arbitrary DSMLs as component languages. Since the structure of models defined in different DSMLs is different, it needs to be configurable how the internals of a model are mapped to its composition interface. This configurability ensures that arbitrary DSMLs can be treated as component languages.
- **Requirement 2 (Arbitrary Composition Languages)** It should be possible to use different DSMLs as composition languages. That is, we require a method to extract composition programs from models defined in arbitrary DSMLs.
- **Requirement 3 (Abstract Composition Interfaces)** It must be possible to define abstract composition interfaces for models. That is, interfaces must be independent of the component language such that components with similar interfaces can be modelled in different DSMLs. Wrt. Requirement 1, this means that similar interfaces must be mapped to model internals differently for different DSMLs. This ensures that the composition language and composition engine work language-independent (i.e., universal). If this requirement is met, information in different formats can be composed with a single composition program.
- **Requirement 4 (Universal Composition Language)** To meet Requirement 2, a universal composition language is needed, which is independent of the DSMLs in which composition information is defined. This language should be both expressive enough to express all composition information extracted from different types of models and minimal (i.e., without redundancy) to be simple to understand and process. This ensures that information defined in different composition languages can be unified in one format for processing by a universal composition engine.

2.1.2 Combining Multiple Composition Languages

Above, we discussed the extraction of information from one source model by regarding the DSML of that model as composition language (OpenDocument in the example above). However, as soon as information is defined in different types of models, multiple DSMLs need to be treated as composition languages. An example of this can be seen in Figure 2.3, where additional information about the *BookTicket* use case is defined in UML use case diagrams. Thus, to extract this information also UML use case diagrams have to be treated as composition language (Figure 2.6).

Let us first consider the case of adding the actor Hall, which is not mentioned in the textual use case description but added through a use case diagram (Figure 2.3 right). As illustrated in

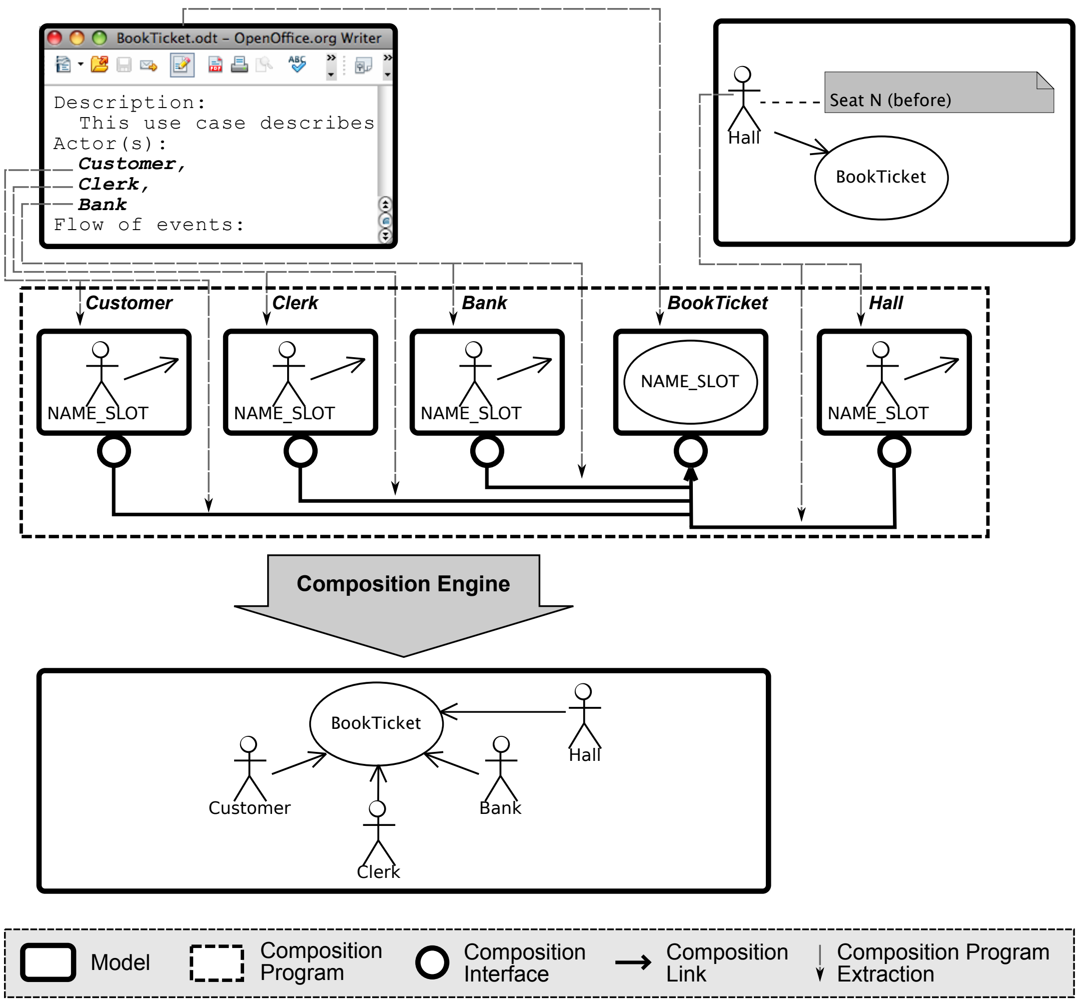


Figure 2.6: UML use cases as additional composition language.

Figure 2.6, this is similar to the way the other actors were added in the composition program. However, the source from which this composition information is extracted is now a use case diagram. The joint composition program can now be used to compose an integrated use case diagram with the new Actor (Figure 2.6) but also to compose the corresponding Java code if one exchanges the use case diagram components with Java components (as was done in Figure 2.5). To allow for this, it must be possible to aggregate composition information from multiple models in one composition program of uniform format. Therefore, we formulate the following requirement:

- **Requirement 5 (Aggregatable Composition Programs)** It must be supported to aggregate composition information from multiple models in one composition program. This allows for the gathering of composition information from arbitrary models and therewith the usage of arbitrary DSMLs as composition languages in combination. Thus, this requirement extends the features of a universal composition language (Requirement

4) to support aggregation of multiple composition programs into one. If this requirement is met, all composition information for one system developed with CB-MDSD can be collected in one composition program which, at any point during development, represents the complete architecture of the modelled system.

2.1.3 Combining Multiple Modularity Concepts

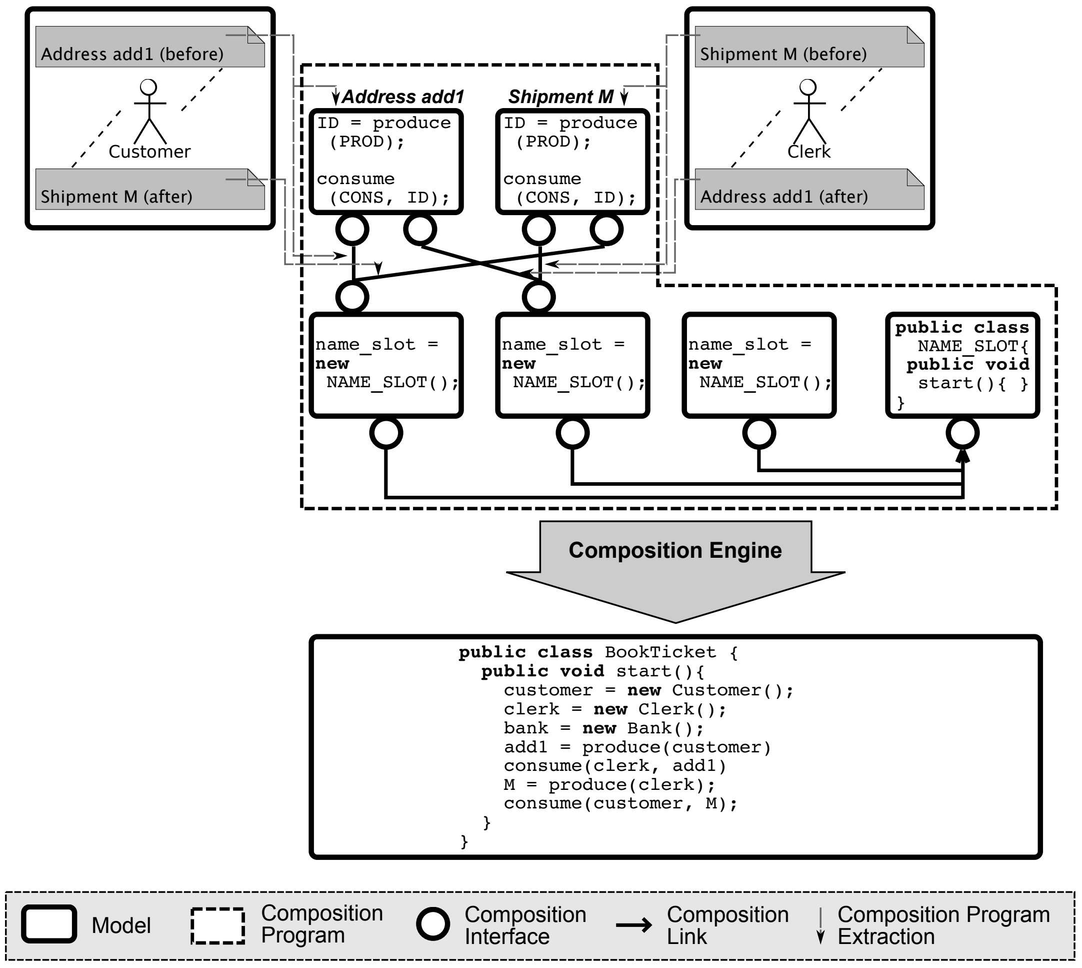
Although we have used two different composition languages in Figure 2.6 (OpenDocument and UML use cases) we composed the same kind of information (actors and use cases). Concretely, this means that the composition engine composed either UML Actors and UseCases (in case of UML as component language; Figure 2.4) or Java Methods and Statements (in case of Java as target component language; Figure 2.6). If we take a language-independent view on this, we performed an "actor/use case" composition; independent of the fact that actors are at one time represented by UML Actors and at other time by Java Statements. Formulating such *modularity concepts* becomes necessary, if multiple of them are used in parallel.

Let us again consider the refinement of use cases with additional UML use case diagrams (Figure 2.3). Here we find another modularity concept which is "actor/value" composition. As Figure 2.7 illustrates, the combination of actors with value added invariant annotations can again be extracted to further enrich the unified composition program. Only this time, the modularity concept of "actor/value" composition is used. In the concrete case of Java as component language, this modularity concept is realised by a specific composition between Java Statements that represent actors and Java Statements that represent the exchange of business values between actors (Figure 2.7). Since different modularity concepts may be needed, we require means to aggregate them:

- **Requirement 6 (Aggregatable Composition Interfaces)** In CB-MDSD, different modularity concepts are used in parallel and in combination with multiple DSMLs. In Requirement 3 we formulated the need for component language-independent composition interfaces. Here we extend this, by requiring these interfaces to be aggregatable. This way, the interfaces required by multiple modularity concepts can be aggregated to a joint interface for one fragment.

2.2 CB-MDSD with a Universal Composition Language

Once we defined a universal composition language (Requirement CL2) we can also consider using this language directly for CB-MDSD. Then, instead of extracting composition information from different models, the universal composition language is used by developers directly to create composition programs. This can be a suitable solution in situations where development is centred around a single DSML and transformation of information into different formats is not an issue. Still, in such cases, component support is a concern sooner or later to allow for modularisation and reuse. To add such component support, an architect can consider to integrate the universal composition language directly, which spares the effort of defining additional DSMLs or DSML extensions for the purpose of modularity. Instead, he or she can reuse the existing universal composition language and its tooling out-of-the-box.



```
ID = produce
  (PROD);

consume
  (CONS, ID);
```

```
name_slot =
new
 NAME_SLOT();
```

```
public class
  NAME_SLOT{
 public void
  start(){ }
}
```

```
public class BookTicket {
  public void start(){
    customer = new Customer();
    clerk = new Clerk();
    bank = new Bank();
    add1 = produce(customer)
    consume(clerk, add1)
    M = produce(clerk);
    consume(customer, M);
  }
}
```

Figure 2.7: “actor/value” composition as additional modularity concept.

As illustration, consider Figure 2.8 in which UML class diagrams are composed using a new aspect-like modularity concept. Here, classes are extended with new attributes, associations, and methods by defining a composition program in a universal composition language. In the example, a file system is modelled consisting of a `FileSystem`, an `FSFolder`, and an `FSFile` class. This model is extended with observer/subject functionality, which is defined in a separate model by the `Observer` and `Subject` classes. The composition engine merges properties of the `Observer` and `Subject` classes into the other classes. Instead of extracting a composition program, it is defined directly in the universal composition language. The composition program is indicated by the boxes in the figure and the arrows between them. (In a real composition language, one would probably not see the models inside the boxes, but only the boxes, representing model components, and the circles, representing the composition interface.)

Using the universal language directly is possible in this case, because the composition interfaces are simple enough such that the developer can understand the complete composition

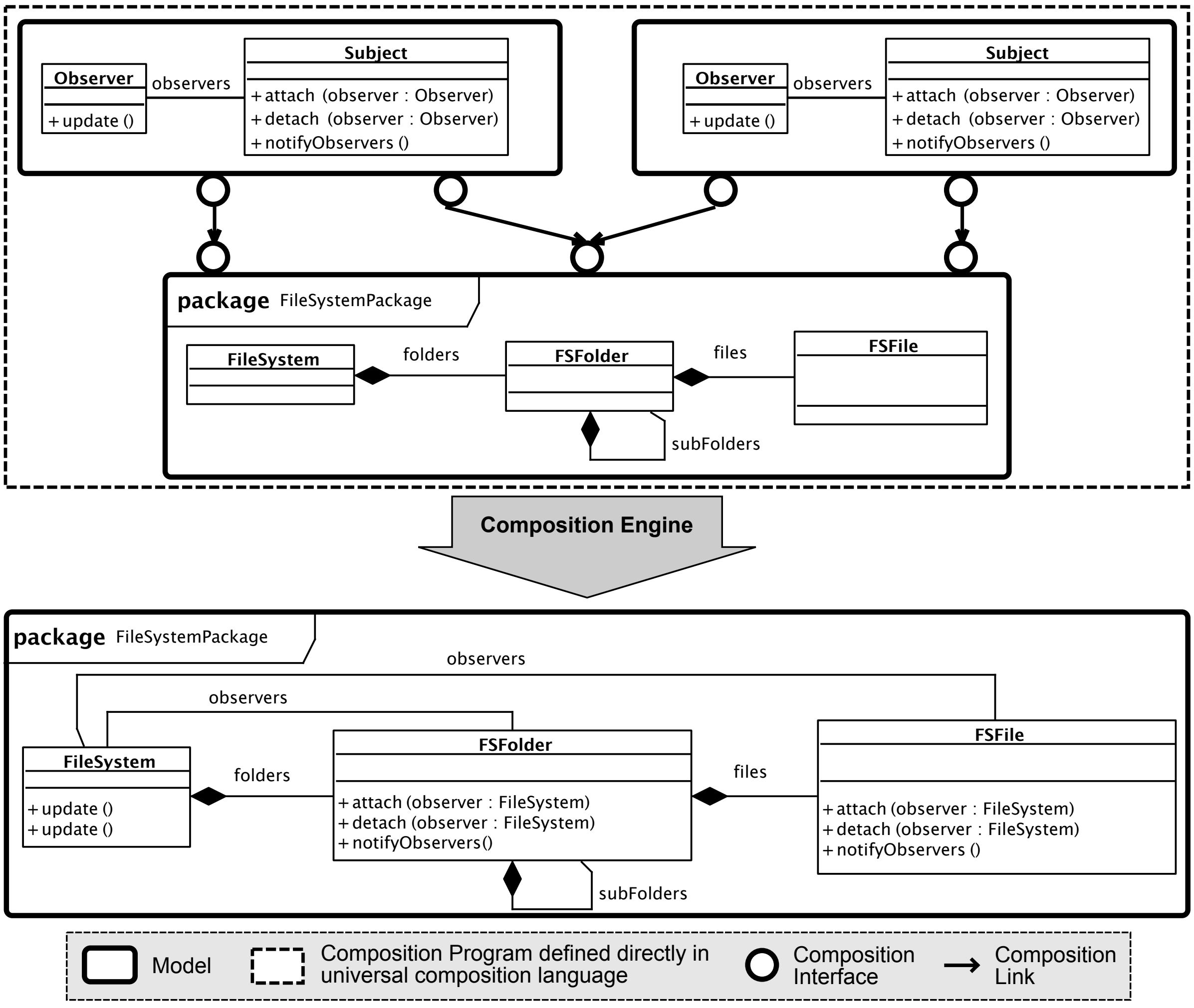

Observer
+update ()
observers
Subject
+attach (observer : Observer)
+detach (observer : Observer)
+notifyObservers ()
package FileSystemPackage
FileSystem
folders
FSFolder
files
FSFile
subFolders
Composition Engine
+attach (observer : FileSystem)
+detach (observer : FileSystem)
+notifyObservers()
Model
Composition Program defined directly in universal composition language
Composition Interface
Composition Link

Figure 2.8: Universal composition language used directly to compose UML class diagrams.

program and the impact of changes to it. In contrast, a composition program that represents the complete architecture of a system defined by multiple DSMLs and covers several modularity concepts at once is difficult to understand for direct manipulation (e.g., Figure 2.7 already starts to get complex although it only represents a small part of the complete system architecture). The component language (UML class diagrams in Figure 2.8) can still be exchanged for another language (e.g., Java or another object-oriented language) and the composition program can be reused.

We argue that in cases like this, where the composition programs are of reasonable size and the composition interfaces of components are not overloaded, it should be considered to use the universal composition language directly. However, it is a trade-off that depends on the concrete model-driven architecture to be realised. We discuss this further in Chapter 9.

In this chapter, we showed examples for our idea of CB-MDSD and derived requirements for our composition technique from them. The remainder of Part I introduces this composition

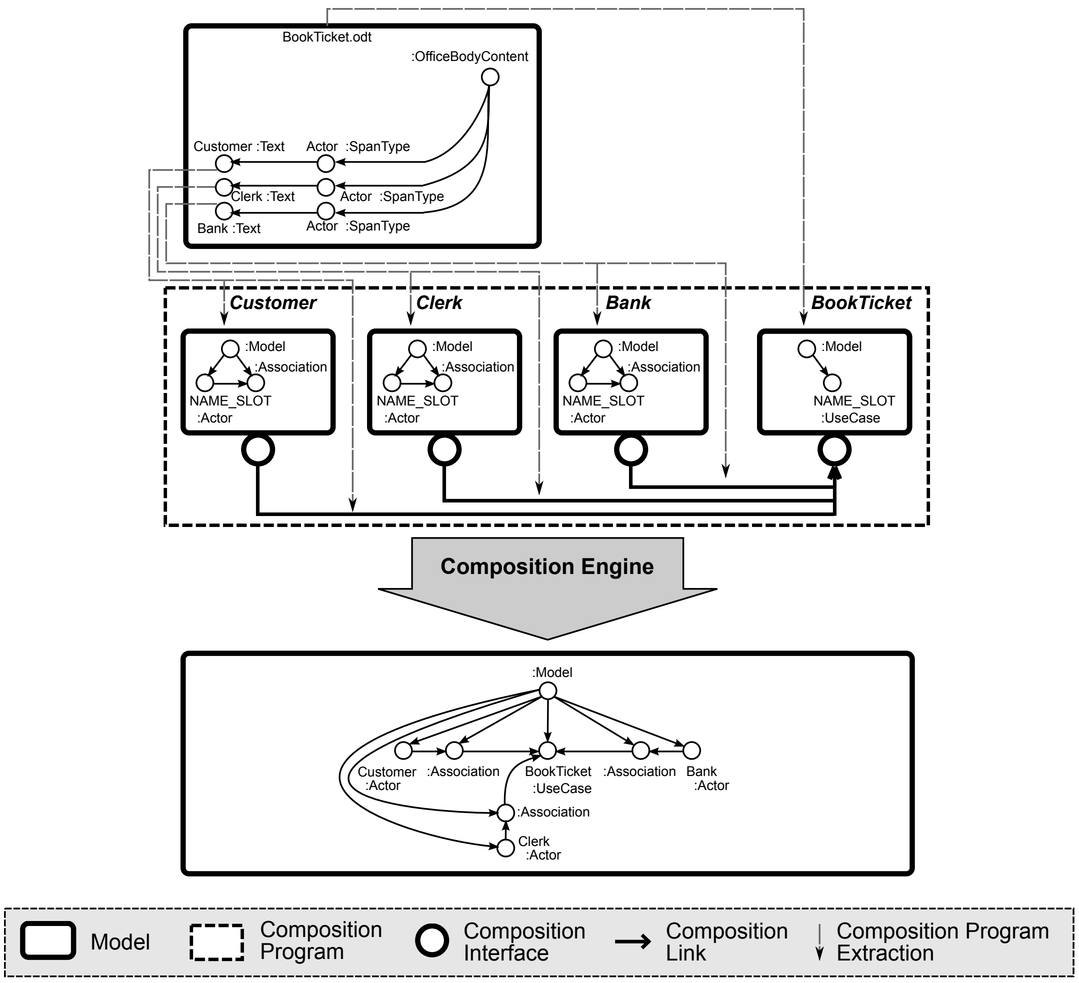


Figure 2.9: Figure 2.4 with all models represented as graphs.

technique—called U-ISC/Graph—and shows how the requirements are met by it. For this, we will continue to use the examples of this chapter for illustration. To give a hint of the direction we are going, consider Figure 2.9. It is similar to Figure 2.4 only that the models are not shown in their concrete (diagrammatic) syntax but as typed and directed graphs. This is how all models are represented "under the hood" if they are defined in a DSML that was specified with EMOF. Our composition technique works on these graph representations of models and is therefore an extension of U-ISC/Tree, which is a composition technique that works on trees. U-ISC/Tree is introduced in the next chapter as a foundation for our work.

3 Invasive Software Composition Systems

This chapter explains how CB-MDSD is regarded as a composition system. It introduces Invasive Software Composition (ISC) and Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree) as foundations for such composition systems.

To allow a better understanding of the relationships between models and their composition in CB-MDSD, we introduce the term *composition system.* According to [Aßm03], a composition system is a triple of *component model*, *composition language*, and *composition technique* (cf. Figure 3.1). First, a component model provides *component languages* for the definition of components (cf. Chapter 2 for examples of component languages in CB-MDSD). Second, a composition language is used to define *composition programs* which in turn define a system by composition of components (cf. Chapter 2 for examples of composition programs in CB-MDSD). Third, a composition technique interprets the composition programs to compose a system from components. An important role in a composition system play *composition interfaces*, through which different parts of a composition system interact. They are the interfaces that are used to connect components in composition programs and to access components for composition by the composition engine (cf. Chapter 2 for examples of composition interfaces in CB-MDSD). As Figure 3.1 indicates, the three parts of a composition system overlap. It depends on the concrete composition system how strong these overlaps are.

We call a composition system with unspecified parts that have yet to be defined a *generic composition system.* A generic composition systems may hence be used as a common basis for a set of composition systems.

A model-driven architecture can be realised as a composition system with multiple component and composition languages. This is indicated by the examples in Section 2.1, which suggests the usage of multiple component and composition languages in combination. Since we aim at a generic solution for CB-MDSD, we define a generic composition system in the forthcoming Chapters 4–6. The foundations for that are presented in this chapter.

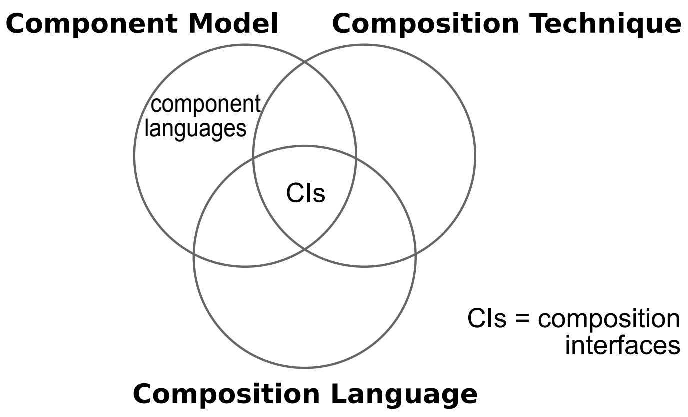


Figure 3.1: Composition systems consist of a *composition technique*, a *component model*, and a *composition language* which interact.

Generic composition systems that can be configured to work with different languages are provided by Invasive Software Composition (ISC) [Aßm03] and Universal Invasive Software Composition (U-ISC) [Hen09]. However, ISC and U-ISC are not concerned with MDSD and, therefore, we first need to extend them to provide all the features we require for our generic CB-MDSD composition system. To understand what can be reused and where extension is required, we discuss ISC in Section 3.1 and U-ISC in Section 3.2.

An overview of the generic composition system that we develop in this work is given in Figure 3.2. The figure shows the eight main features of the generic composition system that we develop in Chapter 4 (Feature 1), Chapter 5 (Features 2–4), and Chapter 6 (Features 5–8).

The features that directly concern the component model and the composition languages (Features 1, 4, 6, and 8) are derived from the requirements formulated in Chapter 2. Feature 1 expresses the need for abstract composition interfaces (Requirements 3 and 6) for which we introduce a concept called *fragment collaborations*. Feature 4 requests exchangeable component languages (Requirement 1), while Features 6 and 8 are both concerned with composition languages, demanding a universal composition language (Requirements 4 and 5) and exchangeable composition languages (Requirement 2) respectively. The remaining features (Features 2, 3, 5, and 7) are all related to the composition technique which will be based on the composition technique of ISC. Their need will therefore be clarified in the remainder of this chapter.

Once the generic composition system is available, it can be configured for a concrete model-driven architecture as illustrated in Figure 3.3. For generic composition systems, we distinguish *composition system developers* and *composition system users*. Developers fill the generic parts of a generic composition system and user utilise the such obtained complete system either to define components or composition programs. In our case, the process developer, who sets up a model-driven development process by selecting and combining DSMLs, acts as composition system developer by plugging the DSMLs as component and/or composition languages into the generic composition system. The such defined composition languages are then used by architects for modelling-in-the-large and the component languages are used by developers for modelling-in-the-small. Architects and developers are thus both composition system users.

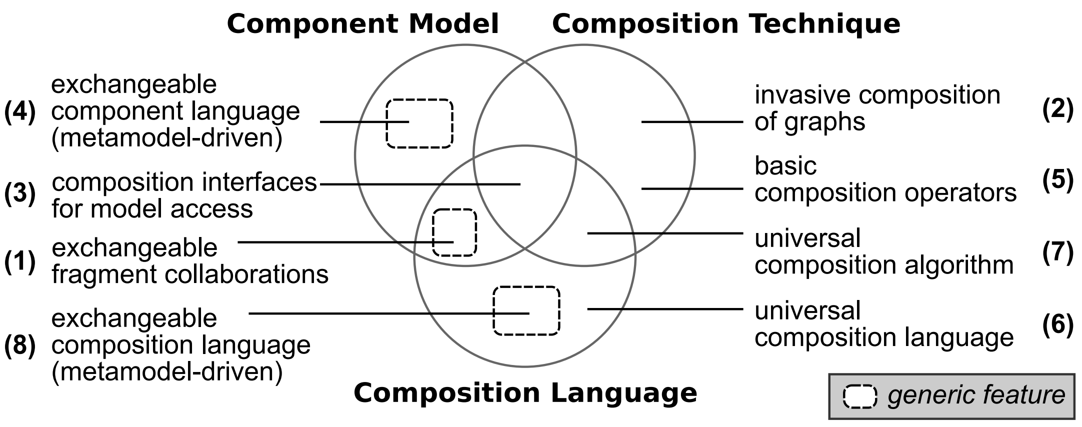

Component Model
Composition Technique
(4) exchangeable component language (metamodel-driven)
(3) composition interfaces for model access
(1) exchangeable fragment collaborations
(8) exchangeable composition language (metamodel-driven)
invasive composition of graphs (2)
basic composition operators (5)
universal composition algorithm (7)
universal composition language (6)
Composition Language
generic feature

Figure 3.2: Features of a generic composition system for CB-MDSD.

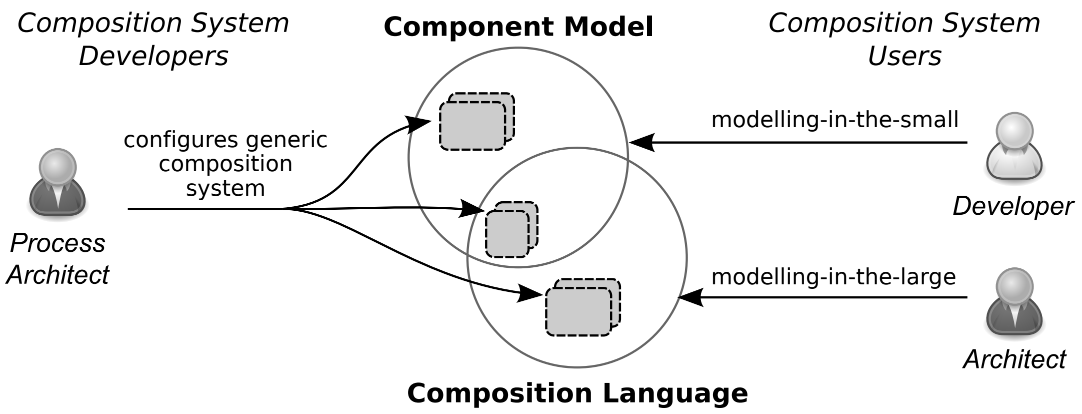


Figure 3.3: Users of a composition system for MDSD.

3.1 Invasive Software Composition (ISC) and the Compost System

The foundation for ISC systems is provided in [Aßm03]. ISC is a program transformation approach that composes partial programs—*fragment components* (or *fragments* in short)—to complete programs by merging them. Arbitrary languages may be chosen as component language to define fragments (in [Aßm03] only Java is used for demonstration purposes). A fragment component (called fragment box in [Aßm03]) has a *composition interface* that consists of *hooks*. A hook identifies a point for variation or extension in a fragment component. A hook can be bound to another fragment component, which means that a fragment component is inserted at the position of the hook during composition. [Aßm03] also introduced the notion of *nested hooks* which combine several hooks into one to form richer composition interfaces. Hooks are either *declared* or *implicit*. Declared hooks are manually defined by fragment developers, implicit hooks always exist for certain types of fragments.

The composition itself is executed by *composition operators* (called composers in [Aßm03]). [Aßm03] identified two basic operators (*bind* and *extend*) which can serve as a base for *complex*

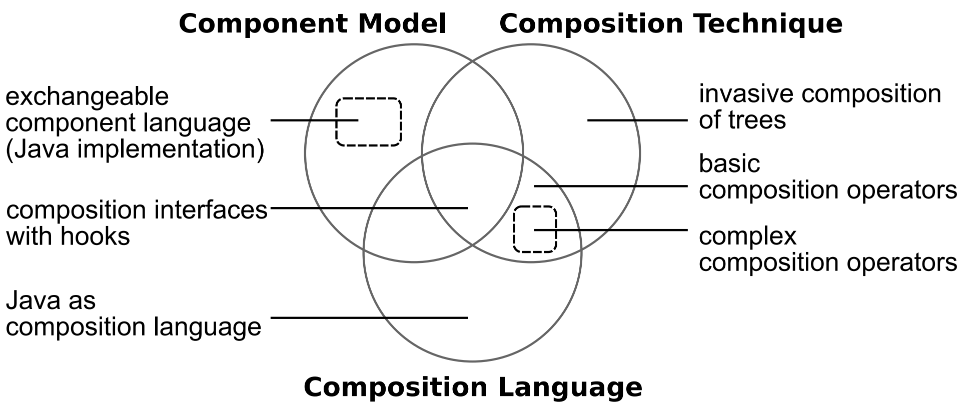

Component Model
Composition Technique
exchangeable component language (Java implementation)
composition interfaces with hooks
Java as composition language
invasive composition of trees
basic composition operators
complex composition operators
Composition Language

Figure 3.4: Compost as generic composition system.

operators (called compound operators in [Aßm03]). A composition program in ISC consist of calls to basic or complex composition operators.

While ISC is as a methodology can be implemented for different component or composition languages, [Aßm03] only provides a generic composition system with Java as composition language called Compost system[1]. It implements the ISC foundation in a Java framework and provides an instantiation of the framework with Java as component language. In these composition systems, Java is a generic composition language which is extended for concrete composition systems by implementing complex composition operators as Java methods. Thus, all Compost-based composition systems use Java, together with a library of composition operators, as composition language. To support new component languages for fragment definition, the Compost system needs to be extended manually for each component language. It was shown that this is possible for XML as component language, but not without considerable effort [Sav03]. Compost as generic composition system is illustrated in Figure 3.4.

Example 3.1. To illustrate the original ISC as it is realised in Compost, we show an example in the following. It is based on the example from Section 2.2, but uses Java instead of UML class diagrams as component language, since Java is supported by Compost while UML is not. The example follows the idea from [Aßm03, Chapter 10] to use ISC for Aspect-Oriented Programming (AOP) [KLM+97]. The idea of AOP is to introduce a new modularity concept into an object-oriented language such as Java that enables us to extend existing classes with additional members. This can be realised through an ISC composition system with a composition operator that extends the list of members of a class invasively.

In our AOP composition systems we have two different kinds of fragment components—*core* and *advice* fragments. Core fragments implement a system core that is to be extended by aspects. An aspect consists of two things: advice code and weaving instructions. In ISC terms, the advice code is realised as fragments, while the weaving instructions are defined in composition programs.

In the following, we introduce an example system consisting of fragments defined in Java as component languages (Listings 3.1 and 3.2) and a composition program defined in Java as

[1]`http://www.the-compost-system.org/`

```java
public class FileSystem {
    protected FSFolder[] folders;
}

public class FSFolder {
    protected String name;
    protected FSFolder[] subFolders;
    protected FSFile[] files;
}

public class FSFile {
    protected String name;
    protected Byte[] content;
}
```

Listing 3.1: Core fragments implementing a file system in Java.

composition language (Listing 3.4). The core is defined in Listing 3.1 with `FileSystem` as the central class that manages the application. It contains a list of `FSFolders`. A `FSFolder` may contain further `FSFolders` and `FSFiles`. In Listing 3.2, we see two advice fragments that implement an observer aspect. This is in line with the observer design pattern [GHJV94] with the exception that we implement the observer relationship in separate classes and not in the same class hierarchy as the core system, since we intentionally do not use inheritance to present AOP as an alternative implementation method. The same example was modelled with UML as component language and an imaginary universal composition language in Figure 2.8.

The advice classes contain declared hooks. The `Observer`, for example, can hold an array of collaborators of yet unspecified type for which it declares the hooks `genericCollaboratorType` and `genericCollaboratorNameIdentifier` (Line 2). These parts of the code are recognised as hooks, because `generic<NAME>Type` and `generic<NAME>Identifier` are naming conventions introduced by Compost to allow hook declaration without extending the Java syntax. Similar hooks are defined in the `Subject` class. Hooks with the same name are grouped into one nested hook.

The desired aspect composition system can be built in Compost since it supports Java as component language. As mentioned, Compost uses Java as generic composition language that can be extended with new composition operators. For our example system, we introduce the operator *weave* that can compose core and advice fragments written in Java according to the composition semantics of our AOP system described above. The *weave* operator, implemented as a Java method, is shown in Listing 3.3. It extends the `member` hook of the core fragment, which is an implicit hook that all fragments containing a Java class (`ClassBox`) have, with all members that are contained in the advice fragment (Line 2). Furthermore, the weave operator takes another fragment component, the `ClassBox collaborator`, as argument. This argument can be used to pass another core class to which the collaborator hooks of the advice code should be bound. For compost the type (hook `genericCollaboratorType`) and the collaborator array name (hook `genericCollaboratorNameIdentifier`) are only strings. Consequently, the weave operator constructs the corresponding strings based on the collaborator class name and binds them to the hooks (Lines 3–5).

```java
public class Observer {
    protected genericCollaboratorType[] genericCollaboratorNameIdentifier =
        new genericCollaboratorType[0];

    public void update(genericCollaboratorType subject) {
        //...
    }
}

public class Subject {
    protected genericCollaboratorType[] genericCollaboratorNameIdentifier =
        new genericCollaboratorType[0];

    public void attach(genericCollaboratorType observer) {
        genericCollaboratorType[] newObservers =
            new genericCollaboratorType[genericCollaboratorNameIdentifier.length + 1];
        for (int i = 0; i < genericCollaboratorNameIdentifier.length; i++) {
            newObservers[i] = genericCollaboratorNameIdentifier[i];
        }
        newObservers[genericCollaboratorNameIdentifier.length] = observer;
        genericCollaboratorNameIdentifier = newObservers;
    }

    public void detach(genericCollaboratorType observer) {
        genericCollaboratorType[] newObservers =
            new genericCollaboratorType[genericCollaboratorNameIdentifier.length - 1];
        int skip = 0;
        for (int i = 0; i < genericCollaboratorNameIdentifier.length; i++) {
            if (genericCollaboratorNameIdentifier[i].equals(observer)) {
                skip = 1;
            }
            else {
                newObservers[i] = genericCollaboratorNameIdentifier[i - skip];
            }
        }
        genericCollaboratorNameIdentifier = newObservers;
    }

    public void notifyObservers() {
        for (int i = 0; i < genericCollaboratorNameIdentifier.length; i++) {
            genericCollaboratorType observer = genericCollaboratorNameIdentifier[i];
            observer.update(this);
        }
    }
}
```

Listing 3.2: Advice fragments implementing observer functionality in Java.

```java
public static void weave(ClassBox coreFragment, ClassBox adviceFragment, ClassBox collaborator) {
    coreFragment.findHook("members").extend(adviceFragment);
    adviceFragment.findGenericType("Collaborator").bind(collaborator.getName());
    String collaboratorName = "_" + collaborator.getName().toLowerCase();
    adviceFragment.findGenericIdentifier("CollaboratorName").bind(collaboratorName);
}
```

Listing 3.3: The *weave* composition operator defined as Java method in Compost.

```
 1 public static void main(String[] args) {
 2     JavaCompositionSystem javaCompositionSystem =
 3         new FragmentCompositionSystem(".").getJavaCompositionSystem();
 4
 5     ClassBox fileSystem = javaCompositionSystem.createClassBox("core/FileSystem");
 6     ClassBox folder     = javaCompositionSystem.createClassBox("core/FSFolder");
 7     ClassBox file       = javaCompositionSystem.createClassBox("core/FSFile");
 8
 9     ClassBox observerCopy1 = javaCompositionSystem.createClassBox("advice/Observer");
10     ClassBox subjectCopy1  = javaCompositionSystem.createClassBox("advice/Subject");
11
12     ClassBox observerCopy2 = (ClassBox) observerCopy1.copy();
13     ClassBox subjectCopy2  = (ClassBox) subjectCopy1.copy();
14
15     weave(fileSystem, observerCopy1, folder);
16     weave(file, subjectCopy1, fileSystem);
17
18     weave(fileSystem, observerCopy2, file);
19     weave(folder, subjectCopy2, fileSystem);
20
21     javaCompositionSystem.print(fileSystem);
22     javaCompositionSystem.print(folder);
23     javaCompositionSystem.print(file);
24 }
```

Listing 3.4: The composition program to compose core and advice fragments in Compost.

Since Java is also the composition language of all Compost-based composition systems, a composition program using the above defined composer is also written in Java as shown in Listing 3.4. After initialising the Compost tooling in Line 2, the core fragments (`FileSystem`, `FSFolder`, and `FSFile`; cf. Listing 3.1) and advice fragment (`Observer` and `Subject`; cf. Listing 3.2) are loaded in Lines 5–10. Copies of the observer and subject fragments are created in Lines 12 and 13 because we want to weave the aspect two times (between `FileSystem` and `FSFolder` as well as between `FileSystem` and `FSFile`). The *weave* composer is then called four times: in Line 15 to weave the observer advice code into the file system, in Line 16 to weave the subject advice code into the folder, in Line 18 to weave the observer advice code a second time into the file system and in Line 19 to weave the subject advice code into the file. Finally, in Lines 21–23, the extended core fragments are printed. The result of the invasive composition can be observed in Listing 3.5. Note that the observer code was woven into `FileSystem` twice with different types because of the different bindings of the collaborator hooks.

ISC defines many concept useful for our generic CB-MDSD composition system based on which we defined Features 2, 3, 5, and 7 in Figure 3.2. ISC is generic wrt. the component language and defines the hook concept as a basic composition interface concept to access fragments that can be integrated into arbitrary component language (Feature 3). Composition by fragment merging which is performed by composition operators (Features 5 and 7), as illustrated in the example, is suitable for model composition, since it is a static composition technique (Feature 2). Also these concepts fit in general, there are certain specifics we need to consider in order to realise the remaining features that were derived from our requirements. We discuss this at the end of this chapter. Before, we investigate Universal and Embedded ISC which define useful extensions of ISC concepts.

```
public class FileSystem {
    protected String name;
    protected FSFolder[] folders;

    protected FSFile[] _fsfile = new FSFile[0];

    public void update(FSFile _fsfile) {
        // ...
    }

    protected FSFolder[] _fsfolder = new FSFolder[0];

    public void update(FSFolder _fsfolder) {
        // ...
    }
}

public class FSFolder {
    protected String name;
    protected FSFolder[] subFolders;
    protected FSFile[] files;

    protected FileSystem[] _filesystem = new FileSystem[0];

    public void attach(FileSystem observer) {
        // ...
    }

    public void detach(FileSystem observer) {
        // ...
    }

    public void notifyObservers() {
        // ...
    }
}

public class FSFile {
    protected String name;
    protected Byte[] content;

    protected FileSystem[] _filesystem = new FileSystem[0];

    public void attach(FileSystem observer) {
        // ...
    }

    public void detach(FileSystem observer) {
        // ...
    }

    public void notifyObservers() {
        // ...
    }
}
```

Listing 3.5: The result of executing the COMPOST composition program of Listing 3.4.

3.2 Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree)

[Hen09] introduces *Universal* Invasive Software Composition. It explores how the development of ISC systems can be eased by generating component models based on a context-free grammar of a language. For this, it builds on concepts originating from *Grammar-Based Modularisation* as found in the Beta programming language [MMPN93]. To better distinguish the approach of [Hen09] from ours, we refer to it as Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree).

In detail, [Hen09] introduces the following approaches that build upon each other.

3.2.1 Universal Grammar-Based Modularisation [Hen09, Chapter 2]

In the object-oriented programming language Beta, modularisation is supported by a *fragment system*. The fragment system is not part of the language itself. Thus, modularisation concerns did not influence the design of the language syntax or semantics. Rather, the fragment system is defined on top of the Beta language grammar. It supports that each language construct defined by the Beta grammar can be instantiated and compiled on its own to a fragment. For instance, one may instate single statements and not only complete classes. Furthermore, Beta's fragment system allows the developer to specify so-called *slots* instead of concrete elements. Slots thus mark places in code where concrete elements are missing. Slots can then be bound to fragments by the fragment system to eventually form a program without slots that can be executed. The fragment system hereby ensures that slots are only bound to fragments that fit the slot following the typing rules provided by the language grammar.

Although Beta's fragment system was only used in Beta, its developers recognised that it can in principle be ported to any language since it is based on concepts of grammars only and not on Beta specifics [MMPN93]. Henriksson takes up this idea for *Universal Grammar-Based Modularisation*, showing that the ideas can be ported to arbitrary languages that are defined by context-free grammars. He argues, that independent compilation of fragments cannot be achieved, because compilation is language-dependent. Enabling the independent instantiation of each language construct of a grammar is, however, possible for arbitrary languages. To enable the specification of slots with proper tool support, Henriksson introduces a formalism to automatically extend grammars in such a way that they would allow slot specification (with a specific syntax for slots) instead of concrete elements on all possible places in a program. The such obtained grammar—called *reuse grammar*—still covers the original language but allows slot specifications in addition. Based on that grammar, tool support (e.g., parsers) can be generated. Furthermore, a language-independent composition engine can be defined that binds slots to other fragments by rewriting abstract syntax trees. This composition engine can utilise the typing given by a grammar to ensure syntactically correct compositions.

3.2.2 Universal Invasive Software Composition [Hen09, Chapter 3]

Henriksson shows that there are many similarities between Beta's fragment system and ISC. He observes that the slot concept (of Beta) and the declared hook concepts (of ISC) are similar. The missing part in Beta is the concept of implicit hooks and the possibility of extension. While slots can only be bound (i.e., replaced), implicit hooks cannot only be bound but also

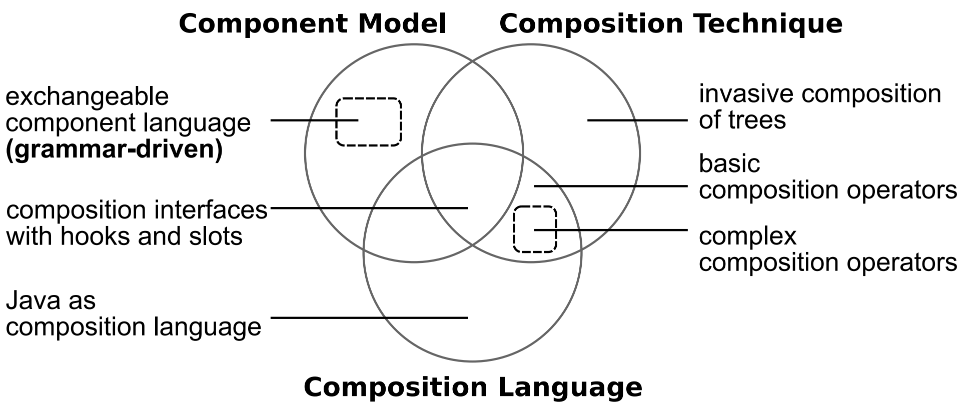

Component Model
Composition Technique
exchangeable component language (grammar-driven)
composition interfaces with hooks and slots
Java as composition language
invasive composition of trees
basic composition operators
complex composition operators
Composition Language

Figure 3.5: U-ISC/Tree as generic composition system.

extended. Furthermore, ISC offers the complex composition operator concept to encapsulate a set of basic bind or extend operations in one operator. Such a concept is not available in Grammar-Based Modularisation in Beta.

Henriksson defines U-ISC/Tree as a generalisation of Universal Grammar-Based Modularisation. He continues to use the grammar extension formalism of Universal Grammar-Based Modularisation, which effectively generates an invasive component model. On the one hand, he allows for tailoring. That is, instead of introducing slots for every language construct, one can decide for which constructs slots should be allowed. On the other hand, special grammar annotations to specify implicit hooks are introduced.

In addition to the grammar extension, Henriksson's approach allows for the specification of complex composition operators (using, as Compost did, Java as specification language for them). Composition programs can then be written in a similar fashion as it was done in Compost (cf. Listing 3.4). However, the component model API (which had to be implemented manually in Compost) is now generated based on the extended language grammar. This allows for Compost-like invasive compositions for arbitrary languages defined by a context-free grammar without implementation effort. Although, it should be noted that the manually implemented API in Compost for Java as component language is richer than the one that U-ISC/Tree would generate from a Java grammar. Generating a component model API for Java or a similar large language was also not performed and not evaluated in [Hen09] and thus a detailed comparison between generated and manually implemented API was not conducted.

To summarise, U-ISC/Tree employs an extension of the concepts of Grammar-Based Modularisation to generate a component model for an arbitrary context-free language and a Java API for that component model. This enables developers to write fragment components in their language of choice and to write complex composition operators as well as composition programs in a Compost-like fashion in Java. Thus, it still uses one generic imperative composition language—Java—in all composition systems.

The advance of U-ISC/Tree over ISC and Compost is that it offers a grammar-driven mechanism to integrate new component languages into a composition system. This is illustrated in Figure 3.5.

```
public class FileSystem {
    weave "advice/Observer" with collaborator "core/FSFile";
    weave "advice/Observer" with collaborator "core/FSFolder";

    protected FSFolder[] folders;
}

public class FSFolder {
    weave "advice/Subject" with collaborator "core/FileSystem";

    protected String name;
    protected FSFolder[] subFolders;
    protected FSFile[] files;
}

public class FSFile {
    weave "advice/Subject" with collaborator "core/FileSystem";

    protected String name;
    protected Byte[] content;
}
```

Listing 3.6: Composition language concepts embedded in Java.

3.2.3 Embedded Invasive Software Composition [Hen09, Chapter 4]

Henriksson emphasises that an important success factor for ISC is the ability to abstract from low level composition operations by defining complex composition operators. He argues however that for a system developer, who would use ISC to compose fragments written in his or her own (domain-specific) language, this abstraction is still not appropriate. The developer should be able to express composition programs in terms of his domain rather than coping with technical ISC issues like loading fragments in Java-based composition programs.

To overcome this issue, Henriksson introduces *Embedded ISC* as another layer on top of U-ISC/Tree. Here he further extends languages, which have already been extended for U-ISC/Tree, with additional constructs to call composition operators. An example of such a construct could be an `import` statement as known, for example, from Java. Such a statement can be connected to a composition operator that performs the import in terms of extend composition operator calls. Furthermore, a generic composition language, called FL^{ABS}, is provided, which offers convenient constructs such as fragment loading. Constructs from the generic language can be integrated into other languages along with the constructs to call composition operators. Thus, Embedded ISC allows the specification of a domain-specific composition language that is directly embedded into the component language.

As an example, consider the extension of Java with a *weave* construct that calls the *weave* composition operator (cf. Listing 3.3) directly from inside a Java program as shown in Listing 3.6, where the weaving is directly specified in the file system classes. With such an integration of component and composition language, there is no need to define an external composition program.

In contrast to COMPOST or U-ISC/Tree, Embedded ISC allows the usage of other composition languages than Java if component and composition language are the same. To support this, it provides a generic composition language that can be integrated into other languages.

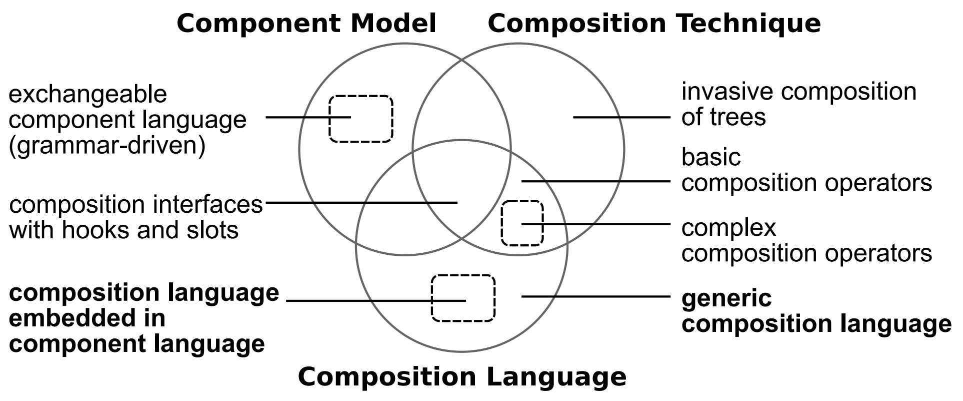

Component Model
Composition Technique
exchangeable component language (grammar-driven)
composition interfaces with hooks and slots
composition language embedded in component language
invasive composition of trees
basic composition operators
complex composition operators
generic composition language
Composition Language

Figure 3.6: Embedded ISC as generic composition system.

3.3 Conclusion

By comparing Figures 3.4, 3.5, and 3.6 with Figure 3.2, we can analyse, where the existing ISC approaches need to be extended to provide all features for the generic CB-MDSD composition system. Concerning the composition technique, we need to extend the tree composition approach to a graph composition approach, since models are represented as graphs (Feature 2). This has impact on the basic composition operators, which have so far been defined to work on trees only (Feature 5).

Concerning the component model, U-ISC supports a grammar-driven method to integrate component languages. For MDSD, where DSMLs are defined with EMOF, we require a metamodel-driven[2] method (Feature 4).

ISC does so far not include a concept to explicitly define modularity concepts to couple component model and composition language independent of the composition technique. Thus, this is a new concept we will introduce (we call the concept *fragment collaborations*; Feature 1).

We also require a universal composition language (Feature 6) as common base to integrate arbitrary composition languages (Feature 8). ISC so far did not investigate in providing such a language or supporting arbitrary combinations of component or composition languages. The key for providing such a universal composition language is the decoupling of composition language and technique. All ISC realisations so far were tightly bound to Java to define complex composition operators. The operators—including the basic ones—were directly called in composition programs. To achieve this decoupling, we will define a universal composition algorithm (Feature 7) that works for fragments defined in arbitrary component languages. The algorithm replaces the complex composition operator concept and therefore the need for operator definition. Furthermore, it hides the basic operators in the composition technique (Feature 5) and makes composition programs independent of them. This will be possible by moving component and composition language-specifics from complex composition operators to the integration of component and composition language (Features 4 and 8) which will be integrated via fragment collaborations (Feature 1).

[2]We refer to language specifications defined in EMOF [OMG06a] as metamodels.

Hence, fragment collaborations (Feature 1) play a central role in our generic CB-MDSD composition system. They will therefore be defined next in Chapter 4. Afterwards we deal with the basics for a composition technique for models (Feature 2) and component model related features (Features 3 and 4) in Chapter 5. The composition operators and composition languages (Features 5–8) are treated in Chapter 6 which concludes this part of the thesis and with that the definition of U-ISC/Graph.

4 Fragment Collaborations (FraCols)

This chapter introduces fragment collaborations (FraCols) that are the basis of each composition system for MDSD. The fragment collaboration concept, which is based on role modelling, is described and a specification language for fragment collaborations is defined.

This chapter, together with Chapters 5 and 6, defines Universal Invasive Software Composition for Graph Fragments (U-ISC/Graph). The three major new concepts in U-ISC/Graph, compared to earlier ISC approaches (cf. Chapter 3), are: (1) the fragment collaboration concept as means to define modularity concepts, defined in this chapter, (2) support for graph fragments, as opposed to tree fragments, defined in Chapter 5, (3) a universal composition language, as base to support arbitrary composition languages, defined in Chapter 6. With that, the three chapters define solutions for the eight features required for the generic composition system for CB-MDSD presented in Figure 3.2. The first feature, *fragment collaborations* (FraCols), is introduced in this chapter (Figure 4.1) as basis for the other features. Features 2–4 and 5–8 are covered in Chapters 5 and 6 respectively.

The purpose of composition interfaces is to define a contract between components. To connect two components via composition interfaces, they have to fulfil their respective contracts. FraCols are for defining such contracts between components but on a language-independent level. That is, the contracts are defined without knowing the concrete component or the component language in which the component is or will be defined.

As such, FraCols are not based on modularity concepts defined inside a language—for example, Java Class or UML Package—but rather define language-independent modularity concepts. This decouples composition interfaces from language constructs (i.e., from metaclasses if the language is defined with EMOF; cf. Section 5.1.1). Only later is a FraCol coupled to one or multiple languages by binding the language-independent modularity concepts defined by the FraCol to concepts of the languages. (This is discussed in Chapters 5 and 6.)

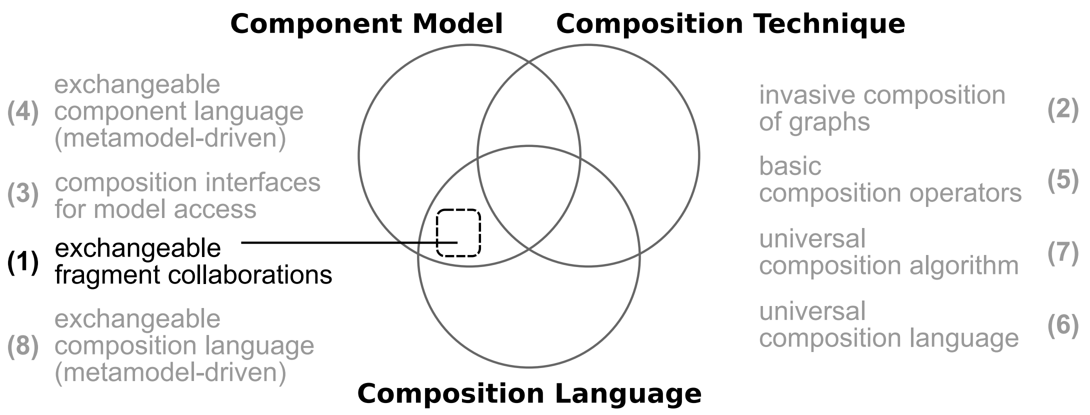

Component Model
Composition Technique
Composition Language
(4) exchangeable component language (metamodel-driven)
(3) composition interfaces for model access
(1) exchangeable fragment collaborations
(8) exchangeable composition language (metamodel-driven)
invasive composition of graphs (2)
basic composition operators (5)
universal composition algorithm (7)
universal composition language (6)

Figure 4.1: Fragment collaboration feature of the generic composition system for CB-MDSD.

To understand the need for FraCols, we recall the example from Section 2.1 that models a ticket shop system with textual use case descriptions and UML use case diagrams. From these models, composition programs are extracted that compose a Java implementation.

In Section 2.1.1, we formulated the requirement for abstract composition interfaces (Requirement 3; p. 16) when we discussed that information has to be transformed into different formats. In the example, information from the textual use case description in OpenDocument format had to be transformed into UML use case diagrams and into Java. For this, a composition program was extracted from the textual use case description. We argued, that this extraction can be reused by using the composition program for both the composition of UML use case fragments and Java fragments. Since the composition program relies on the interfaces of the fragments it composes, these interfaces may not change if the set of UML fragments is exchanged for a set of Java fragments. Therefore, abstract interfaces, which are component language-independent, are required. Fragment collaborations define such abstract interfaces.

Requirement 6, formulated in Section 2.1.3, is also related to FraCols. In this section, we discussed that different kinds of compositions need to be performed in parallel. In the example, we performed a composition of fragments representing actors with a fragment that represents a use case. Then, we needed to add also business values, which were represented by different fragments that had to be composed with the actor fragments in a different way. Thus, the actor fragments needed to support two different modularity concepts—one for the composition with use cases and one for the composition with business values. Consequently, is must be possible that multiple FraCols are supported by a fragment.

Figure 4.1 shows that FraCols concern the component model and the composition language of our generic composition system for CB-MDSD. They concern the component model, because they define modularity concepts that will have to be realised by component languages. They concern the composition language, because composition programs can be formulated on the base of FraCols, which keeps the fragments and the component languages exchangeable. FraCols do *not* concern the composition technique, which is the major difference to existing ISC approaches that do not define a comparable concept (cf. Chapter 3). There, composition interfaces always coupled all three parts—component model, composition language, and composition technique—of a composition system, because they are used directly by the com-

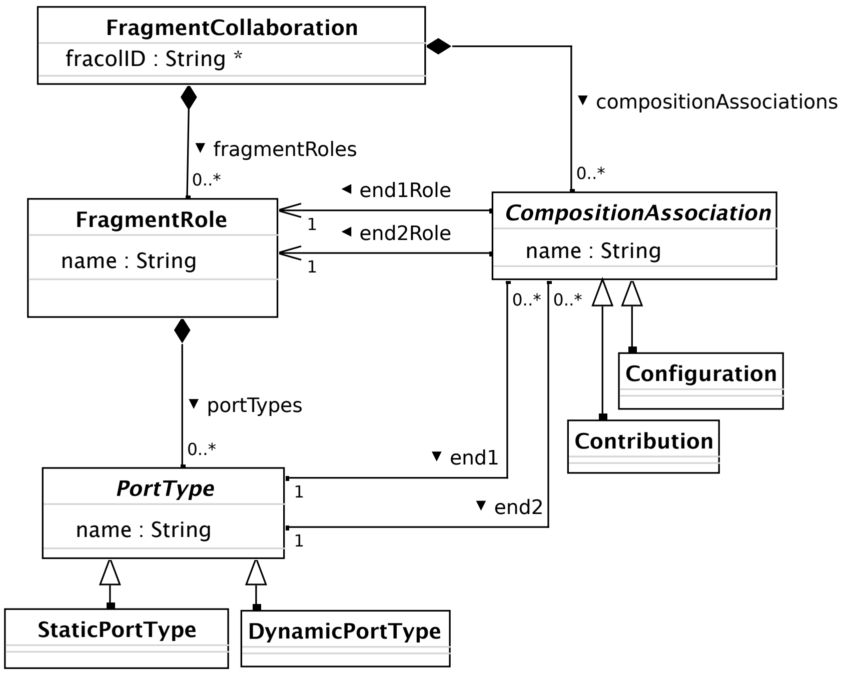


Figure 4.2: Fragment Collaboration Concepts.

position engine to access fragments. In our case, FraCols are used to couple component model and composition language independent of the composition technique and the physical access interfaces of fragments (which we describe in Chapter 5).

In the following, we define the FraCol concepts in Section 4.1 and, based on them, a definition language for FraCols in Section 4.2.

4.1 Fragment Collaborations based on Role Modelling

To achieve that FraCols can be aggregated, we use the role modelling concepts *role* and *collaboration* [RG98]—hence the name *fragment collaborations*. Roles separate features from an entity which it only holds temporarily when it collaborates with other entities by *playing* a role. This is comparable to the use of the term role in acting. In our case, the entities are model fragments which can collaborate in a composition.

The role modelling concepts for FraCols are presented as class diagram in Figure 4.2. A FraCol consists of a set of *fragment roles*[1] and *composition associations*. Composition associations connect fragment roles through *port types*, which are distinguishable points a fragment has to offer on its composition interface when it plays the corresponding fragment role. The distinction between *configurations* and *contributions* as well as *static* and *dynamic* port types is explained below, where we describe the different FraCol concepts in more detail.

[1]Some role modelling approaches make a distinction between a *role*, which is played by an entity, and a *role type*, which defines properties for a set of similar roles played by multiple entities. In this sense, fragment roles are role types (consult [Ste00] for an overview of different role modelling approaches).

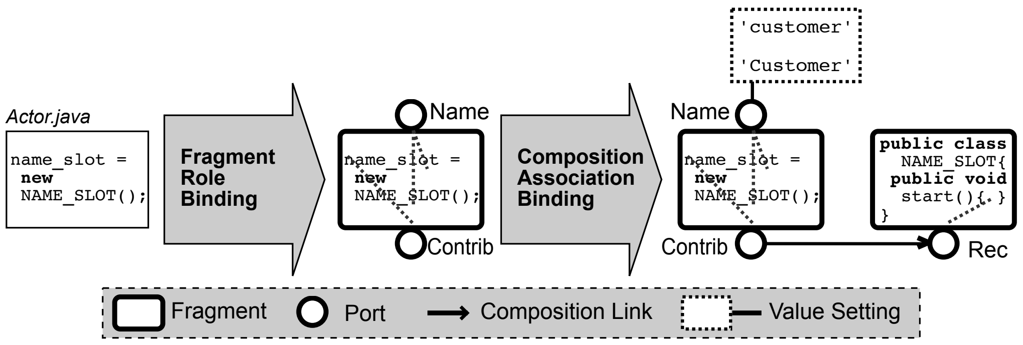

'customer'
'Customer'
Actor.java
name_slot = new NAME_SLOT();
Fragment Role Binding
Name
Contrib
Composition Association Binding
public class NAME_SLOT{ public void start(){ } }
Rec
Fragment
Port
Composition Link
Value Setting

Figure 4.3: FraCol binding process.

Attaching a role to an entity is called *role binding.* In our case, binding a fragment role to a fragment means that the fragment has to offer a composition interface defined in the role. Concretely, the fragment needs to offer *ports* for the port types of the bound fragment role. Fragment role binding is illustrated in Figure 4.3 on a Java fragment from the example of Section 2.1. On the left side, we see the fragment—a snippet of Java code. The fragment role binding adds a composition interface with ports to the fragment that point to places inside the fragment. These places are code snippets that can be reused (e.g., the complete `name_slot = new NAME_SLOT();` statement at which the Contrib port points) or slots (e.g. the `name_slot` and `NAME_SLOT` strings at which the Name port points) where code snippets can be inserted. How these pointers into fragments are established exactly is discussed in Chapter 5. For now, it is important that a fragment role binding creates these pointers and with that connects the composition interface defined in a fragment role with the internals of a fragment. By this, the fragment plays the fragment role.

The composition associations define which ports may be connected and thus determine which *composition links*, and therewith which collaborations, are allowed. The right side of Figure 4.3 shows a composition association binding, which is the creation of a composition link between two ports (Contrib and Rec in the example). The ports relate to port types that are connected by a composition association. Ports with port types that are not connected by a composition association can only be bound to a set of primitive string values. This is illustrated in Figure 4.3, where the Name port is bound to the string values `'customer'` and `'Customer'`. This binding is called a *value setting.* The sum of composition links and value settings makes up a composition program. We consider two possibilities to obtain such composition programs (i.e., perform composition association bindings):

1. Manual composition program definition by using a composition language that supports direct definition of composition links and value settings.
2. Extraction of composition programs from models.

In the following, we describe the details of FraCols and give examples, where we indicate the definition and extraction of composition programs. How composition programs and rules for composition program extraction are defined concretely is discussed in Chapter 6.

Example 4.1. Figure 4.4 depicts the FraCol Participation that can be used for the composition of actors and use cases in the example of Section 2.1. In Figure 4.5, the binding of the roles

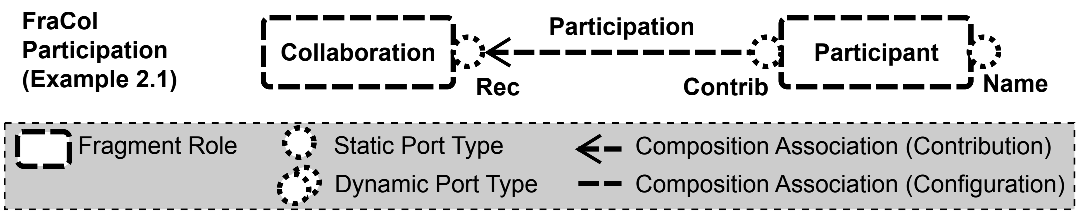
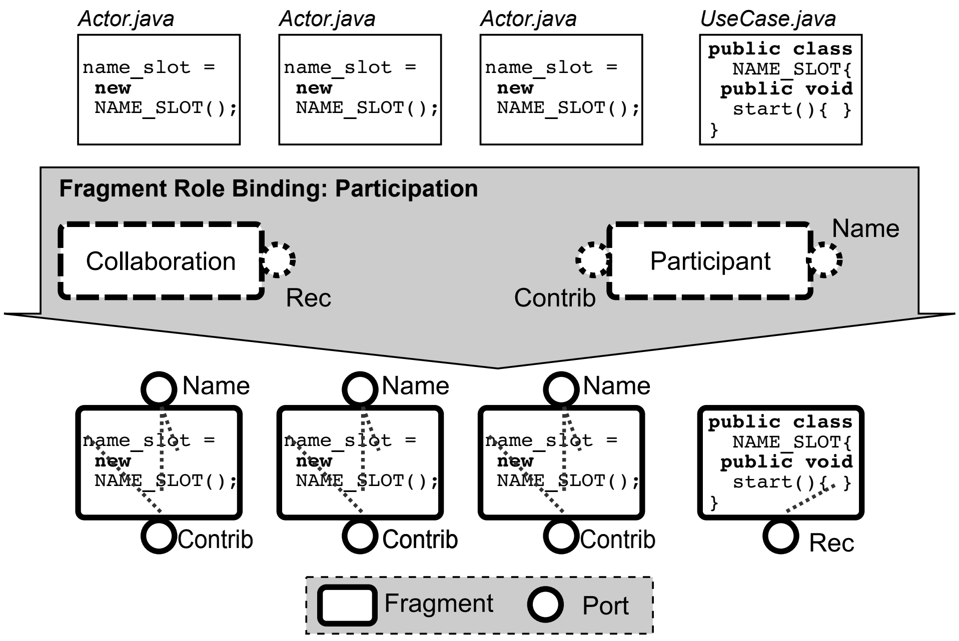

FraCol Participation (Example 2.1)
Collaboration
Rec
Participation
Contrib
Participant
Name
Fragment Role
Static Port Type
Dynamic Port Type
Composition Association (Contribution)
Composition Association (Configuration)

Figure 4.4: FraCol Participation required for the examples of Chapter 2.



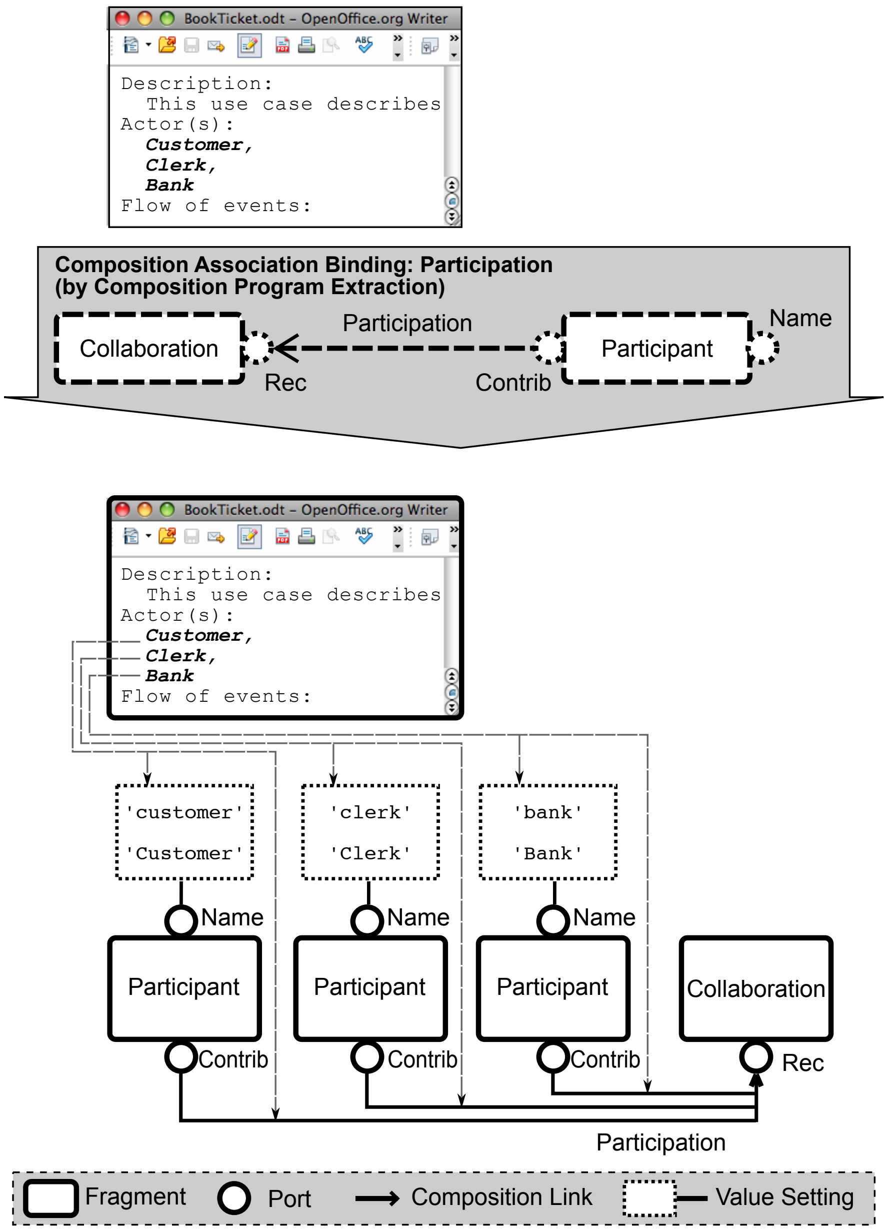
```
name_slot =
  new
  NAME_SLOT();
```

```
public class
  NAME_SLOT{
 public void
  start(){ }
}
```

Figure 4.5: Fragment role binding for Java fragments (cf. Figure 2.5) based on the FraCol Participation (cf. Figure 4.4).

defined in the FraCol to the fragments is shown. Before binding, the fragments do not offer any composition interfaces. After the fragment roles are bound, they have interfaces that point to places within the fragment that are modified in case of a composition. In the example, the first three fragments representing actors take up the Participant role, while the use case fragment takes up the Collaboration role. A fragment role can be bound differently to different fragments. In the example, Java fragments are used, but UML fragments, as in Figure 2.4, may be used as well. For different fragment types, different pointers into the fragment are established during the binding. How this is defined is discussed in Chapter 5.

The extraction of a composition program that uses the same FraCol is shown in Figure 4.6. Here, the FraCol controls the extraction of the composition program from the textual use case description. In the example, we obtain three composition links that conform to the Participation association and three value settings that extract the names of actors (as lower and upper case versions) from the textual use case description.



BookTicket.odt – OpenOffice.org Writer

```
Description:
  This use case describes
Actor(s):
  Customer,
  Clerk,
  Bank
Flow of events:
```

BookTicket.odt – OpenOffice.org Writer

```
Description:
  This use case describes
Actor(s):
  Customer,
  Clerk,
  Bank
Flow of events:
```

Figure 4.6: Composition program extraction from a textual use case description (cf. Figure 2.4) based on the FraCol Participation (cf. Figure 4.4).

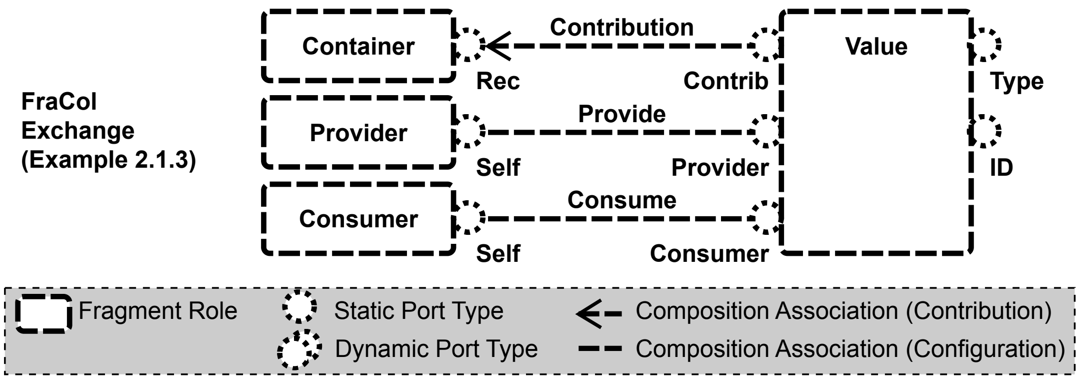
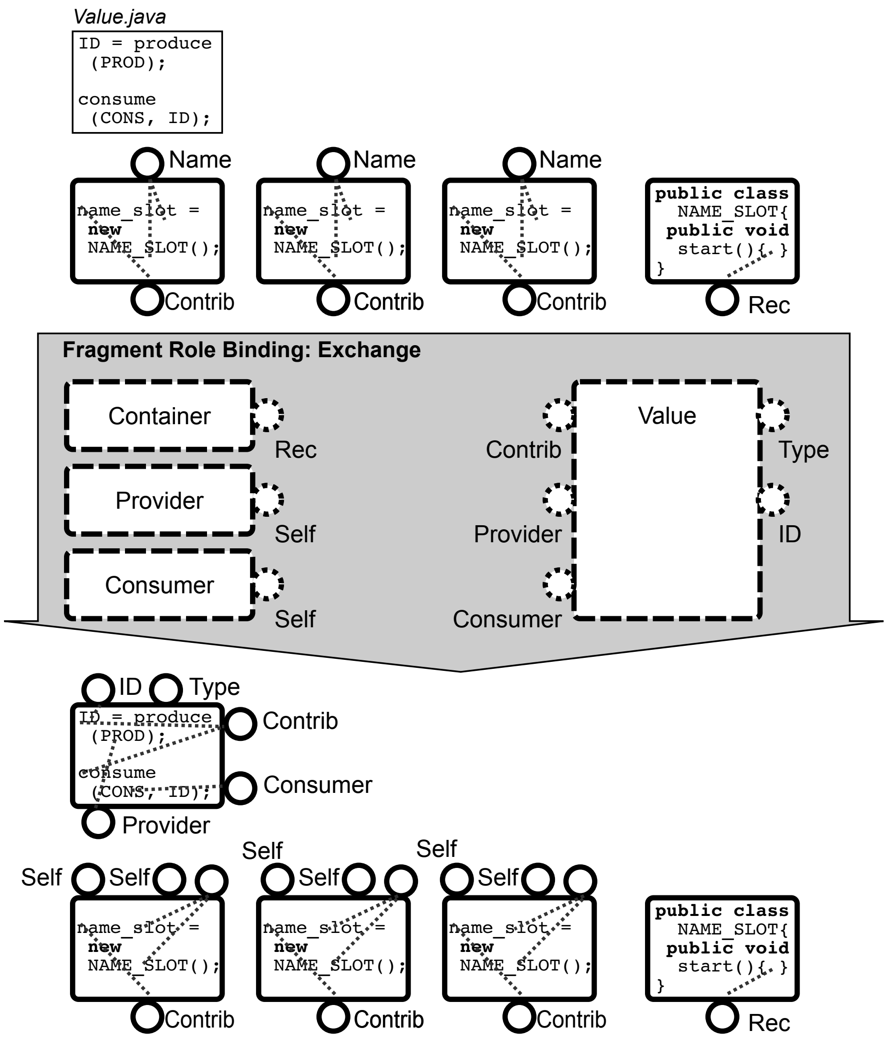


Figure 4.7: FraCol Exchange required for the examples of Chapter 2.



```
ID = produce
 (PROD);

consume
 (CONS, ID);
```

```
name_slot =
 new
 NAME_SLOT();
```

```
public class
  NAME_SLOT{
 public void
  start(){ }
}
```

Figure 4.8: Multiple fragment role bindings (cf. Figure 4.5).

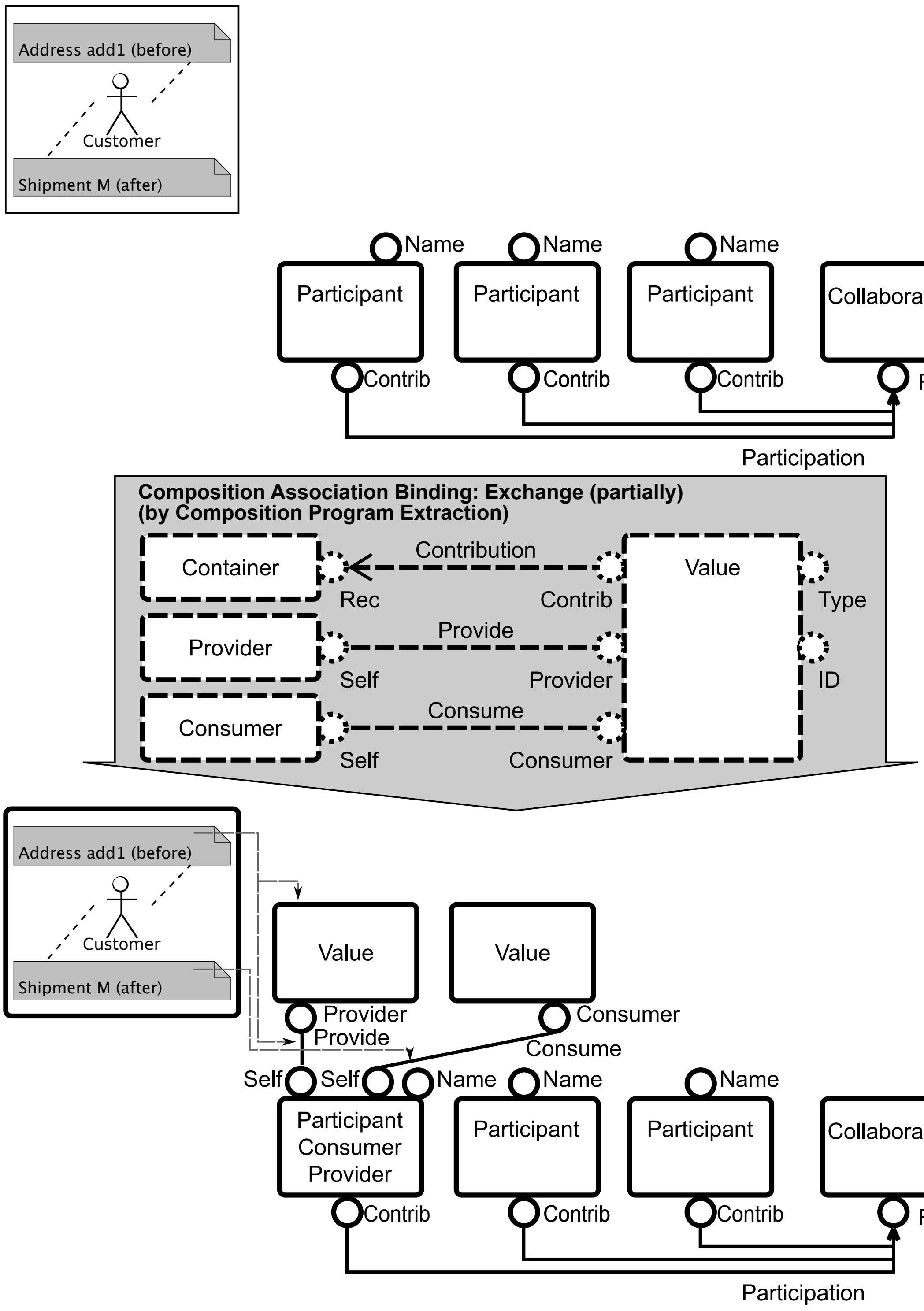


Figure 4.9: Composition program extraction from multiple sources with multiple FraCols (cf. Figure 4.6).

If we combine Figures 4.5 and 4.6, we obtain complete FraCol applications that contains both the connection to composition interfaces of fragments, which are needed to access selected parts of the fragments for composition, and the complete composition program, which is needed to identify which parts of the fragments to merge in the composition.

4.1.1 Fragment Roles

Fragment roles have three important properties. First, when bound multiple times to one fragment, the ports resulting from these bindings are merged. In the example above, the Collaboration role was bound three times to the use case fragment. This binding produced one Rec port and not three Rec ports (right side of Figures 4.5 and 4.6). All Participation composition links use this one port (Figure 4.6).

Second, a fragment can play multiple roles defined in multiple FraCols. This is illustrated in Figures 4.8 and 4.9, where new ports and composition links are added by binding the roles defined in the FraCol Exchange (Figure 4.7) to compose actors with business values. From the fragment perspective, this means that new composition interfaces become available pointing at different positions in the fragment. From the composition program perspective, new links, and with that new collaborations between fragments, are established.

Third, as Figure 4.9 illustrates, a composition program may only require parts of a role to be played by a fragment. While eventually the complete role is required, this allows stepwise binding if the composition program is extracted from multiple sources (which is needed to meet Requirement 5; p. 17). In the example, we would have to add additional UML use case diagrams with business value annotations, as in Figure 2.7, to complete the binding.

4.1.2 Port Types

We distinguish *static* and *dynamic* port types. The port types used in the examples so far were all of *static* nature. This means that there is always one port of that type on a fragment that plays the corresponding role.

Dynamic port types on the contrary allow multiple ports of the same type on one fragment. In this case, the role binding controls which and how many ports of that type are available on one concrete fragment and the number can vary from fragment to fragment. Consequently, for dynamic port types, it is not known from the fragment role specification, how many ports will exist on a fragment. This weakens the contract defined by the fragment role and a composition program extraction requires knowledge about concrete fragments to establish composition links. On the upside, dynamic port types give more flexibility to the developers of fragments and are hence useful in certain situations as the following example illustrates.

Example 4.1.2. Figure 4.11 shows the binding of fragment roles with dynamic port types in the file system example introduced in Section 2.2. The corresponding FraCol ClassWeaving, shown in Figure 4.10, models the core/aspect relationship described for the COMPOST realisation of the example in Section 3.1. In a fragment playing the Core role, each class is individually extensible. Hence, there is a JoinPoint port for each class. In a fragment playing the Advice role, each class offers extension content. Hence, there is a Content port for each class. If new classes are added to one of the example models, the interface dynamically expands with addition JoinPoint or Content ports.

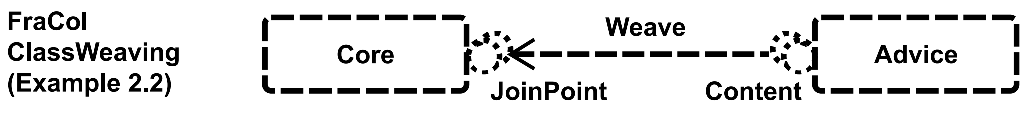


Figure 4.10: FraCol ClassWeaving required for the examples of Chapter 2.

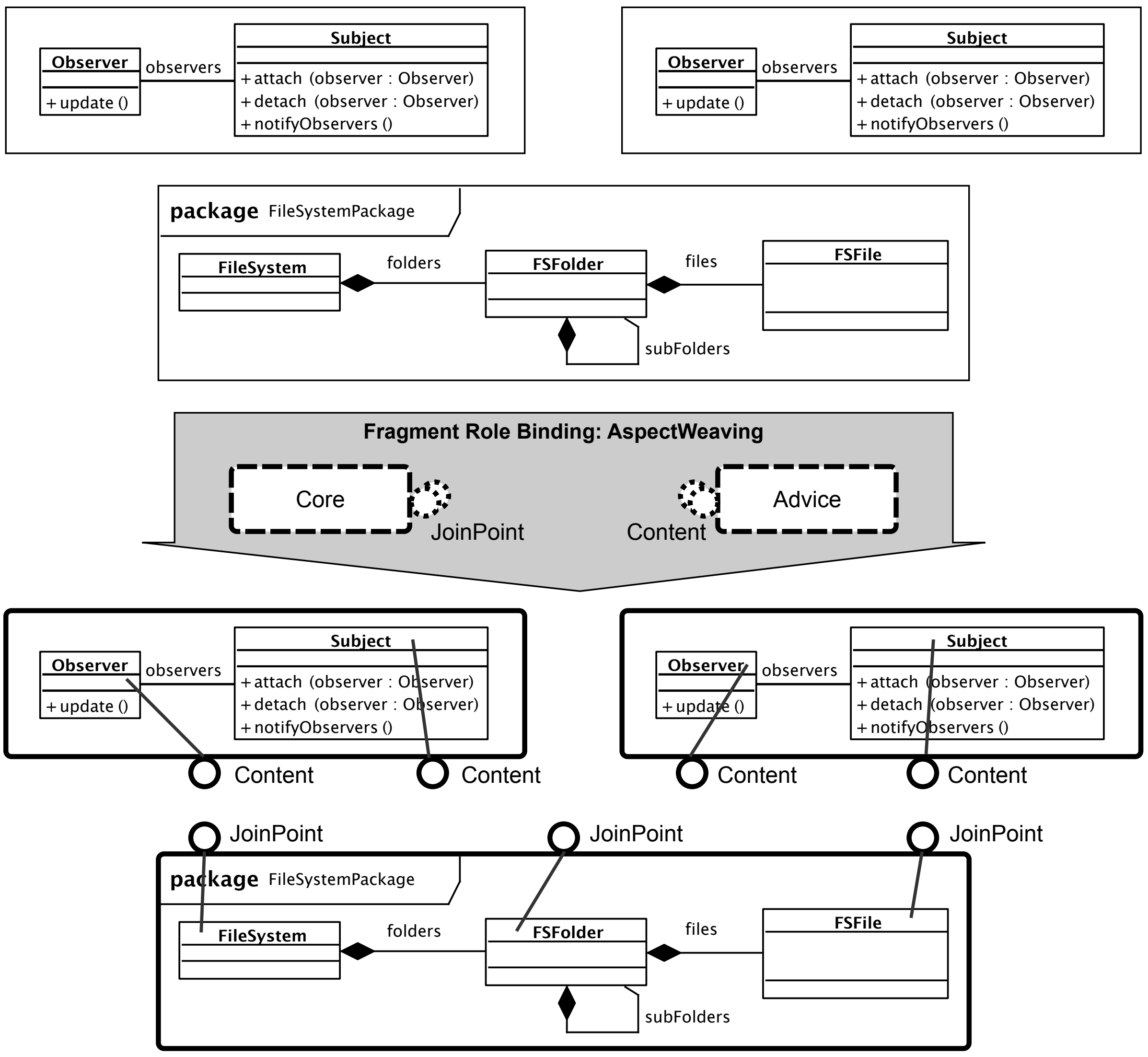


Figure 4.11: Fragment role binding that involves dynamic ports.

```
 1 fracol ID {
 2   fragment role fragmentRoleName {
 3     static  port type portName;
 4     dynamic port type portName;
 5   }
 6   contributing association associationName {
 7     fragmentRoleName.portName --> fragmentRoleName.portName
 8   }
 9   configuring association associationName {
10     fragmentRoleName.portName --> fragmentRoleName.portName
11   }
12 }
```

Listing 4.1: Schematic FraCol specification.

In Section 2.2, we argued that in cases like this example, a universal composition language can be used directly. With such a language, one can define the links based on concrete fragments for which the role binding was already performed (as in Figure 4.11) and the ports with dynamic type are known. (We define this universal composition language in Chapter 6.)

4.1.3 Composition Associations

We distinguish two types of composition associations—*contributions* and *configurations*—to control the composition direction. This is necessary because eventually a composition is executed that merges fragments. *Contributions* are directed and define which fragment is merged into which. That is, the fragment the contribution is pointing at is extended with content from the other fragment. *Configurations*, which are not directed, define additional adjustments inside the merged content of fragments already merged by contributions.

This puts two restrictions on the aggregatability of FraCols. First, FraCols may not be aggregated in a way that the complete composition program contains cycles resulting from contributions. Second, if a FraCol contains only configurations, it needs to be aggregated with FraCols that contain contributions to determine the composition direction.

4.2 FraCol Specification Language

In this section, we present the FraCol specification language. For this language, we use the FraCol concepts defined in Figure 4.2 as metamodel and add a textual syntax to it. With this syntax, developers can define FraCols using the concepts introduced in Section 4.1. (The complete grammar of the FraCol specification language can be found in Section 7.2.)

In the following, the syntax is explained on a schematic FraCol definition shown in Listing 4.1. In Line 1, an ID is given to the specification. Lines 2–5 show the definition of a fragment role. It consists of static (Line 3) and dynamic (Line 4) port type definitions. A contributing association is defined in Lines 6–8 and a configuring association in Lines 9–11. These definitions state which two port types defined above can be connected.

Example 4.2. Listings 4.2, 4.3, and 4.4 show the specifications of the FraCols shown in Figures 4.4, 4.7, and 4.10 respectively, which were discussed in this chapter.

```
fracol org.reuseware.lib.systems.participation.participation {
  fragment role Participant {
    static port type Contrib;
    static port type Name;
  }
  fragment role Collaboration {
    static port type Rec;
  }
  contributing association Participation {
    Participant.Contrib --> Collaboration.Rec
  }
}
```

Listing 4.2: FraCol specification for Participation (cf. Figure 4.4).

```
fracol org.reuseware.lib.systems.exchange.exchange {
  fragment role Container {
    static port type Rec;
  }
  fragment role Value {
    static port type Contrib;
    static port type Provider;
    static port type Consumer;
    static port type Type;
    static port type ID;
  }
  fragment role Provider {
    static port type Self;
  }
  fragment role Consumer {
    static port type Self;
  }
  contributing association Contribution {
     Value.Contrib --> Container.Rec
  }
  configuring association Provide {
     Value.Provider --> Provider.Self
  }
  configuring association Consume {
     Value.Consumer --> Consumer.Self
  }
}
```

Listing 4.3: FraCol specification for Exchange (cf. Figure 4.7).

```
fracol org.reuseware.example.class_weaving {
  fragment role Core {
    dynamic port type JoinPoint;
  }
  fragment role Advice {
    dynamic port type AdviceContent;
  }
  contributing association Weave {
    Advice.AdviceContent --> Core.JoinPoint
  }
}
```

Listing 4.4: FraCol specification for AspectWeaving (cf. Figure 4.10).

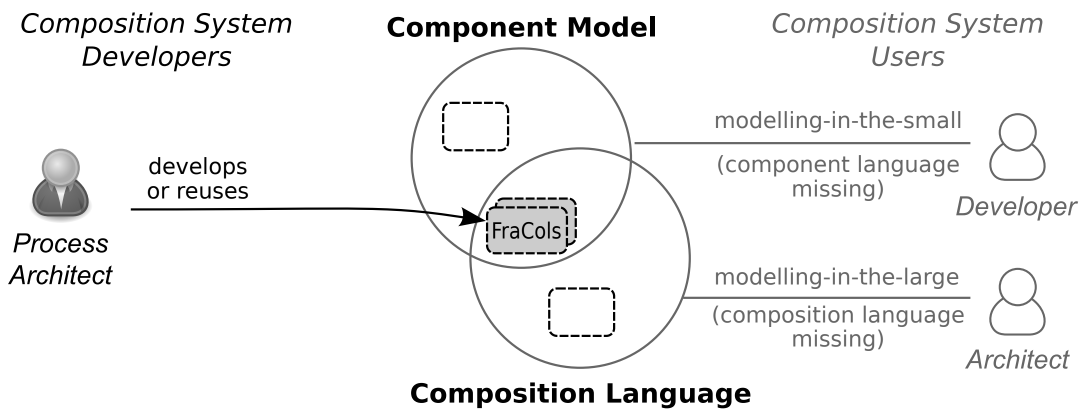


Figure 4.12: FraCols configure the generic CB-MDSD composition system.

4.3 Conclusion

Chapter Contribution This chapter contributes the following:

C1-1 **Contracts between component and composition languages**
FraCols provide the first extension of ISC for explicit contract definition between component languages—as part of a component model—and composition languages in ISC systems. It is therefore the first method to couple component models and composition languages independent of the composition technique of such systems. This contribution has the following sub-contributions:

- *Application of role modelling on the language level*
  For FraCols, we successfully transferred the concepts of role modelling to the level of language development to define contracts between component and composition languages.
- *Specification Language for FraCols*
  As a technical contribution, we provided a FraCols specification language that can be integrated into development environments for invasive composition systems.

The FraCol specification language is the first tool a process architect, in the role of a composition system developer, can use to configure the generic CB-MDSD composition system. This is illustrated in Figure 4.12. To configure the system further, means are required to integrate component and composition languages. These integrations are performed by fragment role binding and composition program extraction which are not substantiated in this chapter. The fragment role binding, which concerns component languages and therewith the component model, is discussed in the next chapter and the composition program extraction, which concerns the composition language, is discussed in Chapter 6.

5 Graph Fragments

This chapter describes how models are treated as graph fragments. It defines variability concepts for graph fragments and introduces a language for component model configuration through which modelling languages can be integrated as component languages into a CB-MDSD composition system.

The previous chapter introduced fragment collaborations (FraCols) to ensure exchangeability of component and composition languages in composition systems for CB-MDSD. As next step towards a generic CB-MDSD composition system (cf. Figure 3.2), this chapter deals with component languages, which are in our case arbitrary DSMLs defined in EMOF [OMG06a]. As Figure 5.1 illustrates, this concerns three features of the generic CB-MDSD composition system:

- (Feature 2) To establish the basics for invasive graph composition as composition technique, we explore how models are treated as graphs and which specifics of such graphs can be exploited for their composition. This is covered in Section 5.1.
- (Feature 3) For accessing models represented as graphs through composition interfaces, concepts are required which can be superimposed on graphs to mark access points. We call these concepts, which we present in Section 5.2, variability types that extend the hook concept of ISC.
- (Feature 4) Finally, to establish composition interfaces for models of arbitrary DSMLs, we require a method to define variability types for models and map these to modularity concepts defined in FraCols. This method is the fragment role binding we indicated in Chapter 4. For that, we define the component model configuration language REX_{CM} in which such fragment role bindings are expressed. Figure 5.2 shows that REX_{CM} can be used by a process architect, in the role of a composition system developer, to configure the component model of the generic CB-MDSD composition system to integrate new DSMLs as component languages. This in turn allows developers to develop model components in that DSML. REX_{CM} is defined in Section 5.3.

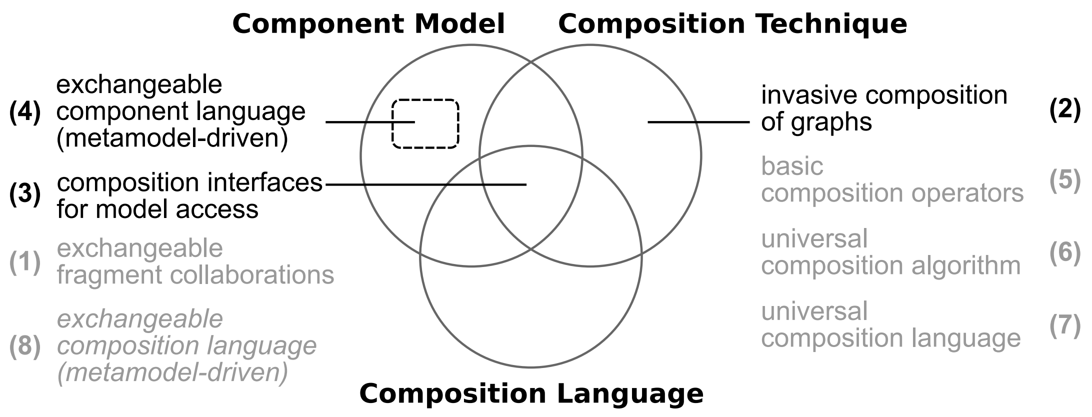

Component Model
Composition Technique
Composition Language
(4) exchangeable component language (metamodel-driven)
(3) composition interfaces for model access
(1) exchangeable fragment collaborations
(8) exchangeable composition language (metamodel-driven)
invasive composition of graphs (2)
basic composition operators (5)
universal composition algorithm (6)
universal composition language (7)

Figure 5.1: Features of the generic composition system for CB-MDSD that concern graph fragments.

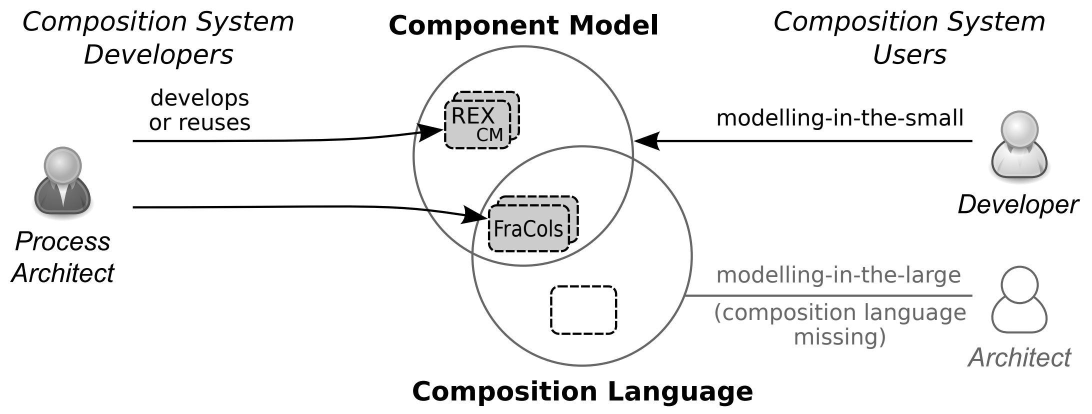


Figure 5.2: REX$_{\mathrm{CM}}$ definitions configure the component model of the generic CB-MDSD composition system.

5.1 Models as Graphs

Models in model-driven development can be represented as graphs. We would like to realise our composition technique as manipulation of these graphs. For this, the composition technique of U-ISC/Tree, which is based on tree manipulation, should be extended. To understand how models can be represented and manipulated as graphs, we first have to understand how graph representations of models are obtained, what their specifics are, and how they can be accessed for manipulation.

Example 5.1. As illustration that any model can be treated as graphs consider the two models shown in Figures 5.3 and 5.4. The first figure shows the UML model of the file system introduced in Section 2.2. The second figure shows the similar system defined in a Java model[1]

[1] As mentioned, we treat all languages defined with EMOF as DSMLs. This includes Java which we further discuss in Section 5.1.2. Consequently, we refer to all artefacts defined in DSMLs as models and do therefore not distinguish between models and code.

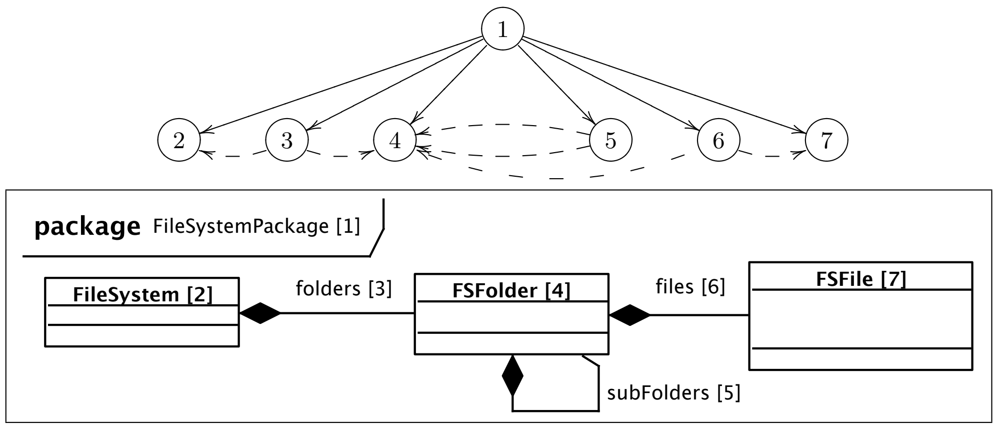
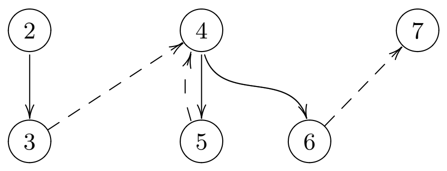


Figure 5.3: A file system modelled in UML represented as diagram and as graph.



```
public class FileSystem2 {              public class FSFolder4 {                  public class FSFile7 {
    protected FSFolder4[] folders3;         protected FSFolder4[] subFolders5;    }
}                                           protected FSFile7[] files6;
                                        }
```

Figure 5.4: A file system modelled in Java represented as text and as graph.

as introduced in Section 3.1. The elements in the model are numbered to relate them to the nodes in the graph that represent them.

Each figure shows the model in its concrete syntax, which is a diagram in the case of UML and a text in the case of Java. The models can be transformed from their concrete syntax representation into a graph and back, which allows manipulation on the graph that reflects in the concrete syntax representation. For this, mappings between concrete syntax and graph representations are required, which differ from DSML to DSML. Such a mapping maps a concept of a DSML to its representation. For instance, in the case of UML, a class is represented as a rectangle, while in the case of Java a class is represented as a text block.

We are interested in manipulating the graph representations of models. For that, we have to explore how graph representations of models, which are defined in EMOF-based DSMLs, are obtained and how modifications on these graphs are performed. Therefore, we discuss how DSMLs are defined in EMOF (Section 5.1.1), how concrete syntax is defined on top of that (Section 5.1.2), and which facilities exist to access parts of the graph representations of models (Section 5.1.3).

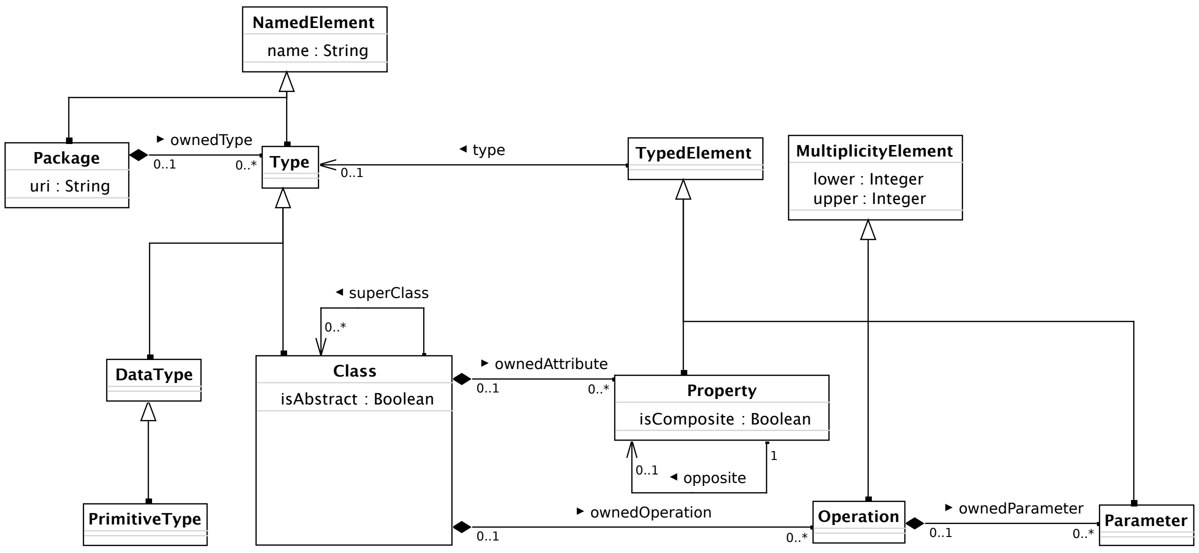


Figure 5.5: EMOF core modelled in EMOF/Ecore (based on Figures 12.2–12.5 in [OMG06a]).

5.1.1 The Essential Meta Object Facility (EMOF)

The Meta Object Facility (MOF) is a standardised metalanguage (or meta-metamodel) to specify DSMLs. Such The first MOF standard (MOF 1.1) was released by the OMG in 1997. MOF is based on UML reusing its class diagram package [OMG10b, Chapter 7]. MOF is used for the specification of modelling language standards including UML and MOF itself.

Since MOF 2.0 [OMG06a] (released in 2006) a kernel called Essential MOF (EMOF) was singled out from the Complete MOF (CMOF). According to the specification, EMOF provides a minimal set of elements to model object-oriented systems. It is therefore easier to map EMOF specifications to object-oriented implementations (e.g., in Java) than to map CMOF specifications. The de-facto reference implementation of EMOF is Ecore [SBPM09] in the Eclipse Modeling Framework (EMF). The first version of Ecore was created prior to MOF 2.0 and had an important influence on the design of EMOF.

Despite minor naming and structural differences, EMOF and Ecore are equally expressive [SBPM09, Section 15.4.5]. In this thesis, we mostly stick to the Ecore terminology because it allows us to transfer the conceptual specification directly to the implementation. Below, we introduce the central concepts of EMOF and show the differences between EMOF and Ecore.

We use EMOF/Ecore for two purposes. First, we use it as metalanguage to describe DSMLs which are used as component and composition languages in CB-MDSD composition systems. Second, we specify the concepts and languages of U-ISC/Graph with EMOF/Ecore and generate the corresponding implementations as part of Reuseware (cf. Chapter 7) from it.

From a scientific point of view, the choice of EMOF/Ecore as metalanguage is not important. Our results can be transferred to other metalanguages (e.g, Resource Description Framework Schema (RDFS) [W3C04]). In Section 5.1.3, we will therefore define general concepts and map them to EMOF/Ecore. By mapping these general concepts to other metalanguages, our results can be transferred.

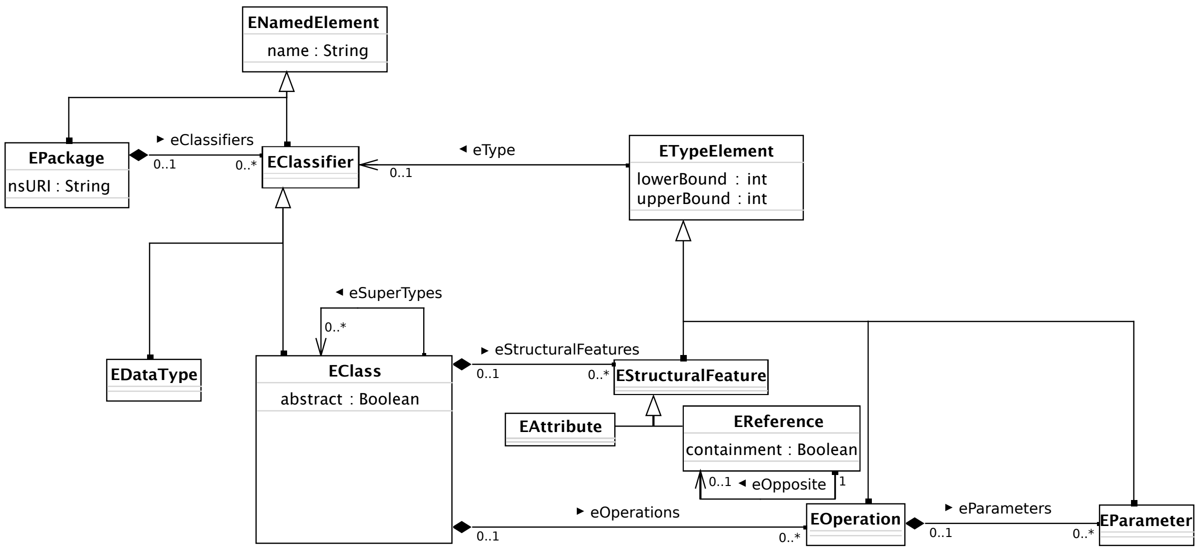


Figure 5.6: Ecore core modelled in EMOF/Ecore (based on Figures 5.1–5.9 in [SBPM09]).

EMOF Concepts

Figures 5.5 and 5.6 show the central concepts of EMOF specified in common UML class diagram notation (i.e., EMOF itself; the graphical notation is shown in Figure 5.7 and described below). Figure 5.5 uses the terminology of the EMOF specification and Figure 5.6 of the Ecore implementation. The EMOF concepts should be familiar since they resemble the base concepts of object-oriented programming languages. We therefore explain briefly Figures 5.5 and 5.6 in parallel using the schema *EMOF terminology (Ecore terminology)*.

A metamodel specifies a set of `Types` (`EClassifiers`) which are collected in a `Package` (`EPackage`). Each package has a unique `uri` (`nsURI`) that is the unique identifier of the metamodel. `Types` (`EClassifiers`) are either `DataTypes` (`EDataTypes`) or `Classes` (`EClasses`). A data type holds *raw* data (e.g., a character sequence or a number literal). It is often predefined and reused from a standard library.

The central concept of EMOF is `Class` (`EClass`) to define metaclasses. Each metaclass represents a concept or a construct in the modelled language. Metaclasses can inherit type and features through the subclassing concept known from object-oriented languages. Subclassing is expressed by the `superClass` (`eSuperTypes`) reference. A `Class` (`EClass`) can be `abstract` (`isAbstract`).

Classes have an arbitrary number of `ownedAttributes` (`eStructuralFeatures`) and `owned Operations` (`eOperations`). All features and operations have a `type` (`eType`)—defined by `TypedElement` (`ETypedElement`)—as well as a `lower` (`lowerBound`) and `upper` (`upperBound`) bound—defined by `MultiplicityElement` (`ETypedElement`).

In EMOF, a feature is represented by a `Property`. (In Ecore, `EStructuralFeature` has two sublcasses—`EAttributes`, where `eType` has to be an `EDataType`, and `EReference`, where `eType` has to be an `EClass`.) A `Property` (`EReference`) can have an `opposite` (`eOpposite`) which emulates a bi-directional association.

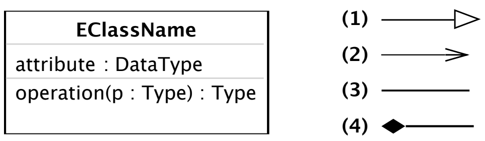


Figure 5.7: Graphical EMOF/Ecore notation.

An `Operation` (`EOperation`) has a set of `ownedParameters` (`eParameters`). Each parameter is also typed and has a multiplicity. EMOF (Ecore) itself does not define a formalism to specify the semantics—the body—of an `Operation` (`EOperation`).

This concludes the description of the EMOF (Ecore) concepts. From now on, we stick to the Ecore terminology. In the following, we discuss the graphical notation of Ecore and give examples. For further details about EMOF and Ecore please consult [OMG06a] and [SBPM09].

Graphical Notation

The graphical notation for Ecore metamodels is summarised in Figure 5.7. The Fujaba [NNZ00] tool can be used to create diagrams using this notation (cf. Chapter 7 for details about the tools we used). The notation follows the common UML class diagram notation [OMG10b, Section 7.4]. However, a few peculiarities to how the notation relates to the Ecore metamodel that should be considered.

The notation consists of boxes (left side of Figure 5.7) with three compartments and lines (right side of Figure 5.7) between the boxes with different end symbols. Each box represents an `EClass`. The first compartment contains the name of the class. The second contains all `EAttributes` of the class (`EAttributes` contained in the `eStructuralFeatures` reference of the class). The third compartment contains the `EOperations` of the `eOperations` reference.

There are basically four types of lines to distinguish. (1) represents a subclassing and therefore sets the `eSuperType` relation between two `EClasses`. All other three lines are instances of either one `EReference` or two `EReferences` which refer to each other as `eOpposite`. (2) stands for a reference from one `EClass` to another, where the first end (no arrow) is the `EClass` containing the `EReference` (in its `eStructuralFeatures`) and the second end (arrow) points at the `eType` of the `EReference`. The `containment` of that reference is set to false. (3) stands for two `EReferences` connected via an `eOpposite` relation where both references have `containment` set to false. (4) identifies two references, where the reference that is contained in the `EClass` at the first end (diamond) has `containment` set to true. The distinction between *containment* and *non-containment* reference is of high importance for our composition approach and in Ecore metamodelling in general, as we will discuss in Section 5.2.

Example 5.1.1 Here we show excerpts of two DSML metamodels which we use in examples throughout this thesis: UML [OMG10b] as realised in [Ecl10d] and Java. The Java metamodel, and the corresponding tooling to create instances of it (cf. Section 5.1.2), was developed during the research for this thesis in collaboration with other PhD projects [HJSW10].

Excerpts of the metamodels are shown in Figures 5.8 (UML) and Figure 5.9 (Java). These excerpts of the metamodels include the class modelling part of UML and Java. Comparing both

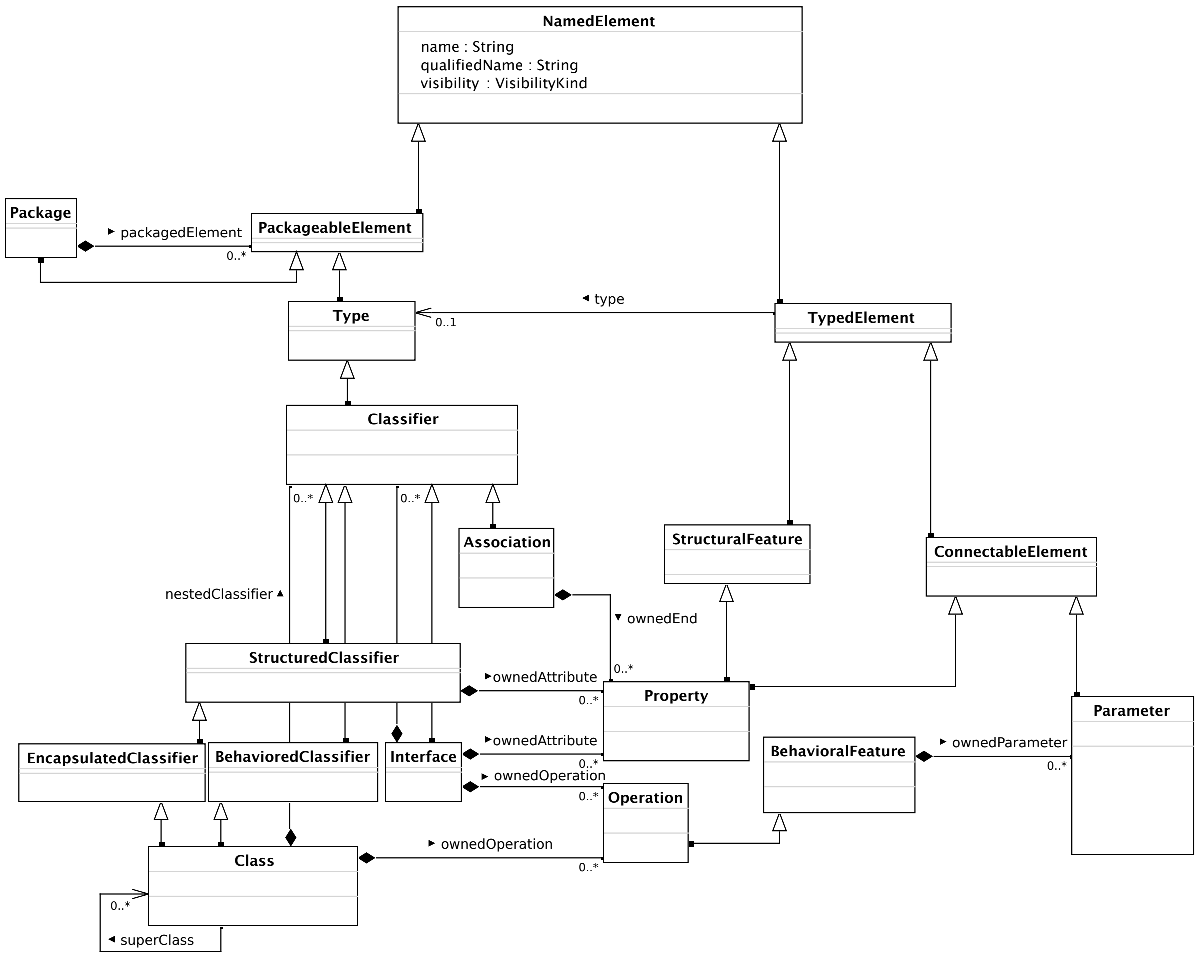


Figure 5.8: An excerpt from the UML metamodel [Ecl10d].

metamodels illustrates how similar concepts can be modelled in different ways with Ecore. For example, Java contains the concept `Field` while UML offers both `Association` and `Property` for a similar purpose. Another example is that the concept of having members is modelled via one containment reference (`members`) in Java, while in UML multiple containment references are used (`ownedAttribute` and `ownedOperation`). A third example is that `TypedElements` in UML directly cross-reference their type via the `type` non-containment reference, while in Java there are explicit reference metaclasses (`TypeReference` and `ClassifierReference` with the `target` non-containment reference to a type).

The binding of metamodels to FraCols, which we present in Section 5.3, needs to be flexible enough to support such structural differences in metamodels. For example, the fragment roles for class weaving presented in Example 4.2 have to be mapped differently to UML and Java due to the structural differences in the metamodels. This is further discussed in Section 5.3.

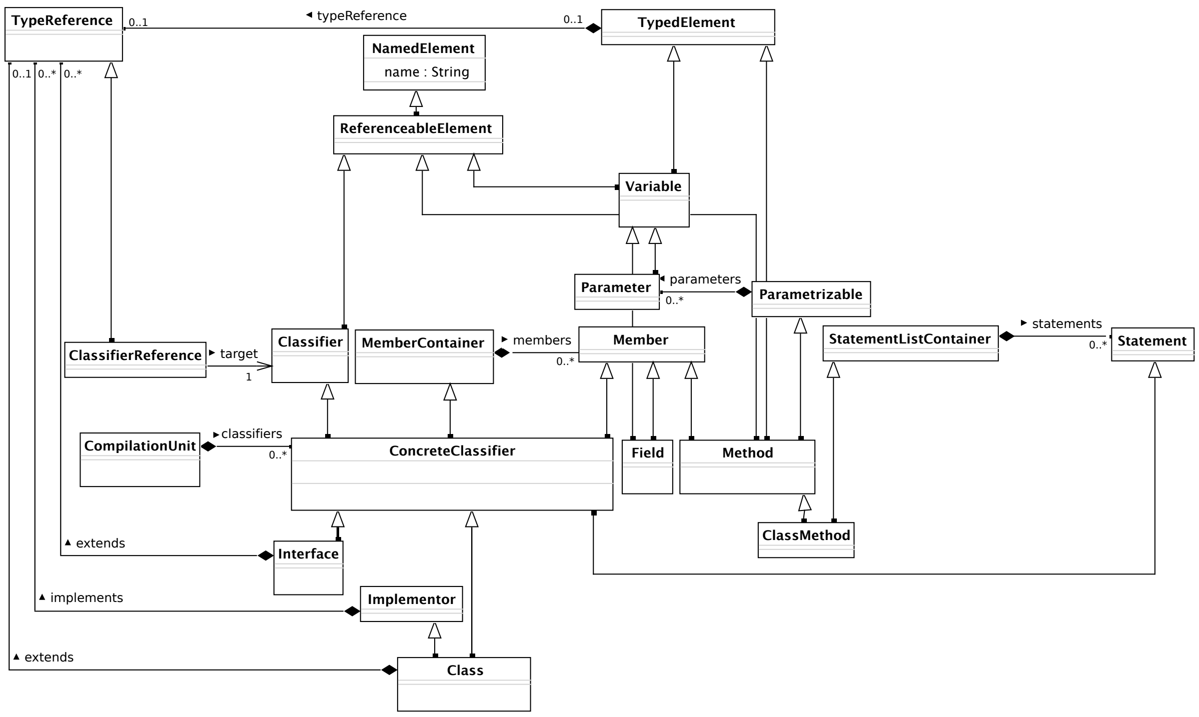


Figure 5.9: An excerpt from the Java metamodel [HJSW10].

5.1.2 Concrete Syntax of EMOF-based Languages

The previous section discussed EMOF/Ecore to specify a DSML by means of its concepts and their relationships. Such a specification is also called *abstract syntax* of a DSML. To allow developers to create models with a DSML, a *concrete syntax* is needed in addition. How such a concrete syntax can be defined on top of an Ecore metamodel is presented in this section.

Below, we discuss graphical syntax and textual syntax. In the latter case, we elaborate on the relationship between Ecore-based languages with textual syntax and grammar-based languages to emphasise that this thesis is an extension U-ISC/Tree (cf. Section 3.2) which worked with grammar-based languages.

Graphical Syntax

Traditionally, models in MDSD have graphical syntax. The syntax is implemented in graphical editors used to create and modify these models. The best example is UML which defines graphical notations for all its constructs and for which many open and commercial graphical editors exist (e.g., [IBM10b, IBM10a, TOP10, Bor10, NoM10, NNZ00]).

A graphical representation is a graph visualisation, where nodes and edges are visualised by shapes and lines. Consequently, layout information, such as position and size of shapes needs to be managed in addition to a model (which is an instance of an Ecore metamodel). Such layout information is usually stored together with the model to preserve the layout a developer has created manually.

In the graphical representation of a model, a node is often represented by one shape and an edge is represented by a line. Nevertheless, there can be more complex relationships between visualisation and model. For example, an edge can also be represented by a shape, or one shape can represent multiple nodes and edges. This depends on how the graphical syntax is defined.

Graphical syntax is often hard-coded into tools. In recent years however, technologies and standards emerged to declare graphical syntax that can be interpreted or from which tools can be generated. Two prominent candidates for the generative approach are GMF [Gro09] and TOPCASED [TOP10] that both work with the EMF and languages defined in Ecore. Furthermore, there are approaches that aim at simplifying the specifications of graphical syntax. One of these approaches is EuGENia [KRPP09]. EuGENia also uses GMF for editor generation in the background, but may target other technologies as well in the future. There is also an OMG standard related to graphical syntax—the UML diagram interchange (UML-DI) standard [OMG06c]—which was designed to make layout information exchangeable between tools. Although both GMF and TOPCASED do not conform to that standard, there are ongoing discussions on how to align them with a future version of the standard.

Graphical model editors, hand-crafted and generated alike, work with the Model-View-Controller principle. That is, the model and the layout information is loaded in memory and edited there by using the dedicated graphical editor. The models are stored and loaded using the XML Metadata Interchange (XMI) [OMG07] format that is an OMG-standardised XML dialect for serialising EMOF-based models. The files in which the models are stored are not intended to be edited directly (i.e., with a text or XML editor).

Example 5.1.2. (1/2) For illustration, consider the file system UML model (cf. Example 5.1) shown again in Figure 5.10. The bottom of the figure shows the model as it appears in the TOPCASED editor. The upper part shows the model as it exists in memory—as a graph. The graph consists of nodes for the model elements (◯). Additionally, layout information (□) exists that contains information about the position of model elements in the graphical view. The model elements themselves are unaware of this layout information.

If we edit the model in the TOPCASED editor, the graph is directly modified. Also, if the graph is modified by other means (e.g., if there is another editor that operates on the same model) the changes are directly shown in the TOPCASED editor. If the model is saved to XMI, the graph structure is serialised as it is in memory and is reconstructed when the model is loaded again.

We used GMF to develop a graphical editor for our universal composition language (cf. Chapter 7). Furthermore, the graphical modelling languages used in examples throughout this thesis and in the evaluation (Chapters 8 and 9) are either based on GMF or TOPCASED. In Chapter 8, EuGENia was used in addition to GMF to specify graphical syntax for two DSMLs.

The question arises how the layout information, which is additional information on top of the actual models, can be treated during invasive graph composition. Although this is not in the core of this work, it was nevertheless an important issue to perform the evaluations in Chapters 8 and 9, because composition results cannot be presented adequately to developers without a certain layout preservation in the composition process. Therefore, we addressed this

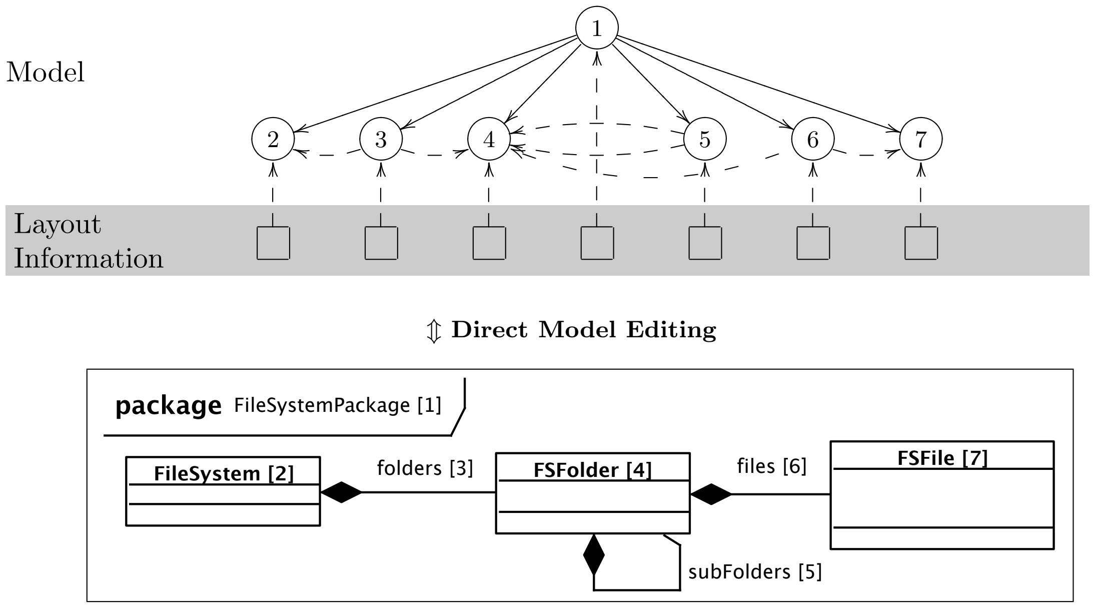


Figure 5.10: UML model directly edited in graphical editor.

issue in the research performed for this thesis. Our results are discussed at the end of this thesis in Section 12.1.

Textual Syntax

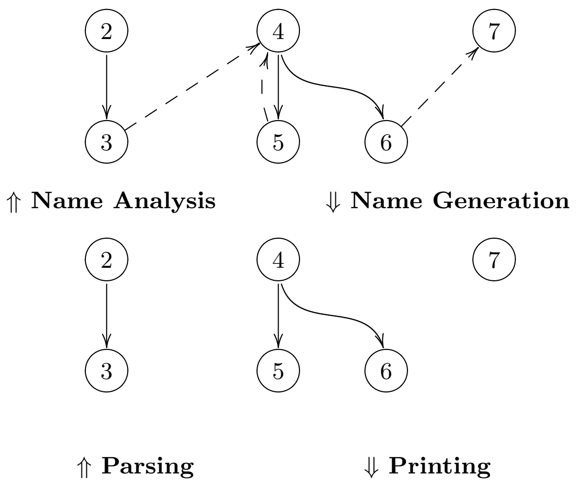
Traditionally, languages in computer science have textual syntax. This originates from the fact that first computers were not capable of displaying graphics. Today, textual languages are still very common—in particular for programming. Compared to graphical syntax, textual syntax has the advantage of a predefined reading order, which is valuable if sentences are interpreted in sequential order. Furthermore, there are many tools that work on the basis of textual representation—for example for comparison, merging, or version control.

Thus, based on the application area of a language, a textual syntax is useful. Textual syntax can also be used as complement to graphical syntax—in particular in the case of Ecore-based languages, where a common abstract syntax exists in form of the Ecore metamodel.

In addition to designing new textual DSMLs, being able to specify textual syntax for Ecore-based languages enables us to describe existing languages in Ecore. The Java metamodel (cf. Figure 5.9), for instance, is useless in practice without being able to transfer Java code written in Java's textual syntax into an instance of the metamodel (and the other way around).

When we started the work on this thesis, there was no tool available that supports the specification of textual syntax for Ecore-based languages in a similar manner as GMF provides it for graphical languages. Therefore, we developed EMFText [HJK$^+$09] in collaboration with other PhD projects.

In principle, EMFText follows the grammar-based language specification idea presented in [Mey90] that distinguishes between abstract and concrete syntax grammars. This idea was also



```
public class FileSystem2 {          public class FSFolder4 {               public class FSFile7 {
    protected FSFolder4[] folders3;     protected FSFolder4[] subFolders5;  }
}                                       protected FSFile7[] files6;
                                    }
```

Figure 5.11: Java model edited by parsing and printing text.

followed in [Hen09] for U-ISC/Tree which was defined on the base of abstract syntax grammars. In EMFText, metamodels are used instead of abstract syntax grammars. In addition, EMFText offers a language to specify concrete syntax grammars on top of metamodels called *Concrete Syntax* (CS). Thus, using EMFText, U-ISC/Graph also works for the textual languages that were addressed by [Hen09].

In contrast to graphical editors, textual model editors—which are generated by EMFText—do not work on the model, but on plain text. Although this has drawbacks, it allows arbitrary text processing tools, such as merge or version control tools, to work on the textual model representations as well. The text can be parsed into a graph and a graph can be printed into its text representation. The tooling for that is generated by EMFText.

Example 5.1.2. (2/2) To illustrate the work with textual syntax compared to graphical syntax, we revisit the file system example defined in Java (cf. Example 5.1) in Figure 5.11. The bottom shows the three Java classes in their textual representation.

To obtain a graph representation, the texts are first parsed into trees (middle of Figure 5.11). Such abstract syntax trees are the structures U-ISC/Tree works on. From the modelling perspective, the containment references (cf. Section 5.1) between model elements are established at this stage, but not the cross-references such as the reference between a `Field` and the `Class` that represents the field's type.

These cross-references are established in a name analysis step that is also performed by the EMFText generated tooling. The name analysis establishes the cross-references such as the reference between the `folders` field (3) and its type `FSFolder` (4) that is identified by its name in the text. The tooling for name analysis might require adjustment if the language has specific scoping rules. We did this adjustment for Java in [HJSW10].

```
1 Class ::= annotationsAndModifiers*
2           "class" name[] ("<" typeParameters ("," typeParameters)* ">")?
3           ("extends" extends)? ("implements" (implements ("," implements)*))?
4           "{" (members)* "}";
5
6 Field ::= annotationsAndModifiers* typeReference arrayDimensionsBefore*
7           ("<" typeArguments ("," typeArguments)* ">")?
8           name[] arrayDimensionsAfter* ("=" initialValue:expressions.AssignmentExpression)?
9           ("," additionalFields)* ";";
```

Listing 5.1: EMFText syntax rule for the Java metaclasses `Class` and `Field` (cf. Figure 5.9).

Once a model is transferred into its graph representation, it can be modified there (e.g., invasively composed). The right side of Figure 5.11 illustrate how to obtain the textual representation of a such modified model by printing the graph back into text. For this, first names have to be generated from all cross-references. Then all elements can be printed according to the textual syntax of the language.

As indicated, Figure 5.11 illustrates one advance of our work over U-ISC/Tree [Hen09]. U-ISC/Tree only works on context-free structures (middle), while our approach works on graph structures with context (top). This guarantees correct compositions of context-sensitive parts of a fragment. For example, in U-ISC/Tree, one can modify the class name `FSFolder` without recognising that it breaks the result because the name is used elsewhere to refer to the class. Working on the graph (top), the names have already been resolved to cross-references and modifying the class name automatically leads to a modification of all usages of the name in the printed text.

All textual languages used in examples throughout this thesis are defined with EMFText. Therefore, we give a quick introduction on the concrete syntax specification language of EMFText called *CS*, which is a variant of EBNF. Listing 5.1 shows two rules of our CS specification for Java [HJSW10]. As mentioned, the specification refers to the Ecore metamodel of the language. For each concrete metaclass, one grammar rule is defined. In Listing 5.1, the rules for `Class` and `Field` are shown (cf. Figure 5.9). On the right-hand side of each rule we find (1) containment references (e.g., `members` in Line 4) that define the positions of contained model elements, (2) attributes (e.g., `name` in Line 2) that define the position of attribute values, and (3) keywords (e.g., "`class`" in Line 2) that are pure concrete syntax and have no counter part in the metaclass. The usual grammar notations of `|`, `?`, `+`, `*` can be used but must follow the multiplicities defined in the corresponding metaclass. For more details, please consult [HJK+09].

We used EMFText to connect Java to EMF and define other textual domain-specific languages in the evaluation (Chapters 8 and 9). In particular, the treatment of Java shows that general-purpose programming languages can be defined in terms of Ecore metamodels and that a distinction between modelling and programming languages is not necessary from this perspective. In fact, leveraging Java to a modelling language (in the sense that it is defined by an Ecore metamodel), allows language-independent approaches and technologies—as the composition approach and the Reuseware tool defined in this thesis—to be used uniformly on models and code. This means that the composition systems built for programming and other textual languages with U-ISC/Tree in [Hen09] can also be built with our approach. Thus, another contribution of this thesis is that it shows that universal ISC can be applied for complete general-purpose languages on the example of Java (not covered in [Hen09]).

5.1.3 Graph Fragments

Above, we discussed how to obtain graph representations of models. We refer to these representations as *graph fragments*, since they are graphs that are treated as components in invasive composition (where the components are called fragments [Aßm03, Hen09]). Because for invasive composition we need to access and manipulate the structure of graph fragments, we define it in more detail in the following.

Definition: Graph Fragment

A *graph fragment* is a directed, attributed, not necessarily completely connected graph $G = (N_G, E_G)$ where N_G denotes the set of nodes and E_G the set of edges of the graph. G has a distinct spanning forest identified by $T_G \subseteq E_G$. The spanning forest consists of spanning trees, where each has one *root node* $n_G \in R_G$, where $R_G \subseteq N_G$ denotes the set of all root nodes of the spanning forest. Each $n_G \in N_G$ is either target of one edge in the spanning forest $e_G \in T_G$ and $n_G \notin R_G$ or the root node of one spanning tree of the spanning forest $n_G \in R_G$. Furthermore, if $n_G \in N_G$ then A_n denotes the set of attributes of n_G. There are the two functions $i_G : N_G \to E_G$ and $o_G : N_G \to E_G$, where i_G computes the list of all incoming edges of a node and o_G computes the list of all outgoing edges of a node. An edge list $o_G(n_G)$ can be ordered.

A graph fragment is typed by its Ecore metamodel. Thus, an Ecore metamodel is a *type graph* $MM = (C_{MM}, R_{MM})$ where the nodes C_{MM} are all `EClasses` and the edges R_{MM} are all `EReferences`. All attributes A_c for each $c_{MM} \in C_{MM}$ are `EAttributes`. We call this typing *domain typing*.

In addition to the *domain typing*, graph fragments are superimposed with a *variability typing* which is introduced in Section 5.2. Before that, we explore how nodes, edges, and attributes in a graph fragment can be accessed and how their domain types can be discovered.

Access to Graph Fragment Elements in Ecore

To reason over the elements of graph fragments and their types, we require generic access and reflection facilities for that. Ecore provides such facilities and in this section we explain how they map to our graph fragment definition above. Conceptually, other graph formalisms and graph manipulation tools can be mapped in a similar manner to use our approach with other meta-languages and tools.

We use the generic access and reflection facilities to define graph matching and rewriting rules on graph fragments without knowledge of the concrete type graphs (i.e., Ecore metamodels, in the case of Ecore). Figure 5.12a illustrates the facilities we utilise for generic graph access with Ecore. (Corresponding facilities are also standardised for MOF [OMG06a, Chapter 13].)

- `EObject` [SBPM09, Section 2.5.3]. Each node $n_G \in N_G$ is an `EObject` independent of its type defined by its type graph (i.e., its metamodel). This allows us to define matching

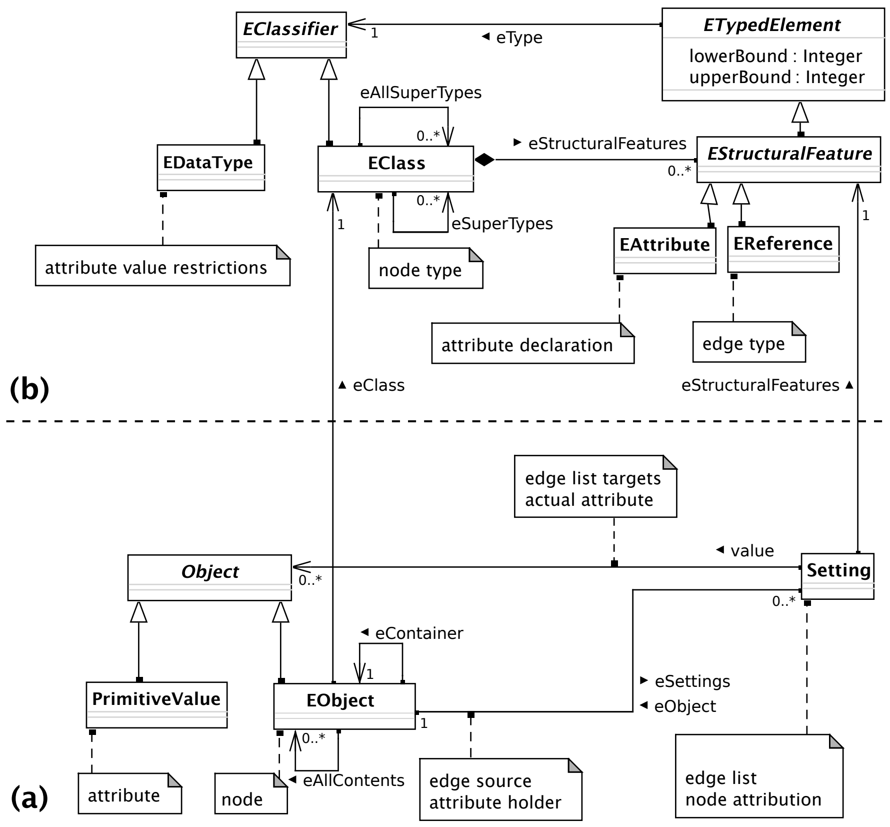


Figure 5.12: Generic graph access (a) and reflective type graph access (b) for Ecore.

and manipulation rules for nodes, without knowing their concrete type. Furthermore, the `eAllContents` edges enable generic access to all contained nodes (i.e., all nodes of the spanning tree below the current node) independent of their position in the tree. The `eContainer` edge allows to navigate to the parent node up the spanning tree.

- `Setting` [SBPM09, Section 16.3]. To perform reflective access to the type graph for edges and attributes, we need a handle for them. This means, we require a node that represents an edge or an attribute. Such a representation is called a `Setting`. A `Setting` either represents a list of edges or a (list) attribute. It has an edge (`eObject`) to the source node n_G of the represented edge list or the attributed node n_G. The `value` edge(s) of a `Setting` point at the target node(s) of the edges and the attribute value(s) respectively. An edge list contains the subset of all outgoing edges of $o(n_G)$ of similar type and is ordered. An attribute can be a list attribute with an ordered list of `PrimitiveValues` instead of a single `PrimitiveValue`.

  `Settings` are not explicitly accessible right away—they implicitly exist as attributes of `EObjects` or edges between `EObjects`. There are two possibilities to obtain a specific `Setting`. First, one can navigate from an `EObject` n_G to `Settings` representing its outgoing edges and attributes $o_G(n_G) \cup A_n$ via the `eSettings` edges. Second, a tool can be used to compute `Settings` for all incoming edges $i_G(n_G)$ of a node n_G.[2]

The following facilities shown in Figure 5.12b are used for reflective access to type graphs (cf. Figure 5.6):

- `EObject.eClass`. For each n_G (i.e., each `EObject`) the `eClass` edge gives access to n_G's type. The node that is reached via `eClass` is the node from the type graph $c_{MM} \in C_{MM}$ that defines the type of n_G. It is an `EClass`, which has the `eSuperTypes` and `eAllSuperTypes` edges. These can be used to navigate the type hierarchy and perform type checks.
- `Setting.eStructuralFeature`. For edges e_G or attributes a_n (i.e., for `Setting`), the `eStructuralFeature` edge gives access to the type. Depending on whether the `Setting` represents an edge or an attribute, `eStructuralFeature` points at an `EReference` or an `EAttribute`.[3] In addition, the `EObject.eContainmentFeature` edge (not shown in the figure) yields the type (the `EReference`) of the containment edge to the current `EObject`.

As typical in object-oriented systems, the typing of a graph fragment is expressed by a total graph morphism, called *instance-of*, denoted as $\rightarrow_{instance-of}$: $G \rightarrow MM$. The $\rightarrow_{instance-of}$ morphism is defined for nodes, edges, and attributes in the following.

For a node $n_G \in N_G$ (an `EObject`), $\rightarrow_{instance-of}$ is defined as follows (following [OMG06a] and [SBPM09]). Let $c_{MM} \in C_{MM}$ be the `EClass` obtained by following the `eClass` edge from n_G. Then $n_G \rightarrow_{instance-of} c_{MM}$. Furthermore, let $C_{super} \subseteq C_{MM}$ be the set of all superclasses of c_{MM} obtained by following all `eAllSuperTypes` edges from c_{MM}. Then, $n_G \rightarrow_{instance-of} c_{super}$ if $c_{super} \in C_{super}$.

For an edge $e_G \in E_G$ (a `Setting`), $\rightarrow_{instance-of}$ is defined as follows (following [OMG06a] and [SBPM09]). Let $r_{MM} \in R_{MM}$ be the `EReference` obtained by following the `eStructural-Feature` edge from e_G. Then $e_G \rightarrow_{instance-of} r_{MM}$.

In addition, an edge has the *containment* property that determines if the edge belongs to a spanning tree. $e_G \rightarrow_{instance-of} r_{MM}$ is a *containment edge* ($e_G \in T_G$) if the `containment` attribute of r_{MM} is set to *true*. $e_G \rightarrow_{instance-of} r_{MM}$ is a *non-containment edge* ($e_G \notin T_G$) if the `containment` attribute of r_{MM} is set to $false$. Thus, all containment edges form the spanning forest (T_G) of the graph fragment. This is of particular importance since the spanning forest can be regarded as a collection of abstract syntax trees (cf. Section 5.1.2) and variability concepts on this part of the graph fragment are therefore close to U-ISC/Tree concepts.

For an attribute $a_N \in A_N$ (a `Setting`) for any n_G, $\rightarrow_{instance-of}$ is defined as follows (following [OMG06a] and [SBPM09]). Let $a_c \in A_c$ (for some $c \in C_{MM}$) be the `EAttribute` obtained by following the `eStructuralFeature` edge from a_N. Then $a_N \rightarrow_{instance-of} a_{MM}$.

[2]In the Ecore implementation in EMF, the `EcoreUtil.CrossReferencer` is provided to compute $i_G(n_G)$ [SBPM09, Section 16.3].

[3]In the Ecore implementation in EMF, the `EObject.eClass` edge is defined as operation `eClass()`, the `EObject.eSettings` edge is defined as operation `eSetting(EStructuralFeature)` and the `Setting.value` edge is defined as operations `get()` and `set()`. These are implementation details. Conceptually these operations can be interpreted as edges as described above which allows us to express graph matchings and manipulation rules over them.

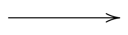
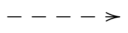
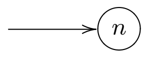
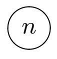
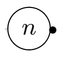
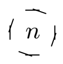
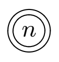
Symbol	Symbol Description	Symbol Semantics
	Solid arrow	Containment edge $e_G \in T_G$
	Dashed arrow	Non-containment edge $e_G \notin T_G$
	Circle with incoming solid arrow	Node inside the graph $n \notin R_G$
	Circle without incoming solid arrow	Root node of one spanning tree $n \in R_G$
	Small circle attached to circle	Variation edge list that is part of the set of outgoing edges of $n \in N_G$
	Dashed circle	Node $n \in N_G$ is a variation node (slot or hook node)
	Double circle	Node $n \in N_G$ is a reference node (anchor or prototype node)

Figure 5.13: Notation for graph fragments.

In Ecore, attributes also have a type (`eType` edge) that always has to be an `EDataType` (cf. Section 5.1). We regard all attributes a_n as *strings* (character sequences). Although in practice an `EDataType` can restrict the set of possible values for an attribute. This kind of primitive typing, however, is not in focus for the variability typing we discuss in the next sections.

This section explained how models are treated as graphs, how these graphs are typed, and how generic access to the elements of these graphs can be performed. This is the basis for invasive composition of graphs as composition technique, which is Feature 2 of our generic CB-MDSD composition system (cf. Figure 5.1).

5.2 Variability Types

Above, we discussed how models are represented as graphs on the example of models defined in an Ecore-based modelling language. Once we have such a graph representation, we can perform model compositions by modifying the graphs. For this, we need to define which parts of a graph can be accessed and modified and how these parts relate to the composition interface of the model.

Example 5.2. Figures 5.14 and 5.15 show the file system and observer UML class models of Example 2.2 in diagrammatic syntax and as graph representations. The graph notation is defined in Figure 5.13. Around the models, a composition interface is indicated that corresponds

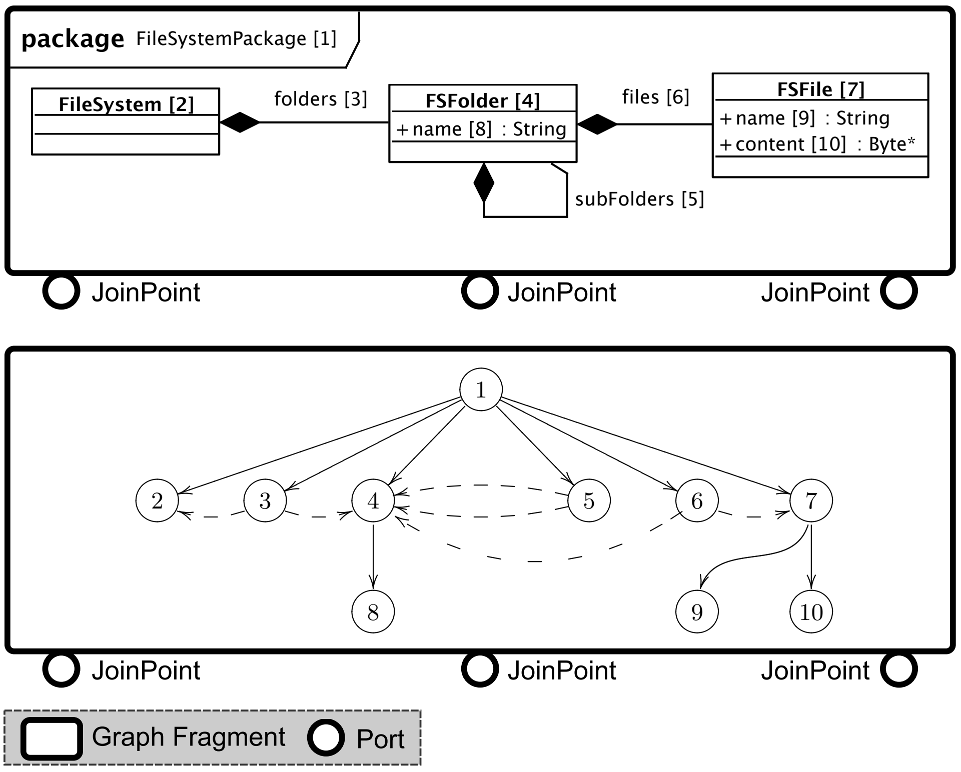


Figure 5.14: A file system modelled in UML as graph fragment.

to the AspectWeaving FraCol used in Example 4.1.2. The question is thus, how the nodes, edges, and attributes of the graphs map to the ports on the composition interface.

To tackle this issue, this section introduces variability concepts for graph fragments that are based on the ones of ISC (cf. Chapter 3) but take the additional requirements that stem from the graph structure of fragments into account. With these concepts, composition interfaces can be established on models.

The strength of ISC is the strong typing of physical composition interfaces, through which selected points of fragments are accessed and modified by the invasive composition technique. Such a point in a fragment is called *variation point* (or *hook* or *slot* which are special types of variation points; cf. Section 3.2). In the grammar-based U-ISC/Tree, type-safe composition was ensured by checking if the result of replacing a variation point with another fragment is still a sentence conforming to the grammar. Effectively, each non-terminal in the grammar defines a type and each node in an abstract-syntax tree conforms to at least one of theses types [Hen09, Section 2.3]. This idea can be transferred to Ecore metamodels and extended to introduce type-safe variability for graphs, which we do in the following.

We distinguish between two typing dimensions. The first is the *domain typing* of a graph that exists when it conforms to a type graph (which is in our case an Ecore metamodel; cf. Section 5.1.3). The second typing dimension is the *variability typing* which we introduce in the following. It is not given by the type graph, but superimposed on the graph by fragment role

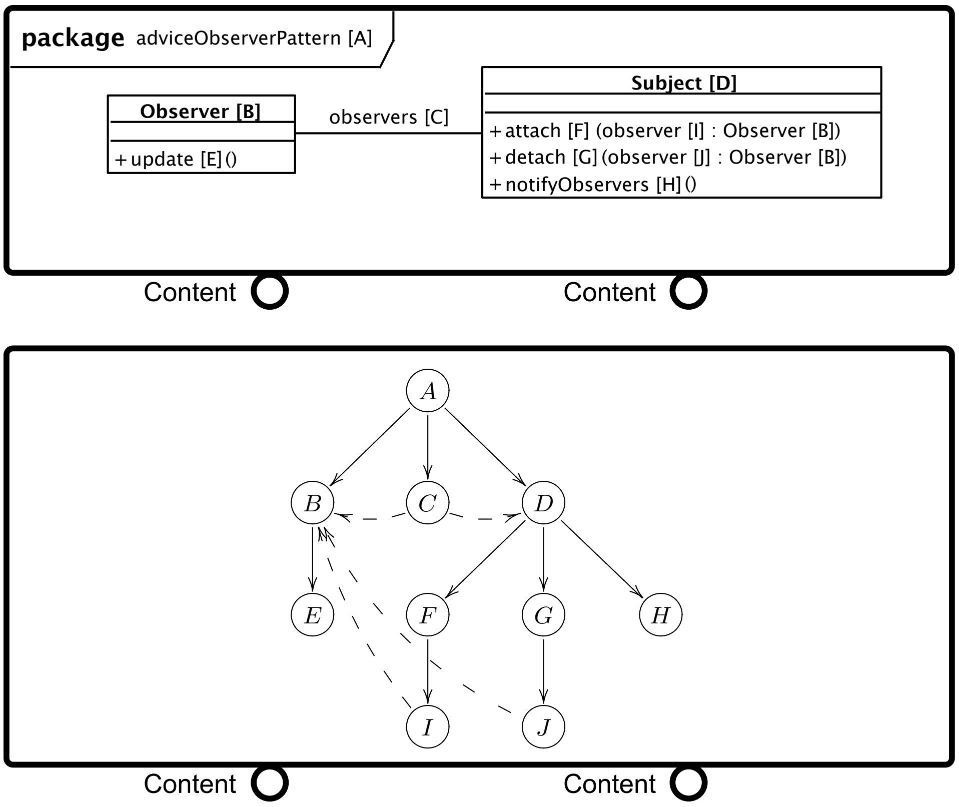

package adviceObserverPattern [A]
Observer [B]
+ update [E] ()
observers [C]
Subject [D]
+ attach [F] (observer [I] : Observer [B])
+ detach [G] (observer [J] : Observer [B])
+ notifyObservers [H] ()
Content
Content
A
B
C
D
E
F
G
H
I
J
Content
Content

Figure 5.15: Observer pattern modelled in UML as graph fragment.

bindings that establish mappings between the type graph (i.e., the metamodel) and a FraCol. Fragment role binding is discussed in Section 5.3. The variability typing, however, builds on and profits from the domain typing.

5.2.1 Variability Typing of Nodes, Edge Lists, and Attributes

As introduced in Chapter 3, the composition technique of U-ISC/Tree is the merging of trees. This is done by replacing one node in a tree with a new subtree. The replaced node and the subtree's root node have to be compatible wrt. the domain typing.

We leverage this composition technique to graph merging. For that, we define that each part of a graph fragment—that is, each node, each edge list, and each attribute—has a *variability type*. A part that can be replaced or extended this way is a *variation point*. A part that may replace or extend a variation point is a *reference point*. All other parts are *hidden*. The term *addressable point* is used to refer to any part that is either a variation or a reference point (i.e., is not hidden).

As indicated, the containment property of an edge (`containment` attribute of `EReference`) is of importance for the variability typing. We remember from Section 5.1 that each node may only have one incoming containment edge while it may have multiple non-containment incoming edges (i.e., each node has exactly one position in the spanning forest of a graph fragment).

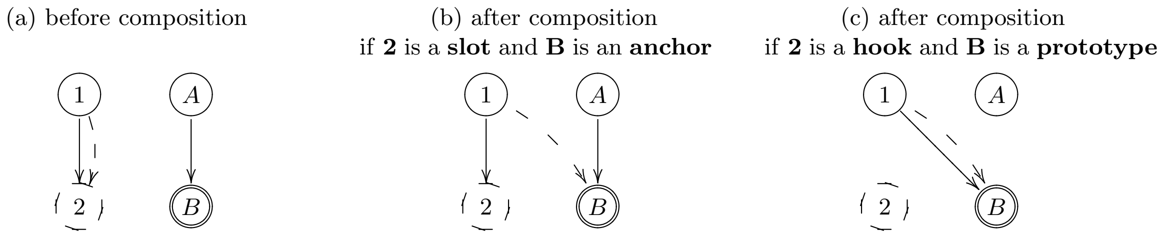

(a) before composition
(b) after composition
if **2** is a **slot** and **B** is an **anchor**
(c) after composition
if **2** is a **hook** and **B** is a **prototype**
1
A
2
B
1
A
2
B
1
A
2
B

Figure 5.16: Containment edge handling in slot/anchor and hook/prototype compositions

Consider a variation point $vn_G \in G$ that has an incoming containment edge $c1_T \in T_G$ and a set of incoming non-containment edges $NC1$ ($NC1 \cap T_G = \emptyset$) and a reference point $rn_G \in G$ that has an incoming containment edge $c2_T \in T_G$ and a set of incoming non-containment edges $NC2$ ($NC2 \cap T_G = \emptyset$). Assume that vn_G is replaced by rn_G. The resulting set of incoming non-containment edges for rn_G will be $NC1 \cup NC2$. However, only one of $c1_T$ or $c2_T$ can point at rn_G after the replacement.

Consequently, we further refine the variability typing. A variation point vn_G that looses its containment edge during replacement is called a *hook*. A reference point rn_G where the containment edge $c2_T$ is replaced by the containment edge $c1_T$ of vn_G is called a *prototype*. A variation point that keeps its containment is called *slot* and a reference point that keeps its containment is called *anchor*.

Figure 5.16 illustrates the slot/anchor and hook/prototype distinctions. In (b), the variation node 2 is treated as as slot and the reference node B as anchor. In (c), 2 is treated as hook and B as prototype. The slot/anchor composition (b) only redirects the non-containment edge 1– –>2 to 1– –>B and the containment edges are not modified. On the contrary, the hook/prototype composition (c) redirects the containment edge 1—>2 to 1—>B as well. However, since B can only have one incoming containment edge, the containment edge A—>B is destroyed in this process

The terminology was chosen for the following reasons:

- The term *hook* was adopted from traditional ISC where each extension point in a fragment is a hook and the composition corresponds to a physical extension of the fragment. In our case, a hook is also a place where new nodes are added to a graph, since the containment hierarchy (i.e., the spanning forest) is modified at these places.
- If the same *prototype* node is bound to several hooks, the node and its contained nodes need to be copied each time since one copy is required for each containment edge. Thus, the original node is a prototype for all copies—hence the term prototype.
- In contrast to binding a hook, binding a *slot* only adds new non-containment edges to the graph, but no new nodes. This can be interpreted as parameterising the graph, in contrast to extending it. Traditionally, the term slot is often used for places of parametrisation (e.g., in Beta fragments [MMPN93] or in templates).
- We chose the term *anchor* as counterpart to slot since binding an anchor node means that it is "anchored" by new edges but the node itself is not moved from its position in the containment hierarchy (i.e., its spanning tree).

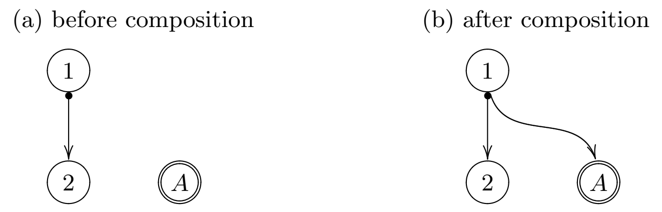


Figure 5.17: Graph composition by adding an edge to an edge list

(a) before composition

value hook (`getVALUE_HOOK`, 3, 12)
value prototype (`MyName`)

(b) after composition

`getMyName`

Figure 5.18: Attribute composition by modifying strings.

If nodes are variability typed, we talk about *variation nodes* and *reference nodes*. If the composition behaviour with respect to containment edges is important, we talk about *hook nodes*, *prototype nodes*, *slot nodes*, and *anchor nodes*.

Variability typing can also be applied to edge lists. More precisely, to each list of edges $\forall e_G \in o_G(n_G) : e_G \rightarrow_{instance-of} r_{MM}$—that is, all edges of similar type with the same source node. Such an ordered list of edges can be extended with additional edges of similar type. This can be used to create additional edges and thus perform composition by creating additional edges instead of replacing existing nodes.

Let $MM = (C_{MM}, R_{MM})$ be the metamodel to which G conforms, where $R_{MM} = CR_{MM} \cup NCR_{MM}$ is the set of `EReferences` with $CR_{MM} \cap NCR_{MM} = \emptyset$. CR_{MM} is the set of `EReferences` with `containment` $= true$ and NCR_{MM} is the set of `EReferences` with `containment` $= false$. To realise composition by creating additional edges of type r_{MM}, we support the definition of a variability type for an edge list consisting of all edges $e_G \in o_G(n_G)$ with $e_G \rightarrow_{instance-of} r_{MM}$. A new edge $e_G \notin E_G$ may be created to connect a reference node $rn_G \in N_G$ without actually replacing a variation node, if $e_G \rightarrow_{instance-of} r_{MM}$. If rn_G is a prototype, $r_{MM} \in CR_{MM}$ has to hold. If rn_G is a slot, $r_{MM} \in NCR_{MM}$ has to hold.

Composition by edge creation is illustrated in Figure 5.17. There, no variation nodes (as, e.g., in Figure 5.16) exist in the graph before composition (a). Only the edges list, which contains the 1—>2 edge, is variability typed as hook (visualised by the small circle attached to node 1). Therefore, new edges can be added to this list which is done in (b), where the new edge 1—>A is added.

We may also type edge lists as reference points. If $e_G \in T_G$ for all edges e_G in the edge list, it is typed as prototype. If $e_G \notin T_G$ for all edges e_G in the edge list, it is typed as anchor. All target nodes of the edges in the edge list are then automatically variability typed correspondingly. Similar to nodes, any edge list can be *hidden*.

If edge lists are variability typed, we talk about *variation edge lists* and *reference edge lists*. If the composition behaviour with respect to containment edges is important, we talk about *hook edge lists*, *prototype edge lists*, *slot edge lists*, and *anchor edge lists*.

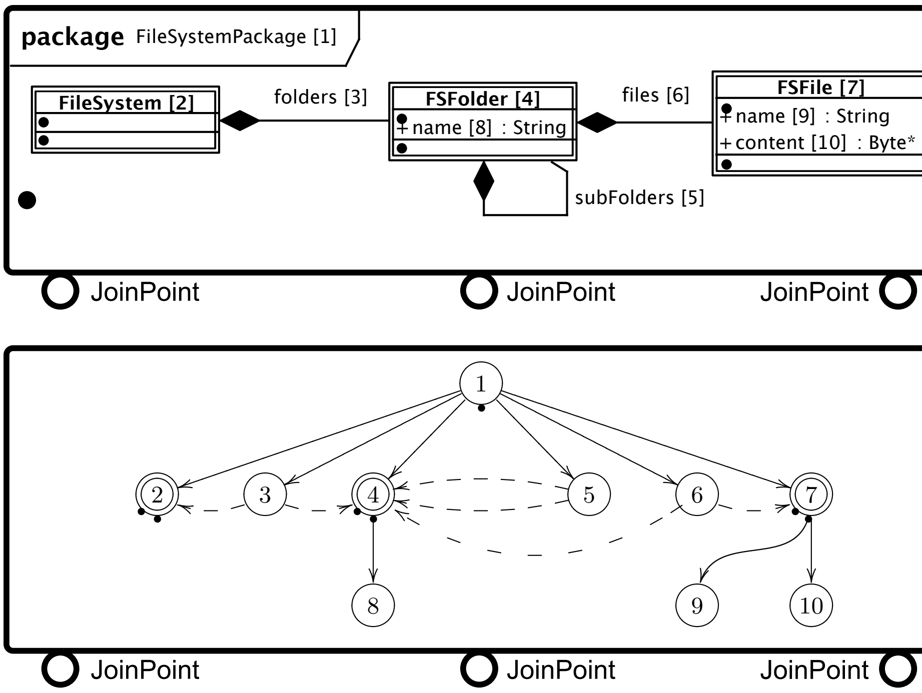


Figure 5.19: A file system modelled in UML with variability typing.

As mentioned, we consider each attribute to be a string (i.e., a character sequence). The concepts of hook and prototype are further refined for attributes to *value hook* and *value prototype*. In case of a value hook, a certain part of the attribute—that is, a sub-character sequence—is defined as replaceable. In case of a value prototype, a certain information is extracted from the attribute as character sequence. In addition to a normal hook, a value hook has a *begin index*, an *end index*, and a *list index* which define the range of characters that are replaced. The list index is needed if we deal with a list attribute. In addition to a normal prototype, a value prototype holds the actual value extracted from an attribute. This value does not need to correspond to the attribute but can also be computed based on the attribute.

Attribute composition by string manipulation is illustrated in Figure 5.18. In the example, the attribute with value hook is `getVALUE_HOOK` and the begin index and end index are 3 and 12 respectively, which means that only the `VALUE_HOOK` sub-string is to be replaced. In the example, this sub-string is replaced with the value prototype with the string value `MyName`.

- The terms *value hook* and *value prototype* were chosen since value hooks and value prototypes are used for extending a graph with additional attribute values (similar to how hooks and prototypes are used to extend a graph with new nodes). The term *value* stems from the fact that primitive string values are treated here instead of nodes and edges.

A distinction between containment and non-containment is not required in the case of attribute values (i.e, there is no value slot or value anchor). This is, because the plain character sequence data is never cross-referenced but always contained. Consequently there is also no equivalent to the `containment` flag found in `EAttribute` (cf. Figure 5.6).

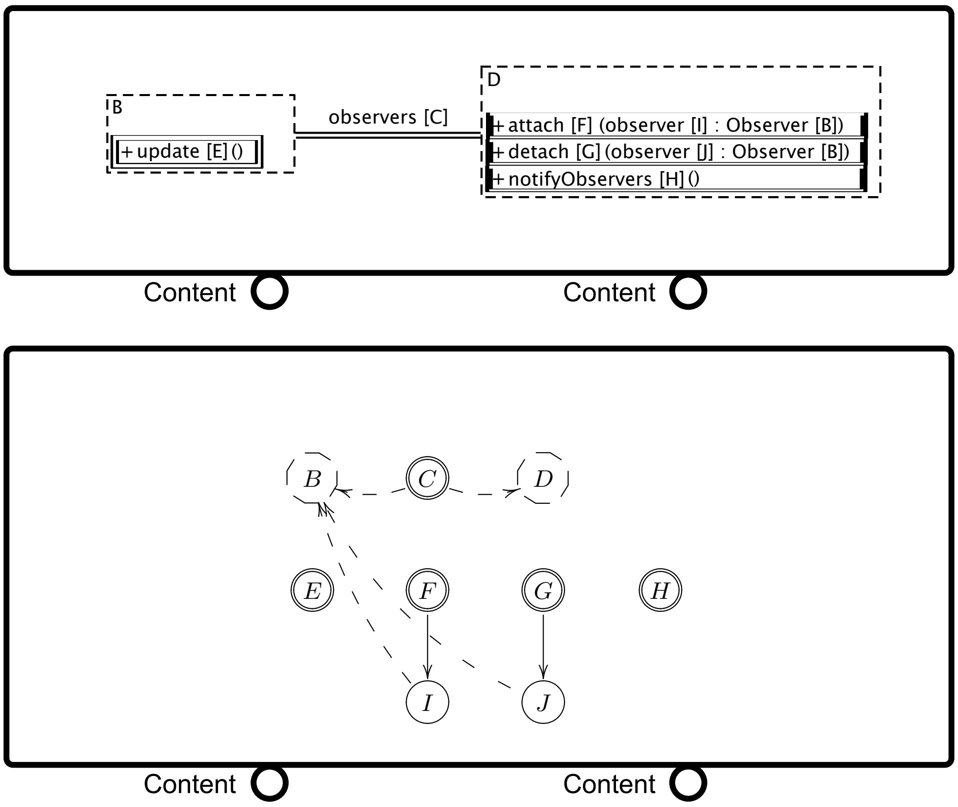


Figure 5.20: Observer pattern modelled in UML with variability typing.

Example 5.2.1. To compose the file system and observer UML models (Figures 5.14 and 5.15) we would like to achieve a similar extensibility for UML as for Java in COMPOST (cf. Example 3.1). For that, we add variability typing to the graph fragment as shown in Figure 5.19. The notation for graphs with variability types is summarised in Figure 5.13. The look of variability typed graph parts (see column *Symbol Description* in Figure 5.13) is also overlaid on the diagrammatic models. We use this notation throughout the rest of this thesis.

In Figure 5.19, both the list of `Properties` and the list of `Operations` (cf. UML metamodel in Figure 5.8) of each `Class` are typed as hook edge lists (●). The `Classes` themselves are typed as anchor nodes (◎: 2, 4, and 7). This allows on the one hand the extension of classes with new `Properties` and `Operations` and on the other hand the access to classes during composition, for instance to insert them as type of a newly added `Property`. Another hook (●) is defined for the list of `Associations` and `Classes` of a `Package`.

The UML model shown in Figure 5.15 models the Observer pattern in a similar fashion as it was done in Java in Example 3.1. To turn this model into an advice that could be woven into the file system model, we add variability typing in Figure 5.20. Here we observe, that the `Model` node A shown in Figure 5.15 does not reappear in Figure 5.20, because it can never be reused. The reason for this is that it is not a prototype node and cannot be reached through containment edges from another prototype node.

From the advice, we want to reuse the Operations and Associations. Thus, they are variability typed as prototypes (◎: C, E, F, G, and H). Furthermore, the types of the Operations, Parameters, and Associations should be exchangeable. For example, Observer should be exchanged for FileSystem as type for the observer parameter of the attach() operation. Thus, all Classes are slots (B and D) that may be replaced by anchors during composition.

5.2.2 Type compatibility

In the following, we illustrate the interplay between the domain and the variability typing dimensions by clarifying which parts of a graph may be replaced or extended by other parts without violating type constraints.

- A prototype node p_G may be added to a hook edge list with the type c_{MM} iff $p_G \rightarrow_{instance-of} c_{MM}$.
- Let C_{EL} be the set of types of the edges in the edge list $i(h_G)$, where h_G is the hook node to replace. A prototype node p_G may then replace h_G iff $\forall c_{MM} \in C_{EL} : p_G \rightarrow_{instance-of} c_{MM}$.
- An anchor node a_G may be added to a slot edge list with the type c_{MM} iff $a_G \rightarrow_{instance-of} c_{MM}$.
- Let C_{EL} be the set of types of the edges in the edge list $i(s_G)$, where s_G is the slot node to replace. An anchor node a_G may then replace s_G iff $\forall c_{MM} \in C_{EL} : a_G \rightarrow_{instance-of} c_{MM}$.
- Each value prototype may extend each value hook.

The variability typing concepts introduced above are defined as metamodel in Figure 5.21a. It contains the variability types Prototype, Hook, Anchor, Slot, ValuePrototype, and Value-Hook with the corresponding superclasses ReferencePoint and VariationPoint. Each such variability typed part of a graph fragment is an AddressablePoint. All other parts are automatically hidden.

To assign a variability type to a part of a graph fragment one assigns it in the context of a node (i.e., an EObject) for which the AddressablePoint has the varTypedEObject reference. To assign a variability type to a node, only varTypedEObject is set to point at the EObject that is the node. To assign a variability type to an edge list, varTypedEObject has to point to the node that is the source of all edges in the list and varTypedEStructuralFeature has to point at the type of the edges (i.e., an EReference). To assign a variability type to an attribute, varTypedEObject has to point at the attributed node, while varTypedEStructuralFeature points at the attribute's declaration (i.e., an EAttribute).

In summary, we define six variability types for parts of graph fragments—four for nodes and edge lists (*hook*, *prototype*, *slot*, and *anchor*) and two for attributes (*value hook* and *value prototype*). *Hook*, *slot*, and *value hook* identify variation points for replacement or extension during composition. *Prototype*, *anchor*, and *value prototype* identify reference points that replace or extend during composition. *Slots* and *anchors* only influence non-containment edges. *Hooks* and *prototypes* also influence containment edges. *Value hooks* and *value prototypes* influence attributes.

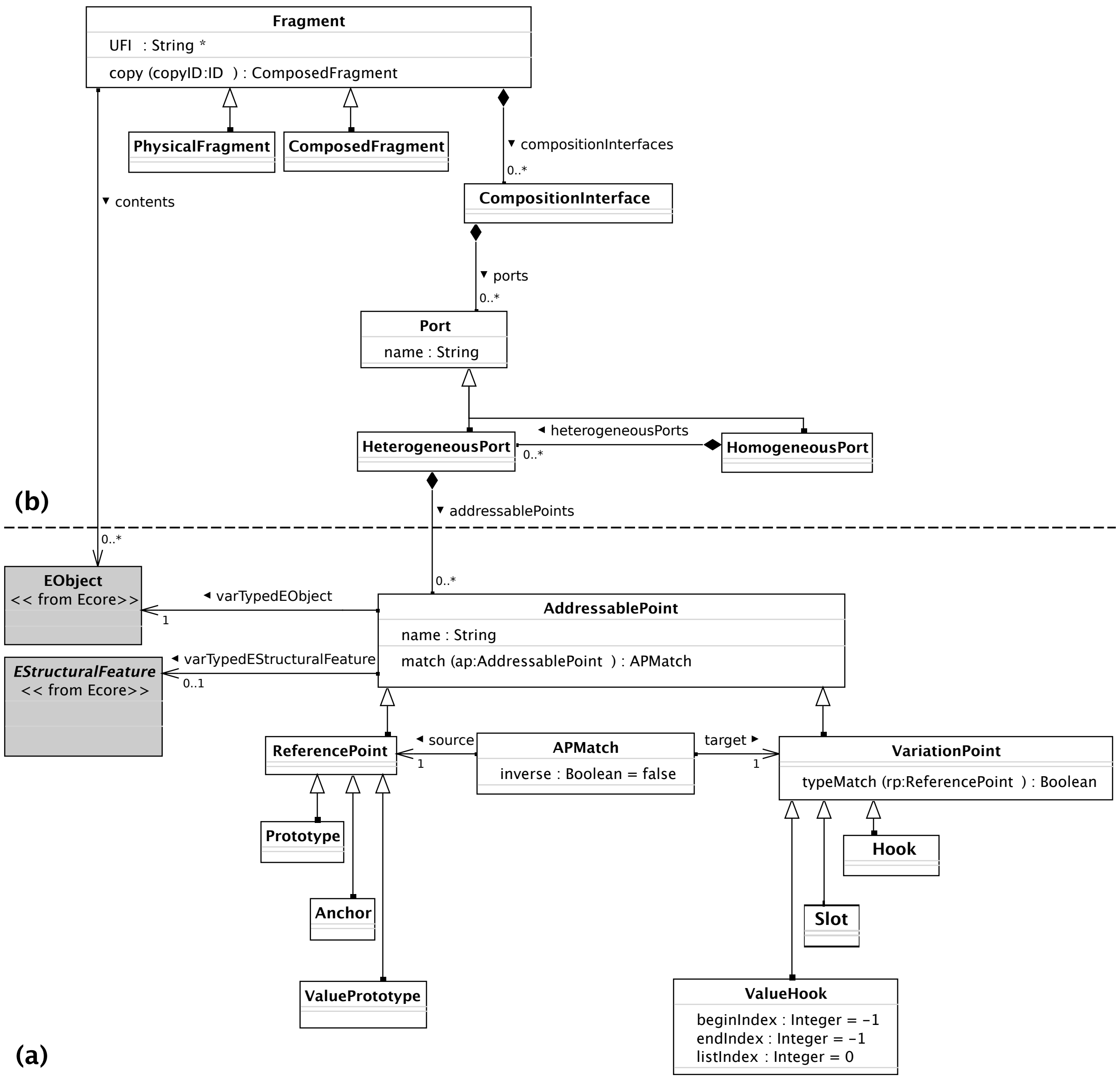


Figure 5.21: Variability typing (a) and grouping (b) concepts for graph fragments.

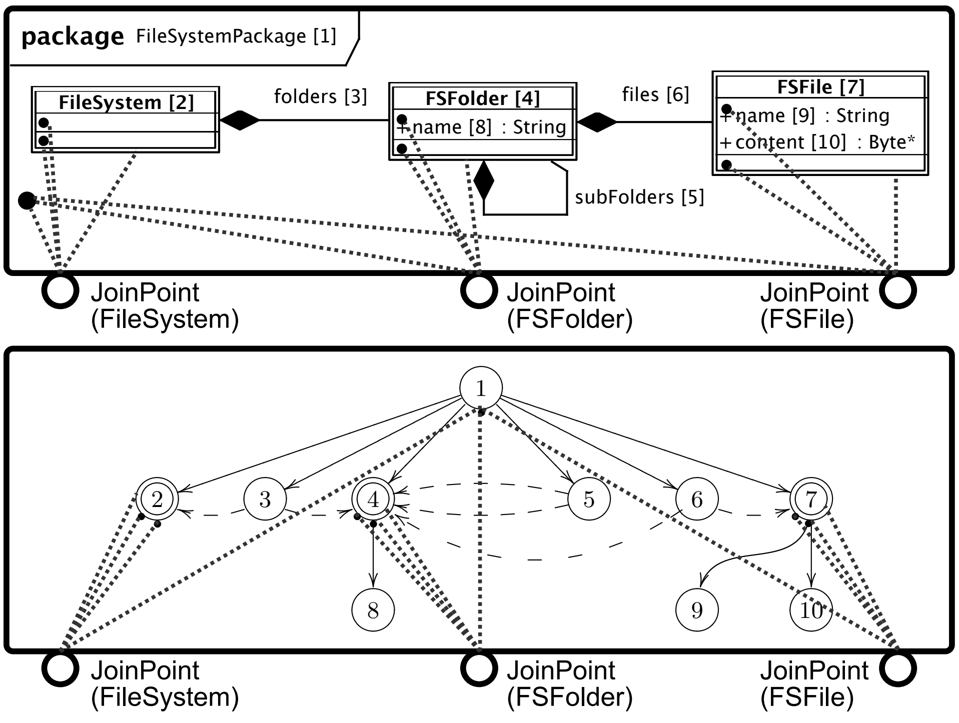


Figure 5.22: Heterogeneous ports on the file system UML model.

5.2.3 Heterogeneous Grouping of Addressable Points

The variability typing discussed above only concerns single locally connected parts (i.e., single nodes, single attributes, or nodes that are targets of edges in one list) of a graph fragment. A port on a fragment's interface however, may relate to several parts inside a fragment as already indicated in the examples in Chapter 4.

When a fragment is reused, not only single nodes are reused, but a set of nodes which can be distributed over the graph. In ISC tree composition, all nodes of one subtree were always considered as one unit of reuse. In graph composition, however, there is a need to address multiple subtrees due to the structure of graph fragments. If we consider a single subtree extracted from one of the spanning trees of the spanning forest T_G of a graph G, this subtree may very well contain a node connected with another node not contained in the subtree by an edge $e \notin E_T$. In that case, this other node (or a subtree in which it is contained) needs also to be considered in the same composition.

Consequently, grouping of addressable points for variability purpose is necessary. Such a grouping of several places that cross-cut an artefact is referred to as heterogeneous cross-cut [ABR06]. Therefore, we refer to this kind of node grouping as heterogeneous grouping and call a concrete instance of such a grouping a *heterogeneous port*. Figure 5.21b illustrates that a heterogeneous port combines a number of addressable points. The heterogeneous port concept has similarities to the nested hook concept of ISC [Aßm03].

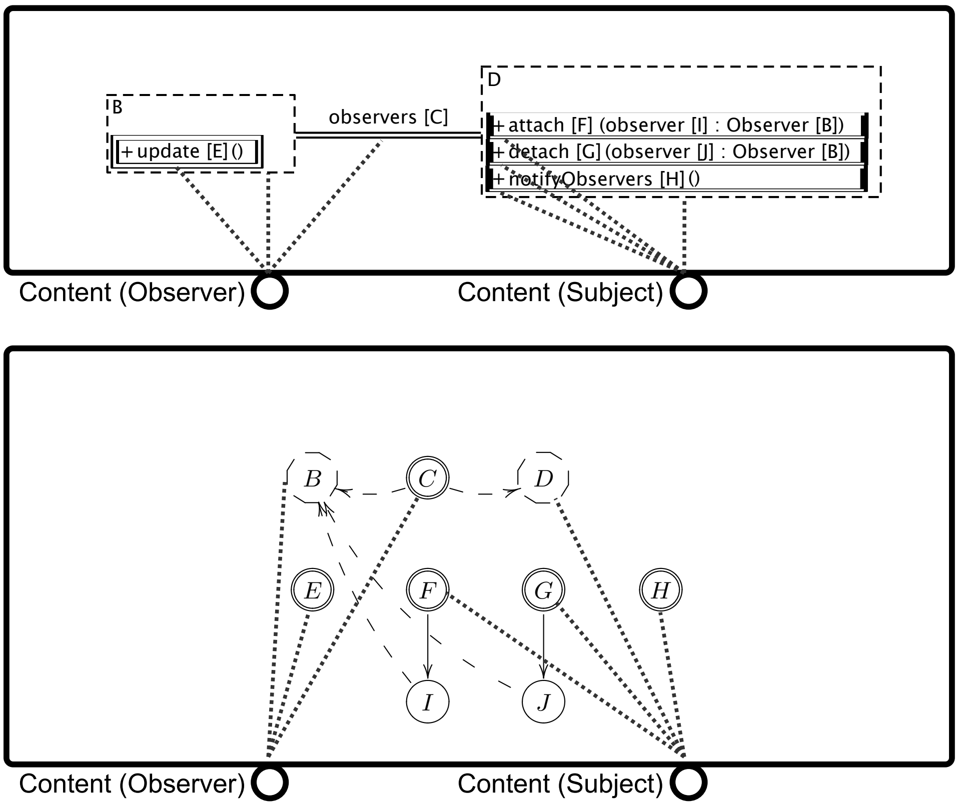


Figure 5.23: Heterogeneous ports on the observer UML model.

A heterogeneous port is a physical interface for accessing a fragment through addressable points grouped by the port. Thus, a heterogeneous port can be directly mapped to a port type defined in a FraCol. With this, we make the connection between FraCols and composition interfaces which allows fragments to play fragment roles. In Section 4.1.2 we introduced the distinction between static and dynamic port types. This influences ports as follows:

- For each static port type, only one port can exist in the composition interface of a fragment and the name of that port has to be similar to the port type's name.
- Multiple ports of the same dynamic port type can exist in the composition interface of a fragment and the name of each of these ports can be freely chosen.

There are almost no restrictions to which addressable points can be grouped into a heterogeneous port, which enables maximal flexibility for defining heterogeneous cross-cuts. The domain typing does not enforce any restrictions to this grouping. The only restriction enforced by the variability typing is that hooks and prototypes may not be grouped together, because during composition nodes are transferred from one fragment into another. This can only be done acyclically to avoid an infinite recursion. Combining prototypes and hooks in one port would lead to such a case where the fragment on the one hand offers nodes but on the other hand needs to receive nodes. This is discussed in more detail in Section 6.3.1.

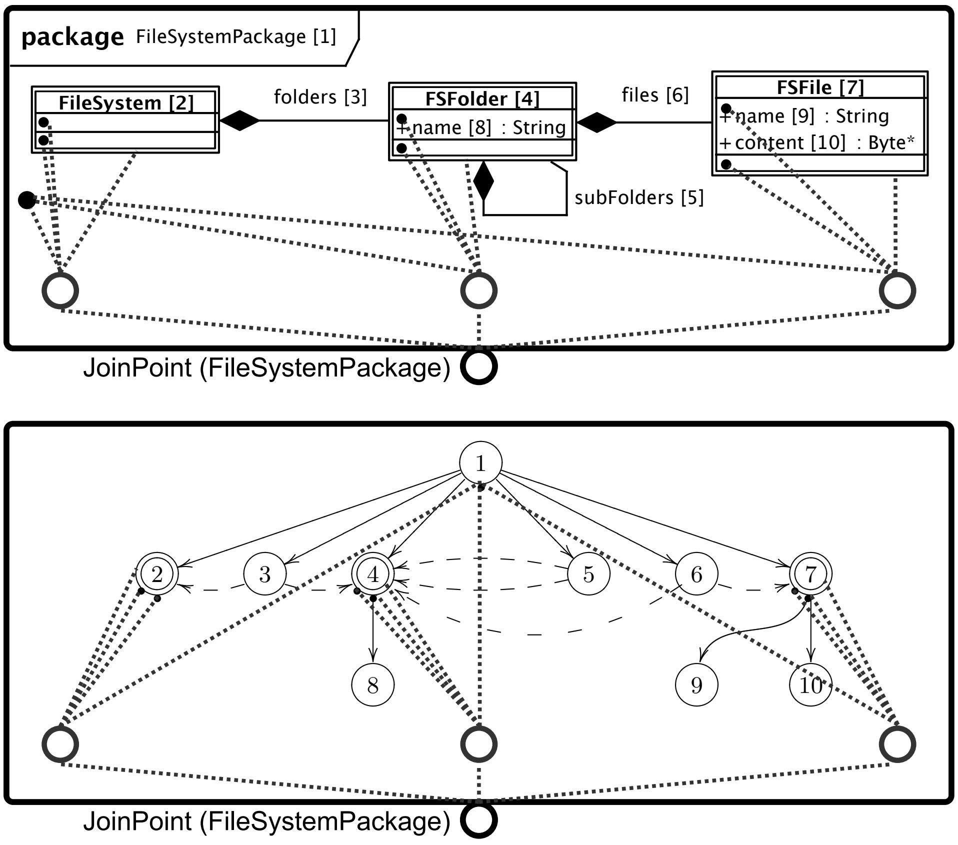


Figure 5.24: Homogeneous ports on the file system UML model.

Example 5.2.3. Consider the example in Figure 5.22 that shows how the addressable points in the file system fragment are mapped to ports and with that to the FraCol ClassWeaving that was defined in Figure 4.10. Here, each class (Node 2, 4, or 7) is an anchor itself and has two hooks in outgoing edge lists. These three addressable points—one anchor and two hooks—are related to one class and therefore form a group of addressable points that should be treated together for the purpose of class extension. Thus, grouping them into one heterogeneous port makes sense. Furthermore, it is sensible to add the hook for associations (on Node 1) to such a heterogeneous port to allow for associations that are connected to a class to be added together with attributes and operations.

Corresponding heterogeneous ports can be formed on the observer advice fragment as shown in Figure 5.23. Here, all operations—which are prototypes—of one class are grouped together with that class—which is a slot (Observer or Subject in the example). An association that is related to that class is added to such a group as well. In Figure 5.23, we define two heterogeneous ports: One representing the observer part of the pattern (B, C, and E) and one representing the subject part (D, F, G, and H).

5.2.4 Homogeneous Grouping of Ports

The replication of fragments is a central concept in fragment composition. In the tree composition of U-ISC/Tree, copying information was the only option when it should reappear at different positions, since cross-referencing was not possible. Although graphs allow for cross-referencing information in general, a concrete metamodel might not allow a certain type of cross-reference.

While there can be different reasons for this situation, the probably most common is that the metamodel was designed with a certain dominant modularity concept that does not foresee the required flexibility for modularising and cross-referencing a certain fragment of information. A prominent example is aspect-orientation [KLM$^+$97] as an extension on object-orientation: The object-oriented decomposition did not allow for a required separation for certain cross-cutting information. Aspect-orientation adds this dimension by extracting code that would reappear at several positions in traditional object-oriented design, into a single fragment called aspect.

When we want to add support for different FraCols to the same language similar situations can appear. Thus, we allow further grouping of heterogeneous into *homogeneous ports* (Figure 5.21b). The heterogeneous ports grouped in a *homogeneous port* need to have similar signatures. That is, they have to group the same types of addressable points. Such a homogeneous grouping is referred to as quantification in aspect-orientation [FF00]. Heterogeneous ports that reside inside a homogeneous port are not visible on a fragments interface.

A homogeneous port has the effect that during a composition the graph parts that are injected through the port are replicated for each of the grouped heterogeneous ports. This is further explained in Section 6.3. From the outside, heterogeneous and homogeneous ports that are visible on the interface are not distinguished. Similar to heterogeneous ports, homogeneous ports can have a static or a dynamic port type.

Example 5.2.4. Consider the fragment of Figure 6.14 which models an ID feature as class extension consisting of an `id` property as well as a setter and a getter operation for that property. A possible composition scenario is to add this ID feature to all classes in one package—for instance all classes in the file system package of Figure 5.19. In that case, the three heterogeneous ports described above (for extending the FileSystem (2), FSFolder (4), and FSFile (7) classes) can be further grouped into one homogeneous port as shown in Figure 5.24. All three heterogeneous ports have the same signature (the class as slot, list of properties as hook, list of operations as hook, and the package's association list as hook).

When the ID fragment is composed into the file system fragment, the composition algorithm would copy the ID fragment three times—one time for each heterogeneous port that is hidden behind the homogeneous port FileSystemPackage. This is further discussed in Section 6.3.3.

5.2.5 Fragment, Port, and Addressable Point Identification

Identification is an important concept in reuse. This is true on different levels of granularity. Unique identification can be used to find reuseable artefacts in repositories, but also to address points on a composition interface. In the case of graph fragments, we have three levels of identification, which are reflected in attributes of metaclasses in Figure 5.21:

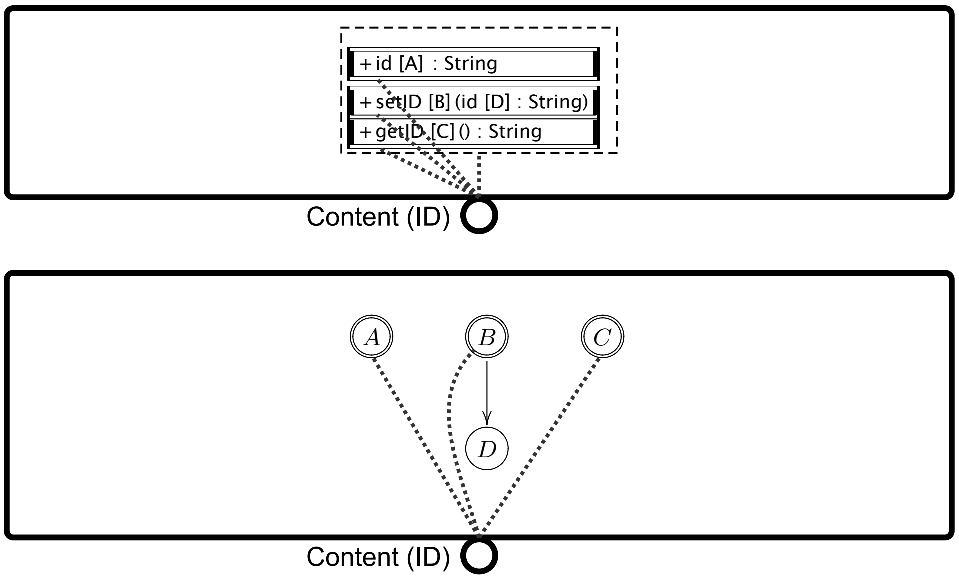


Figure 5.25: ID feature UML class with variability typing and grouping.

1. *Unique Fragment Identifier (UFI)*: A fragment needs a unique identifier if it should be globally reuseable. In such a situation, the fragment does not live in a closed environment (e.g., the runtime memory of one application) but migrates between different environments and physical representations. If fragments are shared among people or organisations, it is important that this identifier is preserved over time and space, since the contexts in which the fragment is reused need to be able to retrieve the fragment when needed. We call such an identifier *Unique Fragment Identifier (UFI)*. An UFI is a list of strings that is unique to each fragment (cf. Figure 5.21a).
2. *Port name*: Ports are the places on a fragment's composition interface that can be addressed for composition. Thus, each port needs an unique name in the context of the fragment such that it can be distinguished from other ports. This is true for homogeneous and heterogeneous ports alike. A port name does not need to be globally unique, since it can be globally identified in combination with the UFI of its containing fragment.
3. *Addressable point name*: The need for naming addressable points arises from the internals of a composition system rather than for the purpose of external accessibility (since addressable points are hidden in ports). As we will learn in Section 6.3.1, names are required in situations, where similar nodes (i.e., nodes of similar type) are grouped but require individual identification.

Example 5.2.5. Consider the fragment shown in Figure 5.22. The fragment needs an UFI via which we can retrieve it for extension (1st level of identification).

If we want to extend it with applications of the observer pattern (cf. Example 5.2.3), each heterogeneous port that represents a class for extension needs to be addressed individually (2nd level of identification). For example, the *FileSystem* is extended with subject functionality, the

FSFolder with observer functionality. These heterogeneous ports have the dynamic port type `JoinPoint` defined in the FraCol ClassWeaving (cf. Figure 4.10). As explained in Section 5.2.3, names have to be chosen for ports with a dynamic port type. This can be done by deriving names from inside the fragment. In this case, the names of the classes themselves are good candidates for the names of heterogeneous ports. In contrast, ports of static type can be addressed via the name of the port type.

If all classes in the file system package are extended with the ID feature (cf. Example 5.2.4), it is sufficient to address the homogeneous port representing all classes of the package. In this case, the name of the package is a good candidate for the name of the homogeneous port (2nd level of identification).

In the examples above, all hooks grouped in a heterogeneous port have different domain types. The hook in the list of properties is of type `uml::Property`; the hook in the list of operations is of type `uml::Operations` (cf. UML metamodel in Figure 5.8). If both lists had the same domain type, we would need another way to distinguish them to perform a deterministic assignment between hooks and prototypes without mixing up properties and operations. In that case, a naming of the hooks would be required in the context of the heterogeneous port that groups the two hooks (e.g., one hook can be named *properties* and one *operations*; 3rd level of identification).

We assume that a *repository* exists in which fragments can be placed and retrieved via their UFI.[4] Fragments that are represented by instances of the metamodel from Figure 5.21 can be placed in and retrieved from the repository. These instances link to the underlying model via the `contents` edges that reference all root `EObjects` R_G of the underlying graph fragment G. How instances of Figure 5.21 are obtained from a model, is discussed in the next section.

This section introduced variability typing for graphs that is used to define physical composition interfaces for fragments through which they can be accessed for composition. This covers Feature 3 of our generic CB-MDSD composition system (cf. Figure 5.1).

5.3 Component Model Configuration with REX$_{\text{CM}}$

Although we explained that a composition interface of a graph fragment can be defined by superimposing variability typing on it, we have not yet introduced a method to specify variability typing. Such a method configures the component model of our CB-MDSD composition system (Feature 4) and with that defines rules for fragment role binding (cf. Section 4.1). In this section, we introduce a *component model configuration* language for that purpose. Specifications written in that language extend the typing of a DSML by adding variability typing to the domain typing, which enables reuse of models defined in that DSML. Hence, we name our component model configuration language *Reuse EXtension language for Component Model configuration* (REX$_{\text{CM}}$). As indicated in the beginning of this chapter, composition system developers can use REX$_{\text{CM}}$ to configure the component model of the generic CB-MDSD composition system to integrate new DSMLs as component languages, which in turn allows developers to develop model components in that DSML (this was illustrated in Figure 5.2).

[4]Such a repository is part of REUSEWARE (Chapter 7).

The concepts of REX$_{CM}$ are closely aligned with the variability typing concepts presented in the previous section. In general, the language has to allow for specifying the conditions under which a node, an edge list, or an attribute in a graph fragment has a certain variability type. Such specifications can be performed on the base of the metamodel (i.e., the type graph) of the language. Additionally, if the rules for variability typing are put in relation to fragment roles defined in FraCols, they act as *fragment role binding rules*. Fragment role binding rules hence define how a certain fragment role is realised by a concrete DSML by stating which parts of the metamodel map to the role.

Example 5.3. As an example consider Figure 5.26 that shows a part of the UML metamodel (top) and the FraCol ClassWeaving from Example 4.1 (bottom). Between the port types and elements of the metamodel, fragment role binding rules are defined, which are annotated with variability types. These rules can be applied to a UML model, which is an instance of the UML metamodel. Variability types are computed for every part of the UML model that instantiates a part of the metamodel for which a rule exists. Furthermore, all variability typed parts of the model are mapped to a port with the corresponding port type. For example, the rules can be applied on the file system and observer UML models to derive the variability types shown in Figures 5.22 and 5.23 respectively.

To express binding rules, we require an expression language which we do not want to define from scratch. Therefore, we integrate the Object Constraint Language (OCL) [OMG06b], which is a standardised language to specify expressions for Ecore metamodels.

In the following, we first give a short description of OCL (Section 5.3.1) since OCL is integrated in REX$_{CM}$ and used in the example later on. Afterwards, we describe the concepts of REX$_{CM}$ and how it is used (Section 5.3.2). Finally, we define the interpretation of REX$_{CM}$ (Section 5.3.3).

5.3.1 The Object Constraint Language (OCL)

OCL [OMG06b] is an OMG standard to express constraints and formulate queries on UML or EMOF/Ecore models. OCL is a language that needs a host language with a UML-like class concept to operate on, which is Ecore in our case. OCL can be used to formulate constraints and queries in the context of a class (in our case an `EClass`). The constraint or query can then be evaluated on a model element that is an instance of that `EClass`.

For formulating a constraint or query, one can navigate over the model starting from the context element, which can be referred to by the `self` keyword. OCL comes with a standard library with collection types and primitive types to handle references and attributes with multiplicities greater than 1 and to perform operations on primitive values such as strings.

Some OCL implementations allow to extend the standard library if desired. The OCL interpreter we are using supports this and we extend the primitive type string—which has only few operations available in the standard library—with the Java string API[5]. (The details of the implementation are presented in Chapter 7.) This is helpful, because we often need to perform operations on strings to derive identifiers.

[5]http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html

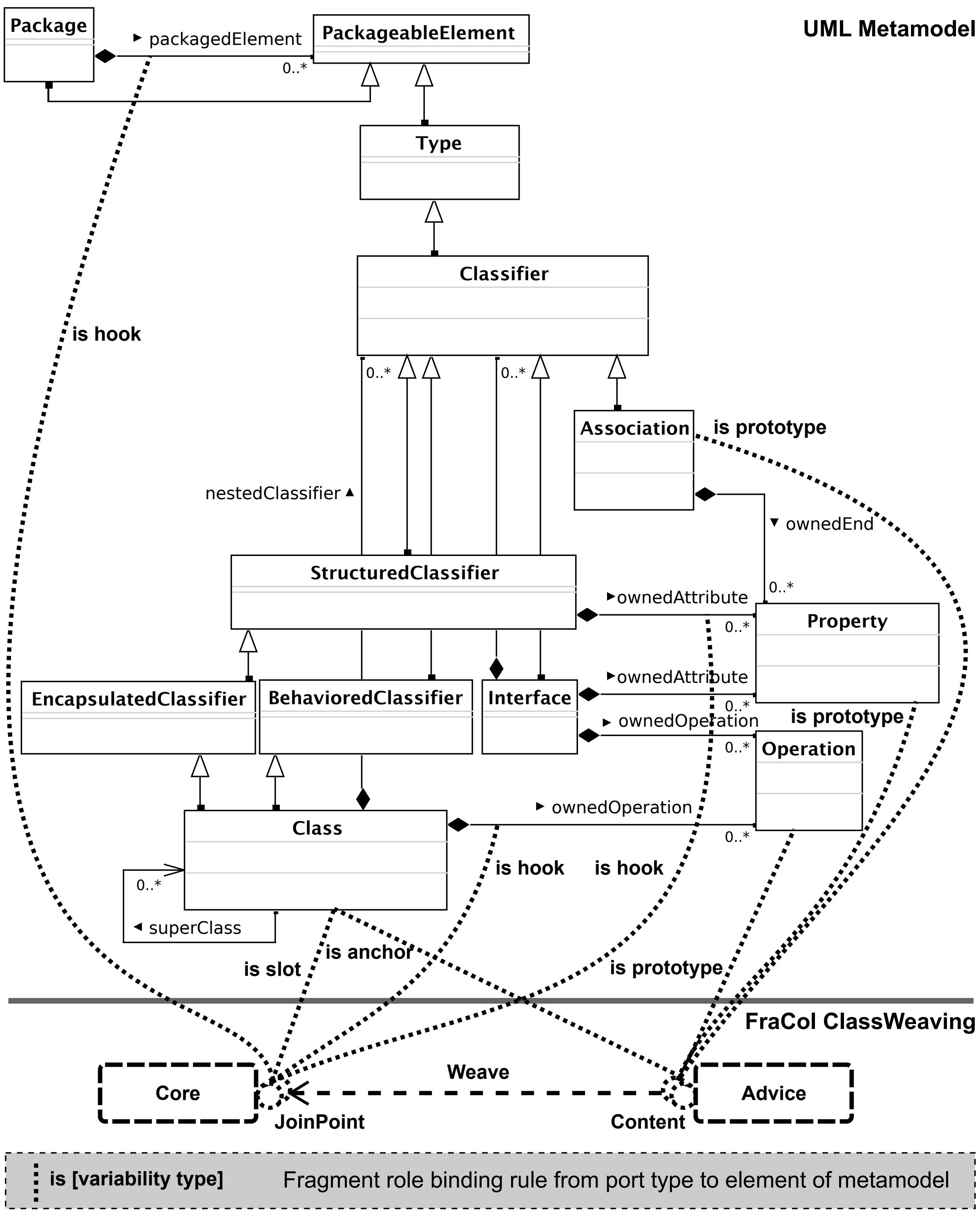


Figure 5.26: Binding with variability type assignment between the FraCol ClassWeaving (cf. Figure 4.10) and the UML metamodel (cf. Figure 5.8).

```
self.packagedElement->select(c | c.oclIsTypeOf(uml::Class))
```

Listing 5.2: OCL query for finding all classes contained in a UML package.

```
self.members->select(c | c.oclIsTypeOf(java::members::Field))
```

Listing 5.3: OCL query for finding all fields contained in a Java class.

A query over Ecore models is formulated in OCL by starting at the `self` element and then navigating over the model using names of `EStructuralFeatures` and the "`.`" operator as known from object-oriented languages such as Java. Operations defined on the metaclasses or on standard library types (as the mentioned string operations) can also be called. Operations on collections are called via "`->`" instead of "`.`".

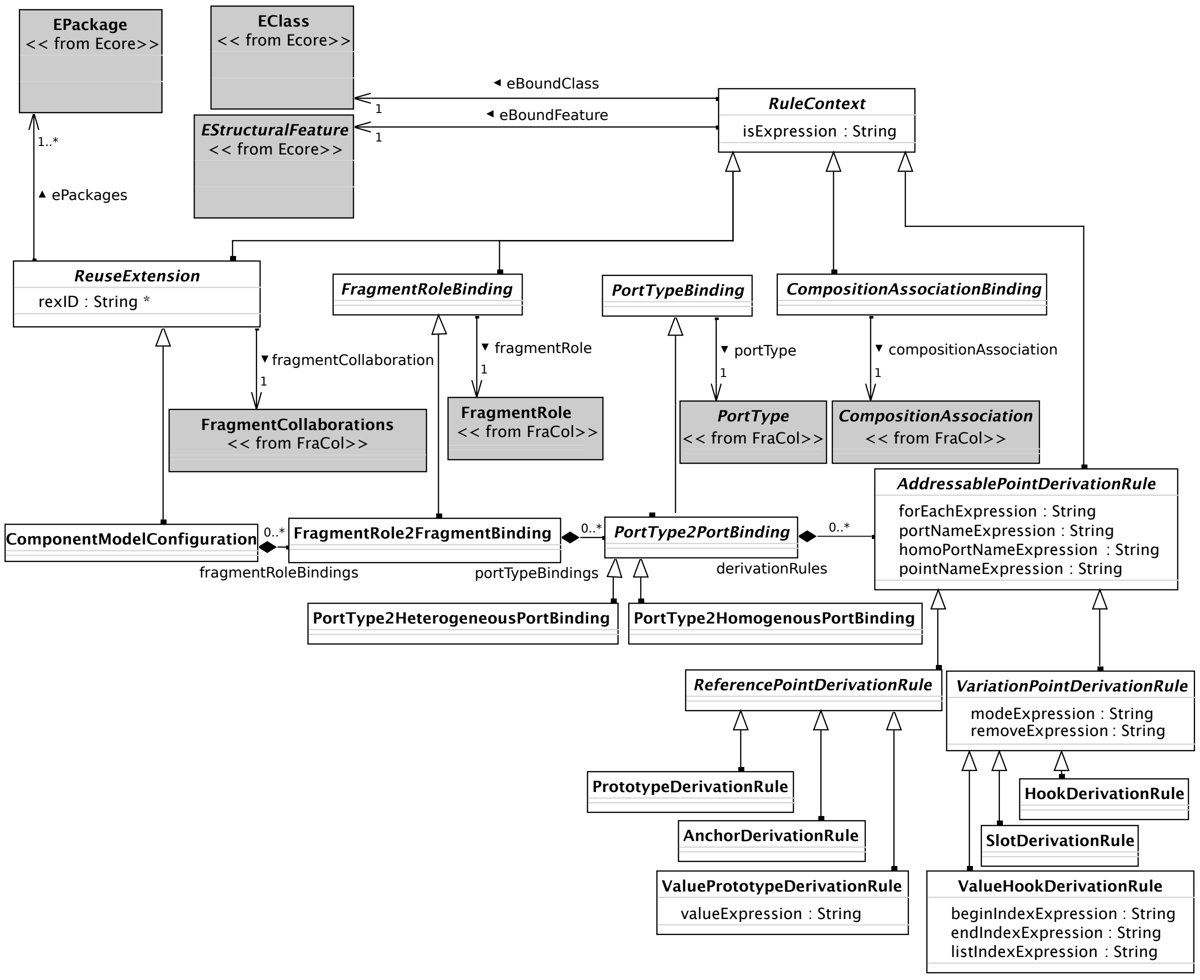
Example 5.3.1. Listing 5.2 shows an example OCL query that is formulated over the UML metamodel in the context of the class `Package` (cf. Figure 5.8). It queries for all elements contained in the package that are of type `Class`. For this, we navigate along the `packagedElement` reference which returns a collection of elements. On this collection we call the `select` operation, which is one of the OCL standard library operations for collections that selects all elements of a collection that meet a certain condition. In this case, we select all elements that are of type `Class` which can be checked by the `oclIsTypeOf()` operation which is also provided by the standard library for model elements of any type. A similar query could be defined, for instance, over the Java metamodel in the context of `ConcreteClassifier` that selects all `Fields` from the list of members (Listing 5.3).

There are more details about OCL we did not mention here. Although we embed the complete OCL language into REX$_{CM}$, we only use a small part of the language in our examples. For further explanations of OCL concepts, please refer to the standard document [OMG06b].

5.3.2 REX$_{CM}$ Language Concepts

In the following, we define the REX$_{CM}$ language. It is a textual language that embeds OCL for rule definition. The language's metamodel is depicted in Figure 5.27. Since REX$_{CM}$ is used to map metamodels to FraCols, it contains relationships to the Ecore metamodel (cf. Figure 5.6) and our FraCol metamodel (cf. Figure 4.2). The concepts found in the metamodel are reflected in the textual syntax of the language which we explain in the following on a schematic REX$_{CM}$ specification shown in Listing 5.4. (The complete grammar REX$_{CM}$ can be found in Section 7.2.)

Lines 1–4 specify the general properties of the component model configuration. In Line 1, an ID is assigned to the component model under which it is placed in the repository (cf. Section 5.2.5). The FraCol that is *implemented* by this component model configuration is identified by its ID in Line 2. In order to refer to `EClasses` in the rest of the specification, a set of `EPackages` is identified in Line 3 by the `EPackages`' `nsURIs` (cf. Section 5.1). The `EClasses` contained in the first `EPackage` can be addressed by their name only in the following. A class from any of the `EPackages` can be addressed using `EPackages`' `nsPrefixes` in the `::`



EPackage << from Ecore>>
EClass << from Ecore>>
EStructuralFeature << from Ecore>>
eBoundClass
eBoundFeature
1
1
1..*
ePackages
RuleContext
isExpression : String
ReuseExtension
rexID : String *
FragmentRoleBinding
PortTypeBinding
CompositionAssociationBinding
fragmentCollaboration
1
fragmentRole
1
portType
1
compositionAssociation
1
FragmentCollaborations << from FraCol>>
FragmentRole << from FraCol>>
PortType << from FraCol>>
CompositionAssociation << from FraCol>>
AddressablePointDerivationRule
forEachExpression : String
portNameExpression : String
homoPortNameExpression : String
pointNameExpression : String
ComponentModelConfiguration
0..*
FragmentRole2FragmentBinding
0..*
PortType2PortBinding
0..*
fragmentRoleBindings
portTypeBindings
derivationRules
PortType2HeterogeneousPortBinding
PortType2HomogenousPortBinding
ReferencePointDerivationRule
VariationPointDerivationRule
modeExpression : String
removeExpression : String
PrototypeDerivationRule
HookDerivationRule
AnchorDerivationRule
SlotDerivationRule
ValuePrototypeDerivationRule
valueExpression : String
ValueHookDerivationRule
beginIndexExpression : String
endIndexExpression : String
listIndexExpression : String

Figure 5.27: Component model configuration Concepts.

prefix notation (cf. Section 5.1). Finally, Line 4 defines the `EClass` (i.e., the context) a root node has to instantiate to make the REX_{CM} applicable to the corresponding graph fragment—optionally followed by a condition formulated in OCL that the root node has to fulfil.

Lines 5–21 show a blueprint specification of a *fragment role binding*. In Line 5, the fragment role to be bound, which has to be defined in the FraCol identified in Line 2, is referred by its name. The name is optionally followed by a condition that has to be fulfilled by the root node when the role binding should be applied. The role binding contains one port type binding for each port type defined in the role. The port types are referred to by their names (Line 6).

Each port type binding contains arbitrary many *addressable point derivation rules* to specify variability types for nodes, edge lists, or attributes. For each type of addressable point, a corresponding derivation rule type exists (cf. Figure 5.21). Which rule type is defined concretely is determined by the *variabilityType* (Line 7) which is either hook, prototype, slot, anchor, value hook, or value prototype. *eClass* is the name of an `EClass` defined in one of the `EPackages` named above. This `EClass` has to be instantiated by a node to apply the rule to it. Optionally,

```
 1 componentmodel ID
 2 implements     fracolID
 3 epackages      <epackage>
 4 rootclass      rootelementEClass if $OCLExpression$ {
 5   fragment role fragmentRoleName if $OCLExpression$ {
 6     port type portName {
 7       eClass.feature is variabilityType if $OCLExpression$ {
 8         foreach $OCLExpression$
 9 
10         homo port = $OCLExpression$
11         port      = $OCLExpression$
12         point     = $OCLExpression$
13 
14         value     = $OCLExpression$  /* only for value prototypes */
15 
16         begin idx = $OCLExpression$  /* only for value hooks */
17         end   idx = $OCLExpression$  /* only for value hooks */
18         list  idx = $OCLExpression$  /* only for value hooks */
19       }
20     }
21   }
22 }
```

Listing 5.4: Schematic REX$_{CM}$ specification.

a condition can be given in OCL that the node has to fulfil in addition. If a *feature* is identified, the rule concerns the variability typing of the edges lists or attributes that are instances of the *feature*. If no *feature* is identified, the rule concerns the variability typing of the nodes that are instances of the *eClass*.

If a node (or an edge list or an attribute) is variability typed, additional information to identify the node (or the edge list or the attribute) might have to be derived. The *homogeneous port name expression* (Line 10), *port name expression* (Line 11), and *point name expression* (Line 12), identify the heterogeneous port, the homogeneous port, and the addressable point itself respectively. Any of the expressions is only required is the corresponding name is required (cf. discussion in Section 5.2.5). The expressions are formulated in OCL and have to compute a string value. Furthermore, the optional *forEachExpression* (Line 8) can be used to create multiple addressable points for a graph part and assign these to different heterogeneous or homogeneous ports. This allows one graph part to be used in multiple places of the composition interface. This is done by querying the graph for a set of nodes, which can be reached from the context of the rule, and interpret the other expressions in the context of each of these nodes.

As discussed in Section 5.1.3, a value prototype needs to extract information as a string. Thus, a *value expression* has to be given for a *value prototype derivation rule* which is expected to return a string (Line 14). For value hooks we might require the definition of a range inside an attribute that shall be manipulated (cf. Section 5.1.3). This can be specified using the *begin index expression* (Line 16) and *end index expression* (Line 17) that are expected to return positive integer values that identify a position in the attribute that is to be manipulated. These expressions are not required if the complete attribute should be modified. If the rule concerns a list attribute, the *list index expression* (Line 18) can be used to specify the position in the list that should be modified.

OCL, which is used for all expressions discussed above, gives the developer who uses REX_{CM} the possibility to seamlessly integrate *implicit* and *declared* variability into the component model [Aßm03, Section 4.1.2] (cf. Chapter 3). Rules that do not require special structures or formats of attribute settings specify implicit parts. For example, a rule that says that each `uml::Class` is a prototype specifies implicit variation, since each `uml::Class` automatically is a prototype and there is no way for a developer to avoid that. Rules that enforce specific patterns the developer is aware of (e.g., naming conventions as used in COMPOST; cf. Example 3.1) specify declared parts of a composition interface. For example, a rule that says that an attribute setting with the value `HOOK` is a value hook gives the developer the possibility to explicitly declare value hooks. Sometimes however, the border between *implicit* and *declared* parts is fluent and also depends on the knowledge of a user of the composition system. For instance, when the user knows that each `uml::Class` is a prototype, one can argue that he or she explicitly declares a prototype each time he or she defines an `uml::Class`.

To illustrate the flexibility for component model configuration that is achieved with REX_{CM}, we give four different examples of REX_{CM} definitions based on the examples discussed so far.

Example 5.3.2. (1/4) The first example realises the component model configuration that was sketched in Figure 5.26 which binds the FraCol ClassWeaving to UML. The configuration is shown Listing 5.5. If we apply it to the file system UML model, we obtain the variability typing and the composition interface shown in Figure 5.22. If we apply the same specification to the observer UML model, we obtain the variability typing and the composition interface shown in Figure 5.23.

In detail, Listing 5.5 defines the following. The UML metamodel is identified by its `nsURI` as defined in the Eclipse UML2 project [Ecl10d] (Line 2). Since the specification only applies for UML models, it is stated in Line 3 that the root element of a model has to instantiate the EClass `uml::Model`, which is the case for all UML models. Since the specification is valid for all UML models, no further condition (via `if` keyword) is defined.

For each of the two fragment roles defined in the FraCol ClassWeaving (cf. Listing 4.4), which are Core and Advice, a fragment role binding is defined. We would like to define that a model contains advice classes (i.e., classes that define extension like Observer and Subject) if its name starts with *advice...*, which is the case for the observer model in Figure 5.23. All other models contain core classes (i.e., classes to be extended). We define the distinction by augmenting each fragment role binding with a condition under which the corresponding role is played by a fragment. The condition that a fragment is an advice is `self.name.startsWith('advice')` (Line 6) and the condition that it is a core is `not self.name.startsWith('advice')` (Line 24).

Each fragment role binding contains four rules. The first two rules of the Core binding (Lines 8–13) state that the list of operations (`ownedOperation`) and the list of attributes (`ownedAttribute`) of a `uml::Class` are hooks. `port = $self.name$` states that for each class, both addressable points are assigned to the same port that is named after the class. This is necessary, because `JoinPoint` is a dynamic port type. For static port types, the port name expression is not required since only one port with the name of the port type is created to which all addressable points are automatically assigned (cf. Section 4.1.2). As a third extension point for each core class (Lines 14–17), the list of packaged elements

```
 1 componentmodel org.reuseware.example.class_weaving.uml
 2 implements     org.reuseware.example.class_weaving
 3 epackages      <http://www.eclipse.org/uml2/3.0.0/UML>
 4 rootclass      uml::Model {
 5
 6     fragment role Core if $not self.name.startsWith('advice')$ {
 7         port type JoinPoint {
 8             uml::Class.ownedOperation is hook {
 9                 port = $self.name$
10             }
11             uml::Class.ownedAttribute is hook {
12                 port = $self.name$
13             }
14             uml::Package.packagedElement is hook if $self.oclIsTypeOf(uml::Package)$ {
15                 foreach $self.packagedElement->select(c | c.oclIsTypeOf(uml::Class))$
16                     port = $self.name$
17             }
18             uml::Class is anchor {
19                 port = $self.name$
20             }
21         }
22     }
23
24     fragment role Advice if $self.name.startsWith('advice')$ {
25         port type Content {
26             uml::Operation is prototype if $self.owner.oclIsTypeOf(uml::Class)$ {
27                 port = $self.owner.oclAsType(uml::Class).name$
28             }
29             uml::Property is prototype if $self.owner.oclIsTypeOf(uml::Class)$ {
30                 port = $self.owner.oclAsType(uml::Class).name$
31             }
32             uml::Class is slot {
33                 port = $self.name$
34             }
35             uml::Association is prototype {
36                 port = $self.ownedEnd->at(1).type.name$
37             }
38         }
39     }
40 }
```

Listing 5.5: REX_{CM} specification for an aspect system for UML.

of the containing package is used as a place where additional associations can be added. For this, we define a rule for `uml::Package.packagedElements`, but use a `foreach` to assign this point to each port that represents a class in the package. The OCL expression `packagedElement->select(c | c.oclIsTypeOf(uml::Class))` yields the list of classes and the expression `port = $self.name$` is then evaluated in the context of each class which assigns the point to the port that is named after the corresponding class. The fourth rule of the `JoinPoint` port type binding (Lines 18–20) adds each class as anchor to the port that is named after the class. This is required to bind the class itself as type of new associations or operations in the composition.

The `Content` port type binding contains four rules to derive prototypes and slots, as counterparts to the hooks and anchors of `JoinPoint` ports (Lines 26–37). In a similar fashion as for core classes, addressable points are grouped into ports named after the classes. Each `uml::Operation` or `uml::Property` prototype is assigned to the port that represents

```
componentmodel org.reuseware.example.class_weaving.java
implements     org.reuseware.example.class_weaving
epackages      <http://www.emftext.org/java>
rootclass      java::containers::CompilationUnit {

    fragment role Core if $not namespaces->asSequence()->last().startsWith('advice')$ {
        port type JoinPoint {
            java::classifiers::Class.members is hook {
                port = $self.name$
            }
            java::classifiers::Class is anchor {
                port = $self.name$
            }
        }
    }

    fragment role Advice if $namespaces->asSequence()->last().startsWith('advice')$ {
        port type Content {
            java::classifiers::Class.members is prototype {
                port = $self.name$
            }
            java::classifiers::Class is slot {
                port = $self.name$
            }
        }
    }
}
```

Listing 5.6: REX$_{\mathrm{CM}}$ specification for an aspect system for Java.

the class owning it. A `uml::Association`, which is owned by a package, is assigned to the class that is at the first end of the association. The name of that class is extracted via `self.ownedEnd->at(1).type.name`.

Example 5.3.2. (2/4) A FraCol can be implemented by multiple component model configurations to integrate multiple component languages that support the same FraCol into a composition system. We illustrate this for the FraCol ClassWeaving in Listing 5.6 that binds this FraCol to the Java metamodel. The same fragment roles are bound as in the case of UML, but different rules are defined that are specific to the Java metamodel (cf. Figure 5.9).

The Java versions of the file system and observer models, which were introduced in the COMPOST example in Section 3.1, are shown in Figures 5.28 and 5.29 with the variability types and composition interfaces obtained by applying Listing 5.6. This illustrates how complexity has been moved from composition operator implementations in Java (as required by COMPOST and the U-ISC/Tree implementation; cf. Chapter 3). to the component model configurations in REX$_{\mathrm{CM}}$ and how this exploits the graph structure of fragments. In the COMPOST example the relationships between Observer and Subject were handled by the *weave* operator (cf. Listing 3.3) which took both as argument (one as `adviceFragment` and one as `collaborator`). The operator used names to bind Java classes and not the classes themselves since it was working on the abstract syntax tree where names had not been resolved. In our case, the relationships between Observer and Subject are captured (cf. Figure 5.29) through the cross-references to D (slot for the Java class that will become the subject) and to B (slot for the Java class that will become the observer).

```
public class FileSystem2 {          public class FSFolder4 {              public class FSFile7 {
    protected FSFolder4[] folders3;     protected String name8;               protected String name9;
}                                       protected FSFolder4[] subFolders5;    protected Byte[] content10;
                                        protected FSFile7[] files6;       }
                                    }
```

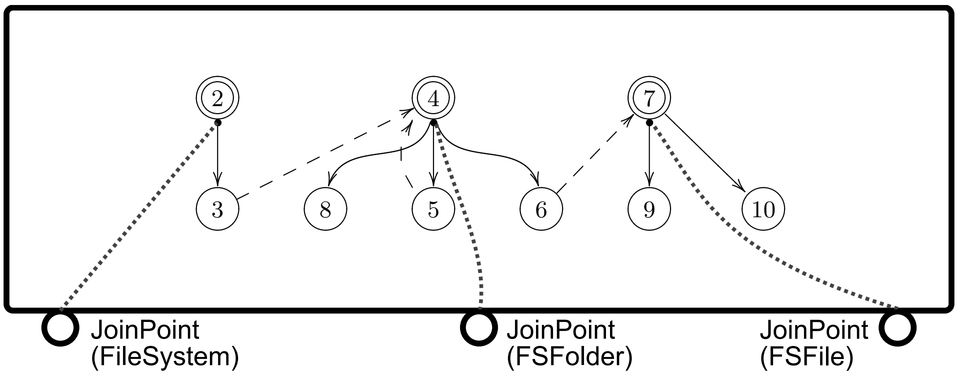


Figure 5.28: A file system defined in Java with variability typing and grouping.

```
public class ObserverB {                              public class SubjectD {
    protected SubjectD[] subjectsC1;                     protected ObserverC[] observersC2;

    public void updateE() {                              public void attachF(ObserverB observerI) {
        //...                                                //...
    }                                                    }
}                                                        public void detachG(ObserverB observerJ) {
                                                             //...
                                                         }
                                                         public void notifyObserversH() {
                                                             //...
                                                         }
                                                      }
```

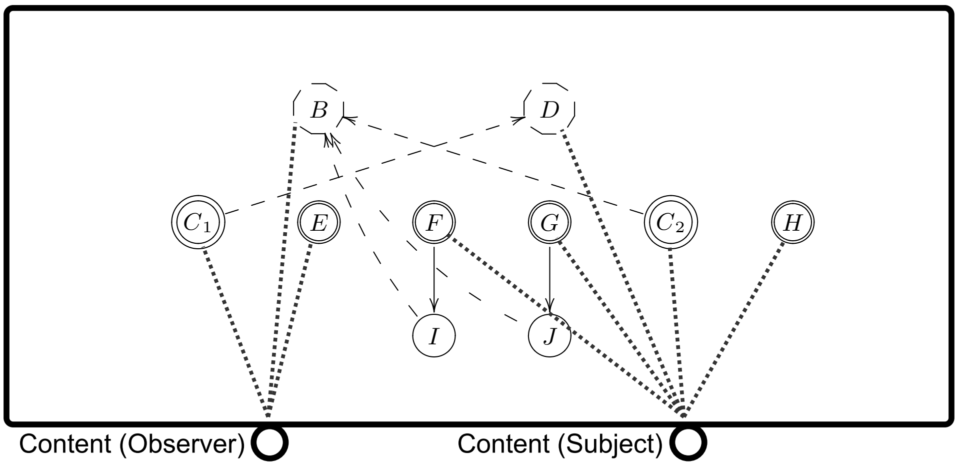


Figure 5.29: Observer pattern defined in Java with variability typing and grouping.

```
componentmodel org.reuseware.example.class_weaving2.uml
epackages      <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass      uml::Model {

    fragment role Core if $not self.name.startsWith('advice')$ {
        port type JoinPoint {
            uml::Class.ownedOperation is hook {
                homo port = $self._package.name$
                port = $self.name$
            }
            uml::Class.ownedAttribute is hook {
                homo port = $self._package.name$
                port = $self.name$
            }
            uml::Package.packagedElement is hook if $self.oclIsTypeOf(uml::Package)$ {
                foreach $self.packagedElement->select(c | c.oclIsTypeOf(uml::Class))$
                    homo port = $self.oclAsType(uml::Class)._package.name$
                    port = $self.name$
            }
            uml::Class is anchor {
                homo port = $self._package.name$
                port = $self.name$
            }
        }
    }

    fragment role Advice if $name.startsWith('advice')$ {
        port type Content {
            uml::Operation is prototype if $owner.oclIsTypeOf(uml::Class)$ {
                port = $self.owner.oclAsType(uml::Class).name$
            }
            uml::Property is prototype if $owner.oclIsTypeOf(uml::Class)$ {
                port = $self.owner.oclAsType(uml::Class).name$
            }
            uml::Class is slot {
                port = $self.name$
            }
            uml::Association is prototype {
                port = $self.ownedEnd->at(1).type.name$
            }
        }
    }
}
```

Listing 5.7: REX$_{\mathrm{CM}}$ specification for another aspect system for UML.

```
componentmodel org.reuseware.lib.systems.participation.cm.usecase_uml
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
    fragment role Participant {
        homo port type Name {
            uml::Actor.name is value hook {
                port  = $'ActoreName'$
                point = $'name'$
            }
            uml::Association.name is value hook {
                port  = $'AssociationName'$
                point = $'name'$
                begin idx = $'0'$
                end idx   = $'0'$
            }
        }
        port type Contrib {
            uml::Actor is prototype {}
            uml::Association is prototype {}
            uml::UseCase is slot {}
        }
    }
    fragment role Collaboration {
        port type Rec {
            uml::Package.packagedElement is hook if $packagedElement->exists(e|e.oclIsKindOf(UseCase))$ {}
            uml::Package.ownedComment is hook if $packagedElement->exists(e|e.oclIsKindOf(UseCase))$ {}
            uml::UseCase is anchor {}
        }
    }
}
```

Listing 5.8: $\mathrm{REX}_{\mathrm{CM}}$ specification that binds UML to the FraCol Participation (cf. Listing 4.2.)

Example 5.3.2. (3/4) The third example, shown in Listing 5.7, illustrates that a FraCol can be bound to a metamodel in different ways on the binding of ClassWeaving to UML. The binding rules are similar to Listing 5.5. Only this time, complete packages of core classes are extended instead of single core classes. Except for the ID, the header (Lines 1–3) and the Advice fragment role binding (Lines 27–42) are similar. Applied to the file system model, this component model gives the variability typing and composition interface shown in Figure 5.24.

The four rules in the `JoinPoint` port type binding (Lines 6–24) are also quite similar to Listing 5.5. The difference is that each rule contains a `homo port = ...` expression that derives the name of a homogeneous port. In this example, the name is derived from the name of the package that contains a `uml::Class`. The individual heterogeneous ports, which are identified by the `port = ...` expression for each class individually, are then grouped in, and hidden behind, homogeneous ports according to their packages. This was illustrated for the file system model in Figure 5.24. Although heterogeneous ports are hidden behind homogeneous ports, they still need to be individually named to construct the correct heterogeneous groupings of `ownedOperation`, `ownedAttribute`, `ownedPackage`, and `uml::Class` that are required by the matching algorithm discussed in Section 6.3.1.

```
componentmodel org.reuseware.lib.systems.participation.cm.java
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
    fragment role Participant {
        homo port type Name {
            java::commons::NamedElement.name is value hook if $name.contains('NAME_HOOK')${
                point = $'name'$
                begin idx = $name.indexOf('NAME_HOOK')$
                end   idx = $name.indexOf('NAME_HOOK') + 'NAME_HOOK'.length() - 1$
            }
        }
        port type Contrib {
            java::statements::ClassMethod.statements is prototype if $self.name = 'PLACEHOLDER'$ {
                point = $'INIT'$
            }
        }
    }
    fragment role Collaboration {
        port type Rec {
            java::statements::JumpLabel is hook if $name.toUpperCase() = name$ {
                point = $name$
                remove = $true$
            }
        }
    }
}
```

Listing 5.9: REX$_{\text{CM}}$ specification that binds Java to the FraCol Participation (cf. Listing 4.2)

```
componentmodel org.reuseware.lib.systems.exchange.cm.java
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
    fragment role Container { port type Rec {} }
    fragment role Value {
        port type Contrib {}
        port type Provider {
            java::commons::NamedElement is slot if $name = 'GIVER'${}
            java::commons::NamedElement.name is value hook if $name = 'ID_SLOT'$ {
                point = $'ID'$
            }
        }
        port type Consumer {
            java::commons::NamedElement is slot if $name = 'TAKER'${}
        }
    }
    fragment role Provider {
        port type Self {
            java::commons::NamedElement is anchor if $name.contains('NAME_HOOK')${}
        }
    }
    fragment role Consumer {
        port type Self {
            java::commons::NamedElement is anchor if $name.contains('NAME_HOOK')${}
        }
    }
}
```

Listing 5.10: REX$_{\text{CM}}$ specification that binds Java to the FraCol Exchange (cf. Listing 4.3).

Example 5.3.2. (4/4) This brings us back to the example of a ticket shop system developed with CB-MDSD initially defined in Section 2.1. Listings 5.8, 5.9, and 5.10 show three component model configurations that are required in this example to handle UML use case diagrams and Java code. They realise bindings for the FraCols Participation and Exchange (Figures 4.4 and 4.7). Listing 5.8 defines the interfaces for UML use case fragments needed to compose them as shown in Figure 2.6. Listings 5.9 and 5.10 define the bindings for Participation and Exchange to Java that was already presented in an abstract way in Example 4.1.

5.3.3 REX_{CM} Semantics

Above we explained the meaning of the REX_{CM} concepts and clarified the effect that an interpretation of a REX_{CM} configuration on a graph fragment has. The complete semantics of this interpretation are defined in story diagrams in Appendix A.2. (Consult the introduction of Appendix A for an explanation of the story diagrams formalism.)

5.4 Conclusion

Chapter Contributions This chapter introduced graph fragment support for ISC as part of U-ISC/Graph and therewith contributes the following:

C1-2 **Type-safe invasive composition interfaces for graphs**
We defined novel variability typing concepts for graphs, which extend the variability typing concepts for trees as defined in U-ISC/Tree [Hen09] by introducing three different types of variation and three different types of reference points to distinguish (1) between containment references and cross references as well as (2) between nodes, edge lists, and attributes. [Hen09] only (1) handled containment references and (2) only operated on nodes and gave no special treatment to edge lists or attributes. The variability typing concepts take the specifics of graphs typed by EMOF/Ecore metamodels into account which is the base for invasive component models that support stronger typed and more complex invasive composition interfaces than earlier works.

C1-3 **Abstraction of physical composition interfaces**
We defined novel heterogeneous and homogeneous grouping concepts for composition interfaces of fragments. These allow for forming groups of variation points that concern one port type which has been defined in a FraCol and thus form an abstraction over the physical interface of a fragment. This is the base for binding FraCols to composition interfaces. The idea of grouping variation points is based on on the nested hook concept of ISC [Aßm03] but extends it. In U-ISC/Tree, no comparable concept exists. We allow for grouping of variation and reference points in the same group and support the combination of heterogeneous and homogeneous grouping. This has not been realised in this form before. With this, complexity can be hidden behind interfaces rather than in composition operators. This contribution has the following sub-contribution:

- *Component model configuration language*
  In this chapter, we defined the component model configuration language REX_{CM} for binding fragment roles defined in FraCols to EMOF/Ecore metamodels. With these bindings, the variability typing of a model is derived. Thus the REX_{CM}

allows for variability typing of arbitrary models defined in EMOF/Ecore-based languages. In contrast to the grammar-based component model specification language of U-ISC/Tree, REX$_{\mathrm{CM}}$ does not require modification of a language's metamodel to support new modularity concepts.

While this chapter dealt with the integration of new component languages into the generic CB-MDSD composition system, the next chapter is concerned with integrating composition languages.

6 Universal Composition Language (UCL)

This chapter describes the Universal Composition Language (UCL) and a composition technique based on universal operators to execute composition programs defined in UCL. On top of UCL, a formalism is provided to integrate modelling languages as composition languages into a CB-MDSD composition system.

In the previous chapter, we explained how models are treated as graph fragments and presented a technique to define composition interfaces to access parts of these fragments. This chapter explores how these interfaces are used in composition programs to define compositions of graph fragments. These composition programs are defined in the *Universal Composition Language* (UCL) and are executed by a universal composition algorithm. Both, UCL and the algorithm, are developed in this chapter. In contrast to other composition languages used in ISC so far, UCL can be used to compose arbitrary fragments without adjustments; in this sense UCL is universal. For this, it exploits FraCols as means to specify abstract relationships between composition programs—defined in UCL—and graph fragments. In earlier ISC approaches, composition languages were directly coupled to operators that had dependencies to component languages and thus needed adjustments for different types of fragments. This section hence provides the remaining four features of our generic CB-MDSD composition system as illustrated in Figure 6.1.

- (Feature 5) To enable the physical composition of graph fragments, we have to extend the basic composition operators of ISC to work with graph fragments, which we do in Section 6.1.
- (Feature 6) The concepts to define composition programs with UCL are introduced in Section 6.2. They are based on the concepts of FraCols and graph fragment interfaces.
- (Feature 7) The universal composition algorithm, which interprets composition programs defined in UCL, is defined in Section 6.3. It translates a UCL composition program into calls to the basic composition operators and therewith performs the invasive composition of graph fragments.

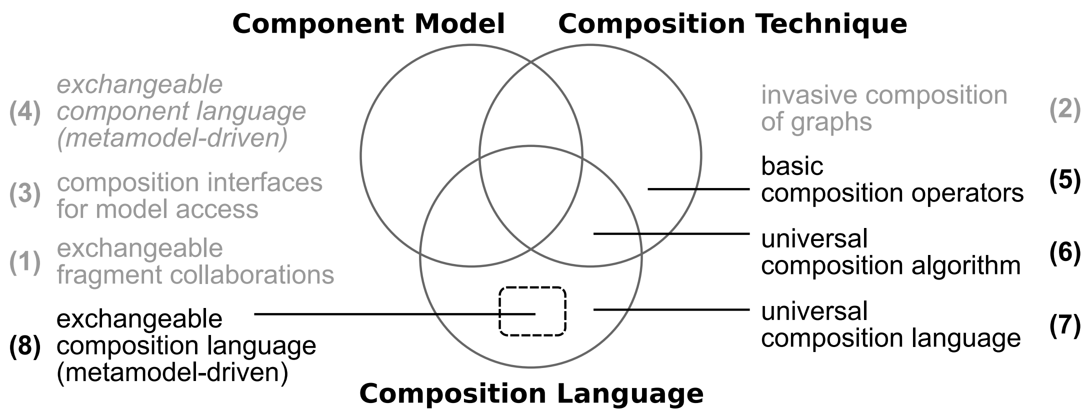

Component Model
Composition Technique
Composition Language
(4) exchangeable component language (metamodel-driven)
(3) composition interfaces for model access
(1) exchangeable fragment collaborations
(8) exchangeable composition language (metamodel-driven)
invasive composition of graphs (2)
basic composition operators (5)
universal composition algorithm (6)
universal composition language (7)

Figure 6.1: Features of the generic composition system for CB-MDSD that concern the universal composition language.

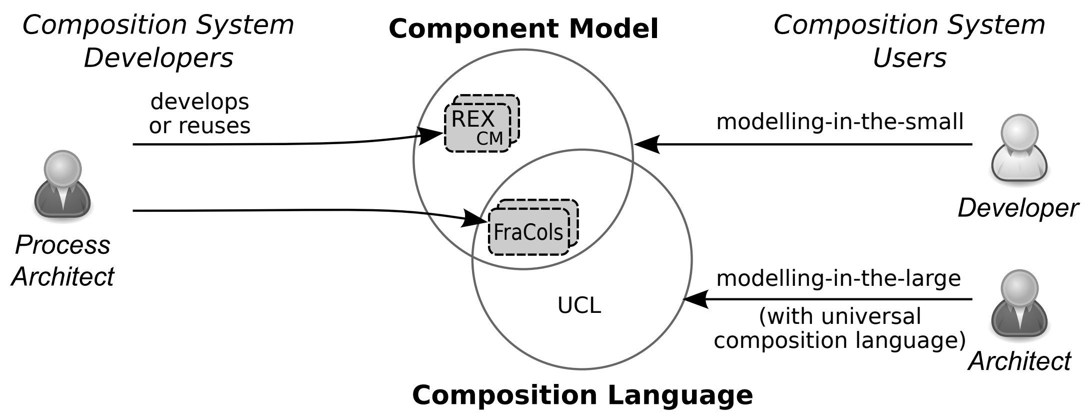


Figure 6.2: UCL used directly in a CB-MDSD composition system.

Afterwards, we discuss the usage of UCL in a CB-MDSD composition system in Section 6.4. Figure 6.2 illustrates that UCL can be used directly by architects, in the role of composition system users, to define composition programs. In this case, no integration effort for a composition language is required from a composition system developer.

- (Feature 8) UCL is a base composition language through which other composition languages can be integrated into a composition system. To enable composition system developers to specify such integrations, we define the language REX_{CL}. Figure 6.3 shows that REX_{CL} can be used by a process architect, in the role of a composition system developer, to integrate DSMLs as new composition languages into a CB-MDSD composition system. This in turn allows architects to develop composition programs in these DSMLs. REX_{CL} is defined in Section 6.5.

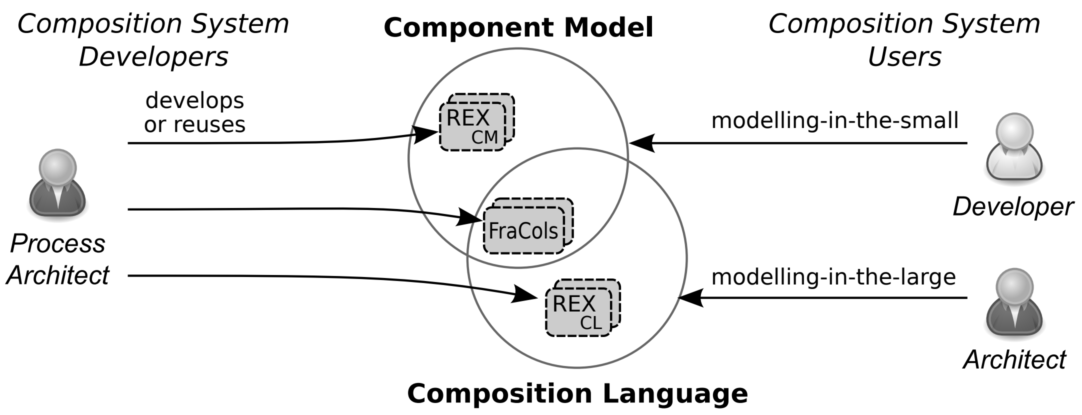


Figure 6.3: REX_{CL} definitions integrate new composition languages into the generic CB-MDSD composition system.

6.1 Basic Composition Operators

A powerful feature of ISC is that it provides a common composition technique on which multiple invasive composition systems build. This basic composition technique is founded on a set of well-defined basic composition operators. In ISC and U-ISC/Tree, the basic operators worked on trees and composed these trees. To facilitate graph fragment composition in a similar fashion, we redefine the basic composition operators using graph rewriting rules, leveraging the composition technique from tree to graph composition. For this, the reflective access to graph representations of models, discussed in Section 5.1.3, is utilised.

The basic composition operators of ISC defined in [Aßm03, Section 4.3] are *Bind* and *Extend*. The difference between the Bind and the Extend operator is that Bind applied on a hook replaces the hook (i.e., it removes the hook from its containing fragment) while Extend applied on a hook does not modify the hook itself but uses it as a position for extension (i.e., the hook remains in its containing fragment). Extend can only work on hooks in lists. The position to where a new fragment is added to the list upon extension is unspecified. However, [Aßm03, Section 4.3] introduces the complex operators *Prepend* and *Append* that can be used to perform extensions before or after the hook respectively.

The composition algebra of U-ISC/Tree [Hen09, Section 3.2.2] refines the basic composition operators. U-ISC/Tree therewith distinguishes between *slots* (i.e., tree nodes that are replaced) and implicit *hooks* (i.e., places where new tree nodes can be inserted). [Hen09, Section 3.2.2] argues that the Bind operator can effectively be used for extension by binding a slot with a list that contains fragments and a new slot. Still the Extend operator is needed for implicit hooks, where no tree nodes are present that explicitly mark a slot. In this case, the Prepend and Append operators also need to be available as basic operators since they cannot be expressed as combinations of Bind and Extend calls. Thus, Prepend and Append are also basic operators (defined as different modes of Extend) in U-ISC/Tree [Hen09, Section 3.2.2].

We make the basic composition operators from U-ISC/Tree applicable for graph fragment composition by defining them in terms of graph rewriting rules (Figures 6.6–6.10). For us, *Bind*, *Extend*, *Prepend*, and *Append* are all basic composition operators. In addition, we introduce

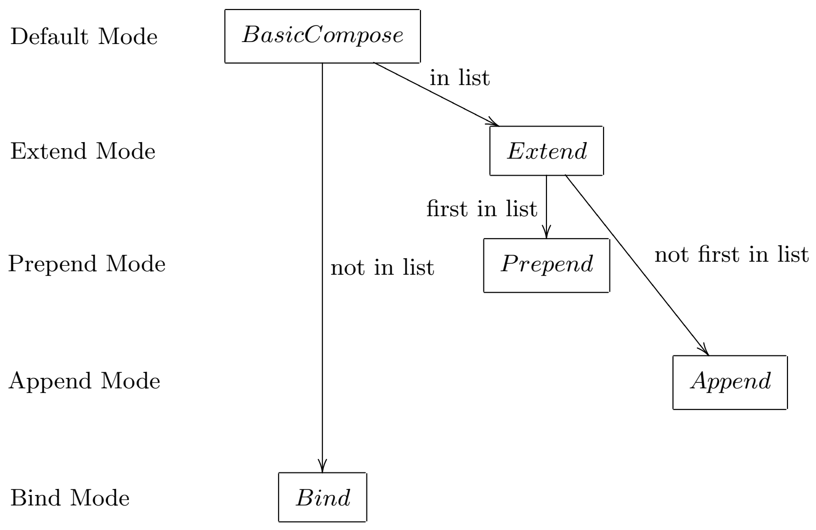
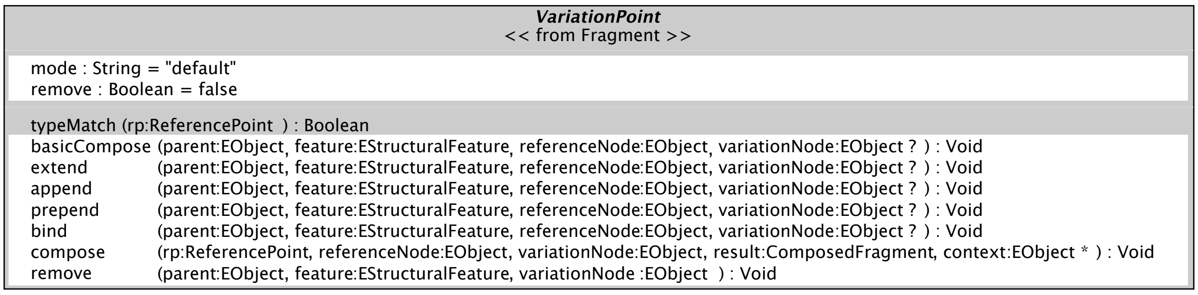


Figure 6.4: Delegation between basic composition operators.



```
VariationPoint
<< from Fragment >>

mode : String = "default"
remove : Boolean = false

typeMatch (rp:ReferencePoint ) : Boolean
basicCompose (parent:EObject, feature:EStructuralFeature, referenceNode:EObject, variationNode:EObject ? ) : Void
extend       (parent:EObject, feature:EStructuralFeature, referenceNode:EObject, variationNode:EObject ? ) : Void
append       (parent:EObject, feature:EStructuralFeature, referenceNode:EObject, variationNode:EObject ? ) : Void
prepend      (parent:EObject, feature:EStructuralFeature, referenceNode:EObject, variationNode:EObject ? ) : Void
bind         (parent:EObject, feature:EStructuralFeature, referenceNode:EObject, variationNode:EObject ? ) : Void
compose      (rp:ReferencePoint, referenceNode:EObject, variationNode:EObject, result:ComposedFragment, context:EObject * ) : Void
remove       (parent:EObject, feature:EStructuralFeature, variationNode :EObject ) : Void
```

Figure 6.5: `VariationPoint` enriched with basic composition and complex `compose()` operations (cf. Figure 5.21).

the *BasicCompose* operator that may be used universally instead of explicitly selecting one of the other operators as explained below. Only Bind, Prepend, and Append perform direct composition operations by graph rewriting. BasicCompose and Extend delegate to one of the other three operators as described in the following.

In our composition technique, the composition operators are decoupled from the composition language (cf. Figure 6.1). This means that there will be no explicit operator call concept in the universal composition language. Rather, composition operators will be automatically selected and called implicitly. By default, this automation calls BasicCompose, which delegates to another operator depending on the nature of the variation point it is called on by using a default delegation strategy. Still, developers should be able to influence this delegation in cases where the nature of the variation point would allow different operators (e.g., both Prepend and Append can operate on a variation point in a list).

Therefore, we allow developers to influence the delegation strategy via the variation point itself for which we add a *mode* flag (cf. Figure 6.5). The mode can be set to *default*, *extend*, *prepend*, *append*, or *bind*. If no mode is specified, *default* is automatically selected. Figure 6.4 shows the operators and how they delegate to each other. In default mode, the decision between

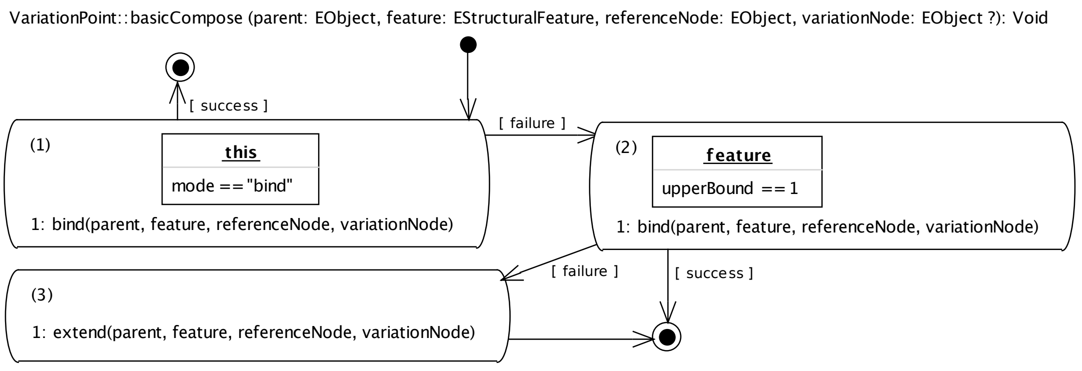

VariationPoint::basicCompose (parent: EObject, feature: EStructuralFeature, referenceNode: EObject, variationNode: EObject ?): Void
[success]
[failure]
(1)
this
mode == "bind"
1: bind(parent, feature, referenceNode, variationNode)
(2)
feature
upperBound == 1
1: bind(parent, feature, referenceNode, variationNode)
[failure]
[success]
(3)
1: extend(parent, feature, referenceNode, variationNode)

Figure 6.6: `VariationPoint.basicCompose()`: The *Basic Compose* composition operator.

Bind and Extend depends on whether we operate on a single node or a list of nodes. The decision between Prepend and Append—in *extend* or *default* mode—is based on the position of the variation point node—if any—in the list of nodes. The details are explained in the following on the specifications of the basic operators (Figures 6.6–6.10). All basic operators are available as operations on `VariationPoints` (Figure 6.5).

To give composition developers the possibility to influence the *mode* of a variation point, we add a `mode = $OCLExpression$` statement to REX_{CM} that can be used in derivation rules to define how the mode flag is computed for a concrete variation point (cf. Listing 5.4). In the following the basic composition operators are defined using story diagrams (for the notation consult Appendix A).

6.1.1 Basic Compose

The BasicCompose operator is defined in Figure 6.6. It delegates to other basic operators as shown in Figure 6.4. It honors the mode setting of the variation point that `basicCompose()` is called on.

1. At first, it is checked if this `VariationPoint` is in *bind* mode. If this is the case, the composition is forwarded to the bind operator.
2. In all other modes, the multiplicity of the `EStructuralFeature` that types the edge, which is created to the reference point upon composition, is used to decide whether to call `bind()`...
3. ... or `extend()`.

6.1.2 Extend

The Extend operator is specified in Figure 6.7. It is realised by delegating to the Prepend or Append operators as shown in Figure 6.4. It honors the mode setting of the variation point `extend()` is called on.

1. At first, we determine the `Setting` which is modified in this `extend()` call.
2. If this `VariationPoint` is in *prepend* mode, we forward to `prepend()`.

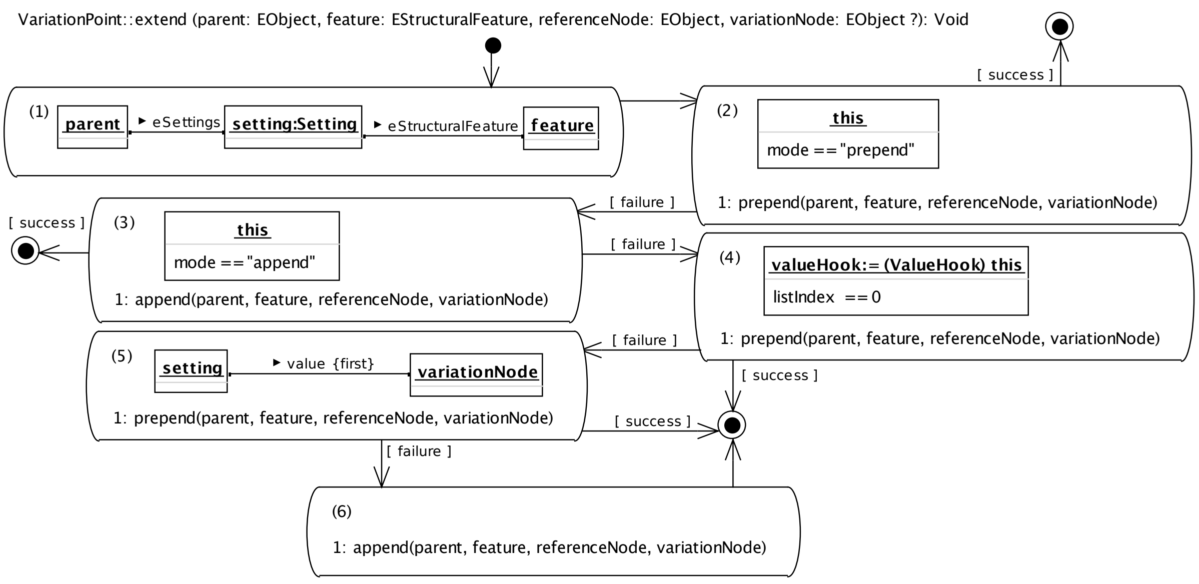

VariationPoint::extend (parent: EObject, feature: EStructuralFeature, referenceNode: EObject, variationNode: EObject ?): Void
(1)
parent
▸ eSettings
setting:Setting
▸ eStructuralFeature
feature
(2)
this
mode == "prepend"
1: prepend(parent, feature, referenceNode, variationNode)
[success]
[failure]
(3)
this
mode == "append"
1: append(parent, feature, referenceNode, variationNode)
[success]
[failure]
(4)
valueHook:= (ValueHook) this
listIndex == 0
1: prepend(parent, feature, referenceNode, variationNode)
[failure]
[success]
(5)
setting
▸ value {first}
variationNode
1: prepend(parent, feature, referenceNode, variationNode)
[success]
[failure]
(6)
1: append(parent, feature, referenceNode, variationNode)

Figure 6.7: `VariationPoint.extend()`: The *Extend* composition operator.

3. If this `VariationPoint` is in *append* mode, we forward to `append()`.
4. In *extend* or *default* mode: If this is a `ValueHook`, the `listIndex` gives the place to which a value should be added in a list of values (i.e., a list attribute represented by the current `Setting`). If `listIndex` is 0, we forward to `prepend()`.
5. In case a node is composed, we check if an explicit variation node exists (passed as parameter) and if this node is the first in the current setting. If it is, we delegate to `prepend()`.
6. If we are not at the first position, we forward to `append()`.

6.1.3 Prepend

The Prepend operator is specified in Figure 6.8 and directly modifies the `Setting` identified by the given parent node and feature.

1. At first, we determine the `Setting` which is modified in this `prepend()` call.
2. We check if a node or a string value is composed, by checking if this `VariationPoint` is a `ValueHook`.
3. In the case that a node is composed, we create a new edge in the edge list (which is represented by the current `Setting`) after the edge to the variation node, if a variation node is given.
4. If no variation node is given, the edge is created as the first edge of the edge list.
5. In case a string value is prepended to a list attribute (i.e., the `Setting` represents a list attribute), a new entry is created in the list attribute at the position given by the `listIndex` of the `ValueHook`. The value itself is extracted from the `ValuePrototype` which is passed in as `referenceNode`, because the string value—being an attribute and not a node—cannot be passed directly.

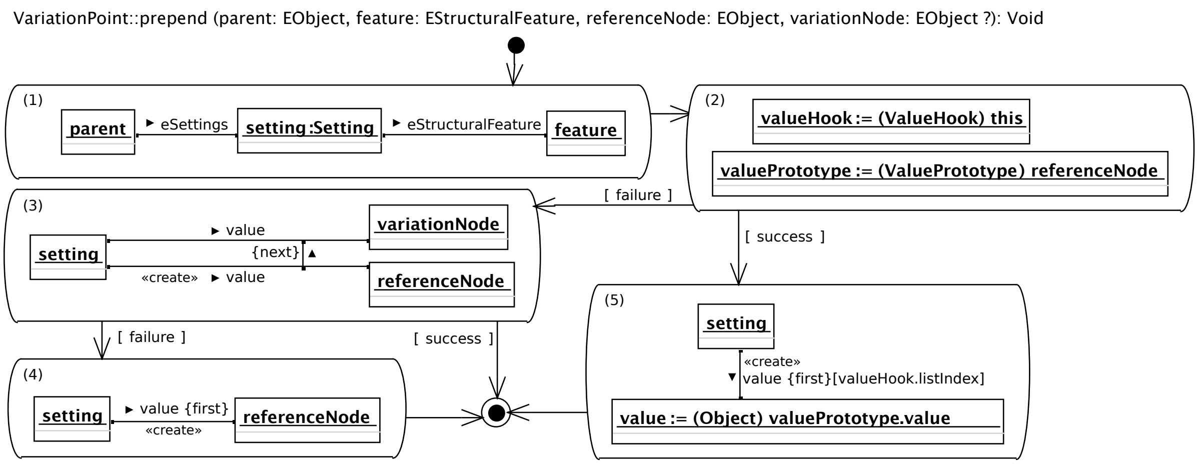

VariationPoint::prepend (parent: EObject, feature: EStructuralFeature, referenceNode: EObject, variationNode: EObject ?): Void
(1)
parent
▸ eSettings
setting :Setting
▸ eStructuralFeature
feature
(2)
valueHook := (ValueHook) this
valueProtoype := (ValuePrototype) referenceNode
[failure]
[success]
(3)
setting
▸ value
{next} ▴
«create» ▸ value
variationNode
referenceNode
[failure]
[success]
(4)
setting
▸ value {first}
«create»
referenceNode
(5)
setting
«create»
▾ value {first}[valueHook.listIndex]
value := (Object) valuePrototype.value

Figure 6.8: `VariationPoint.prepend()`: The *Prepend* composition operator.

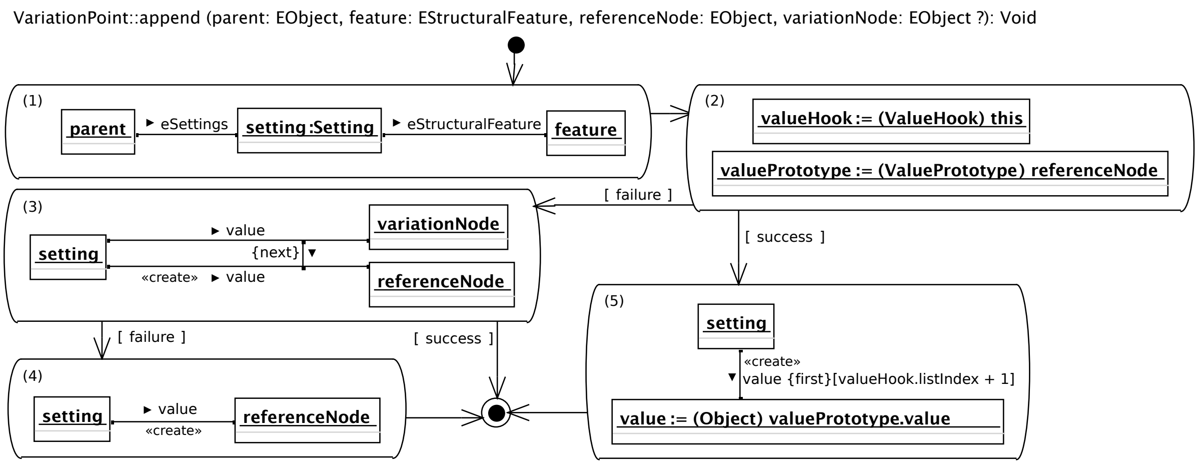

VariationPoint::append (parent: EObject, feature: EStructuralFeature, referenceNode: EObject, variationNode: EObject ?): Void
(1)
parent
▸ eSettings
setting :Setting
▸ eStructuralFeature
feature
(2)
valueHook := (ValueHook) this
valueProtoype := (ValuePrototype) referenceNode
[failure]
[success]
(3)
setting
▸ value
{next} ▾
«create» ▸ value
variationNode
referenceNode
[failure]
[success]
(4)
setting
▸ value
«create»
referenceNode
(5)
setting
«create»
▾ value {first}[valueHook.listIndex + 1]
value := (Object) valuePrototype.value

Figure 6.9: `VariationPoint.append()`: The *Append* composition operator.

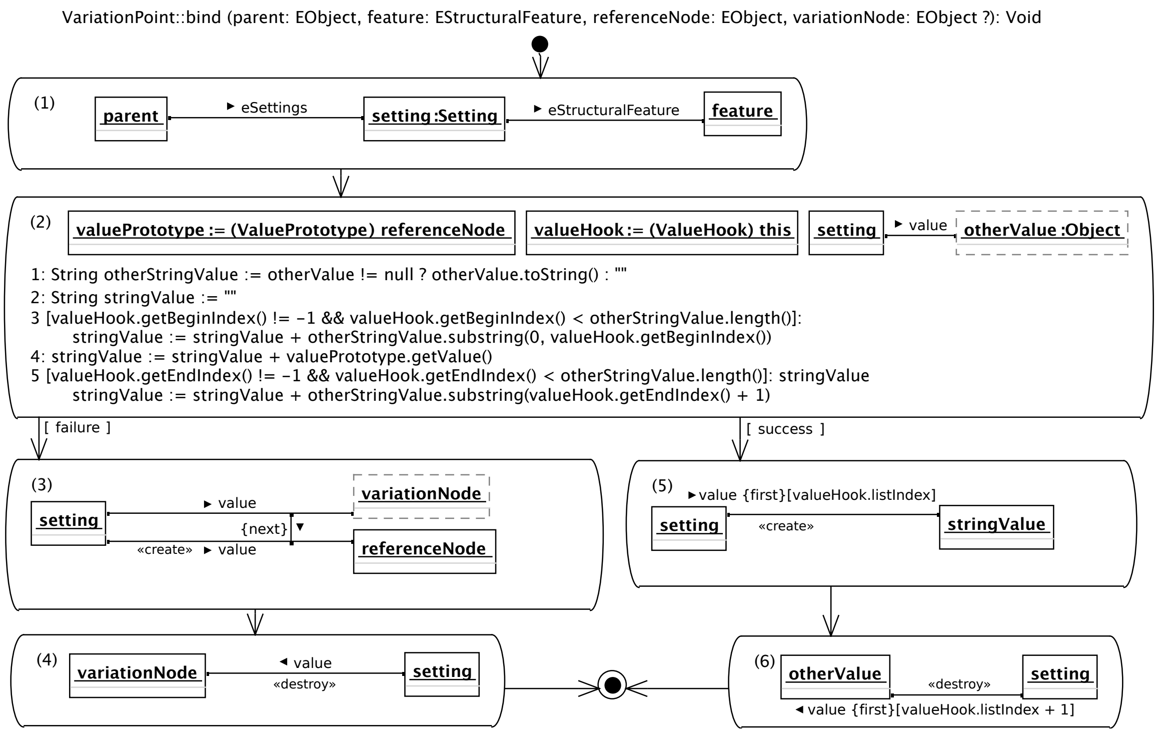
6.1.4 Append

The Append operator is specified in Figure 6.9 and directly modifies the `Setting` identified by the given parent node and feature. It is built similar to the Prepend operator (cf. Figure 6.8) only that the new edge is created after the given variation node in the edge list (Pattern 3) or, if no variation node is given, at the end of the edge list (Pattern 4). An attribute value is inserted after the given index in a list attribute (Pattern 5) and not before.

6.1.5 Bind

The Bind operator is specified in Figure 6.10 and directly modifies the `Setting` identified by the given parent node and feature.

1. At first, we determine the `Setting` which is modified in this `bind()` call.



```
1: String otherStringValue := otherValue != null ? otherValue.toString() : ""
2: String stringValue := ""
3 [valueHook.getBeginIndex() != -1 && valueHook.getBeginIndex() < otherStringValue.length()]:
      stringValue := stringValue + otherStringValue.substring(0, valueHook.getBeginIndex())
4: stringValue := stringValue + valuePrototype.getValue()
5 [valueHook.getEndIndex() != -1 && valueHook.getEndIndex() < otherStringValue.length()]: stringValue
      stringValue := stringValue + otherStringValue.substring(valueHook.getEndIndex() + 1)
```

Figure 6.10: `VariationPoint.bind()`: The *Bind* composition operator.

2. We check if a node or a string value is composed, by checking if this `VariationPoint` is a `ValueHook`. In the case of a string value, the `beginIndex` and `endIndex` attributes need to be taken into account (cf. Section 5.1.3), since the new attribute value may replace only parts of the existing attribute value. Since this operation is not a graph rewriting but a string manipulation, we use Java's string API for this. This API can be accessed in collaboration statements. In these statements, the part of the existing value before `beginIndex` is prepended and the part of the existing value behind `endIndex` is appended to the value that should be bound.
3. In case a node is bound, a new edge is created at the position of the edge to the variation point. Specifying the position is important in cases where bind mode is explicitly selected and binding in the context of an edge lists is performed.
4. The edge to the variation node is destroyed. Note that if the setting represents a single edge (and not an edge list), this edge is removed already at the moment the new edge is created above (because there can only be one edge of this kind).
5. and 6. Adding the computed string value to the attribute or list attribute that is represented by the `Setting` is done in a similar fashion as the creation and deletion of new and old edges in Patterns 3 and 4.

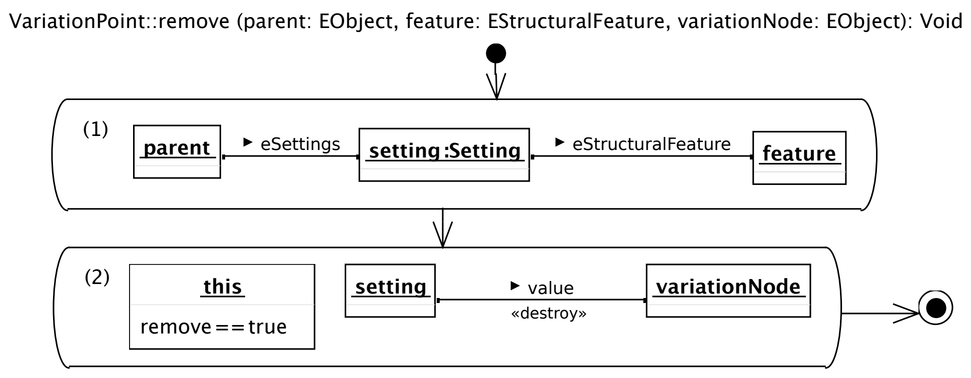


Figure 6.11: `VariationPoint.remove()`: The *Remove* composition operator.

6.1.6 Remove

In addition to the above presented operators, there is another basic operator Remove defined in Figure 6.11 for removing variation nodes from a composed fragment. Remove only has an effect on variation points that are nodes and for which the `remove` flag is set to `true` (cf. Figure 6.5). Thus, `remove()` can be called after a composition has been completed on all variation points to clean the interface of variation nodes that are no longer needed and should not appear in the composed model. Similar as for mode, REX_{CM} is extended with a `remove = $OCLExpression$` statement to control which variation nodes may be removed.

Above we described the basic composition operators of U-ISC/Graph. As mentioned, these operators are implicitly called when composition programs are executed. How these are defined is presented next.

6.2 Universal Composition Language Concepts

In Chapter 2 we motivated the need for a universal composition language. We formulated the following requirements for composition programs defined in that language:

- **Universal Composition Language** (Requirement 4; p. 16) The language has to be independent of component languages in so far that components defined in arbitrary component language can be used in composition programs.
- **Abstract Composition Interfaces** (Requirement 3; p. 16) Composition programs must allow to exchange the set of components they compose for another set of components with similar interfaces defined in another component language.
- **Aggregatable Composition Programs** (Requirement 5; p. 17) It must be possible to extract composition information from different sources and aggregate them in one composition program.

The prerequisite to meet these requirements is the fragment collaboration concepts of Chapter 4. In existing ISC approaches, these collaborations are hidden in component language-dependent composition operators. This couples composition programs, which directly call these operators, tightly to component languages. By explicitly defining fragment collabora-

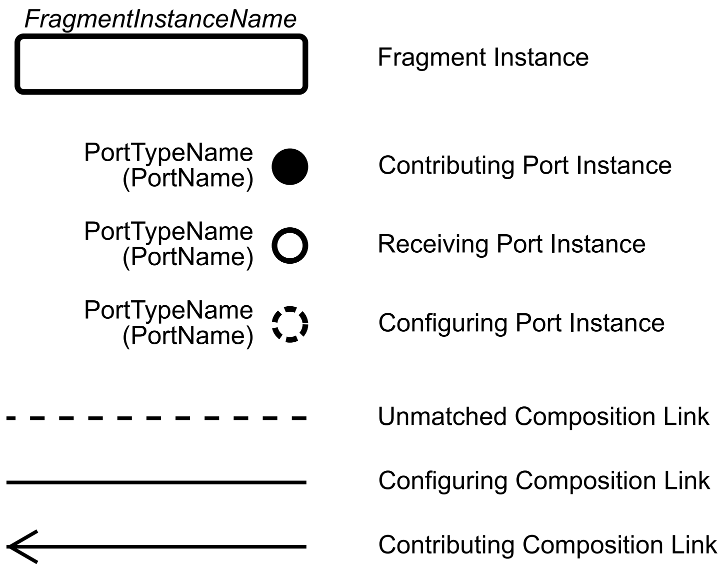

FragmentInstanceName
Fragment Instance
PortTypeName (PortName)
Contributing Port Instance
PortTypeName (PortName)
Receiving Port Instance
PortTypeName (PortName)
Configuring Port Instance
Unmatched Composition Link
Configuring Composition Link
Contributing Composition Link

Figure 6.12: Notation for UCL.

tions independent of component languages, and only binding them later to these languages with REX_{CM}, we are able to define composition programs that are decoupled from component languages to meet Requirements 3 and 4.

To meet Requirement 5, it is important that composition programs are declarative. In existing ISC approaches, composition programs are written and interpreted imperatively as a sequence of operator calls, where each call potentially depends on the composition results of the previous operator call. To ensure that aggregated composition programs always deliver the same result, independent of the order in which composition information is aggregated, we need to avoid this sequential dependency.

Therefore, we define the Universal Composition Language (UCL) that is a declarative composition language based on the fragment collaboration concepts defined in Chapter 4. In the following, we introduce the concepts of UCL using a graphical notation which is summarised in Figure 6.12. Occasionally, we explain some details best seen in the language's metamodel shown in Figure 6.13. The semantics of the language, that is, the translation of declarative composition programs into composition operator calls, is defined in Section 6.3.

6.2.1 Fragment Instance

A *fragment instance*, represented as a box (cf. Figure 6.12), represents a graph fragment in a composition program. In contrast to a fragment (cf. Figure 5.21), a fragment instance is linked to other fragment instances in the context of a composition program. Different instances of the same fragment can have different links in another or in the same composition program. Different fragment instances in one composition program are distinguished through unique names inside the program (`name` attribute of fragment instance). A specific kind of fragment instance is a *referenced fragment* (`reference` attribute of fragment instance). A referenced

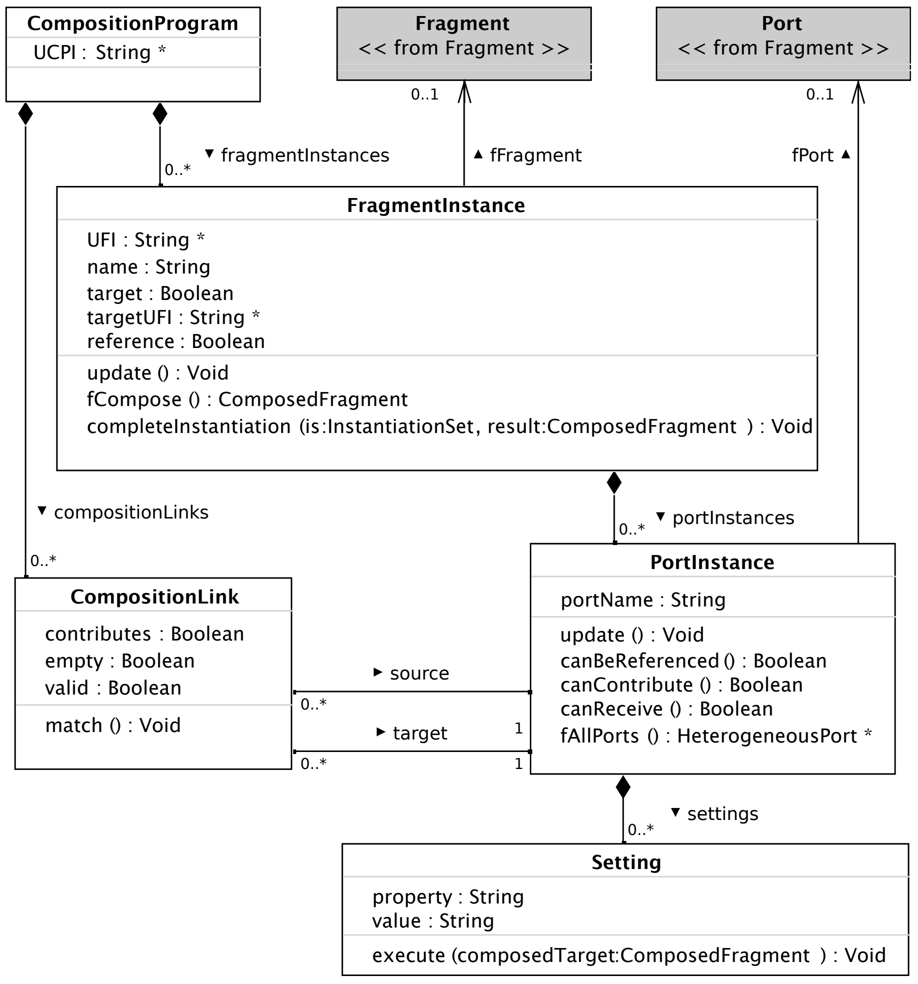


Figure 6.13: Universal Composition Language (UCL) concepts.

fragment is not copied during composition as other fragments are which is described in more detail in Section 6.3.3. The `target` and `targetUFI` attributes of fragment instances are required to define UFIs for composition results such that composed fragment can also be identified for further reuse in other composition programs.

6.2.2 Port Instance

A fragment instance has one *port instance* for each port on its fragment's composition interface. Port instances are visualised as circles attached to the fragment instance boxes. We distinguish three kinds of port instances that depend on the nature of the ports they represent, which has an influence on the composition execution that is discussed in Section 6.3. If a port contains hooks, it is *receiving* depicted by a circle with a solid border but is not filled (cf. Figure 6.12). If a port contains prototypes, it is *contributing* depicted by a filled circle. If a port instance contains only slots and anchors, it is *configuring* depicted by a circle with a dashed outline.

6.2.3 Composition Link

To define invasive compositions of graph fragments, we introduce the concept *composition link*. A composition link has two states: *unmatched* and *matched*. A newly defined link is unmatched depicted by a dashed line (cf. Figure 6.12). It is matched for composition as we discuss in Section 6.3.1. A matched link is depicted by a solid line and if it connects a contributing with a receiving port instance, the composition direction is visualised with an arrow (from the port with prototypes to the port with hooks).

6.2.4 Value Setting

The last UCL concept is *value setting*. A value setting holds a `property/value` pair and can be used to set a value hook to a string value rather than linking it to a value prototype. Consequently, a port instance may have multiple settings. The `property` of the value identifies the value hook to be set by its name. The `value` specifies the value to be set.

Example 6.2. Figure 6.14 (top) shows a composition program that defines a composition between the file system and observer UML models known from earlier examples that were depicted with their composition interface in Figures 5.22 and 5.23. These composition interfaces were obtained by interpreting a REX_{CM} specification (cf. Listing 5.5) that binds the FraCol ClassWeaving (cf. Listing 4.4) to UML.

In this example, the observer UML model is reused twice. Once to define an observer/subject relationship between `FileSystem` and `FSFolder` and once between `FileSystem` and `FSFile`. The corresponding ports contain all necessary variation and reference points. Comparing this composition program with Figures 5.22 and 5.23, illustrates how the addressable points are hidden to the developer of the composition program—he or she only sees the ports on the interfaces and not the internal details.

In the middle and bottom of Figure 6.14, the composed model that results from executing the composition program is shown in graphical UML notation and as graph. We can observe that two copies of the observer fragment were created, because it was instantiated twice in the composition program. For each prototype/hook binding, a separate copy was used (e.g., there are two copies of the `attach` operation (F) that are bound to different hooks). For anchor/slot bindings, no additional copies are produced (e.g., the `FileSystem` class (2) is used as type for the `observer` parameter (I) of both copies of the `attach` operation). How the composition execution that produced this result works exactly is described in the next section.

The composition links do not depend on parts of graph fragments directly. They only connect ports which have port types defined in the FraCol ClassWeaving. Therefore, the fragments behind the fragment instances can be exchanged for other fragments that provide the same composition interfaces conforming to the ClassWeaving collaboration. Thus we could compose the Java version of the file system example (cf. Figures 5.28 and 5.29) with the same composition program. The composition technique, which we present next, is still able to perform the composition, because the knowledge about how to access the Java fragments is defined in the corresponding REX_{CM} in Listing 5.6.

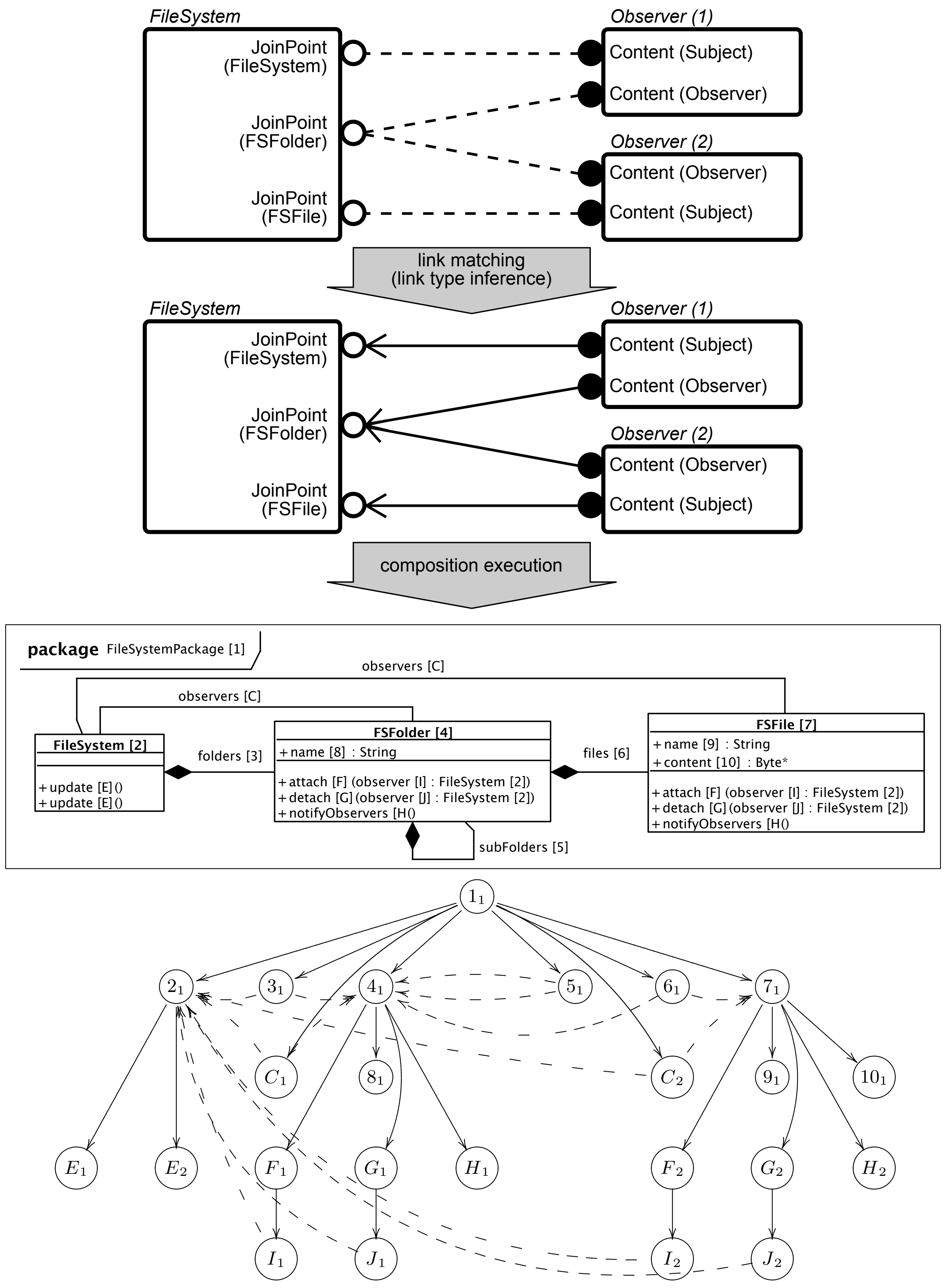


Figure 6.14: Composing file system (cf. Figure 5.22) and observer (cf. Figure 5.23) models.

6.3 Universal Composition Language Semantics

A composition program can be executed to obtain an invasively composed graph fragment. For this, the composition program is analysed and translated into basic composition operator calls. For this, a couple of steps are necessary which are described in the following. The full semantics of composition program execution are formally defined through story diagrams in Appendix A.3.

6.3.1 Composition Link Matching

Figure 6.15 gives an overview of the composition execution illustrated on the stepwise processing of a composition program that contains three fragments (F1, F2, and F3). The first activity (a) is the matching of each composition links. Matching a composition link is essentially the identification of basic composer calls between the addressable points of the two connected ports. This is done by checking for each pair of addressable points if their domain types, their variability types (according to the rules defined in Section 5.2.2), and their names match.

The goal of the match is to find for each involved reference point exactly one variation point to which it is composed later. Consequently, the match fails if there are reference points that were not or ambiguously matched. That is, they were either matched to none or more than one variation point. If the match fails, the composition link is invalid and ignored in the composition execution.

Composition link matching is a type inference activity. It determines if a composition link is contributing or configuring and which are the composition operator calls that need to be executed. Conceptually, one could also define operator calls directly and instead of performing a type inference, we could perform a type check. This is how it was done in COMPOST and U-ISC/Tree (cf. Chapter 3).

6.3.2 Composition Step Identification

The composition execution follows a recursive algorithm that is performed in several *composition steps*. One composition step resembles the execution of all composition operator calls of all incoming contributing links and all configuring links of one fragment instance. We call this fragment instance the *receiving fragment* of the composition step. All fragment instances connected to it via incoming contributing links are consequently *contributing fragments*.

If a contributing fragment can itself act as receiving fragment in another composition step (i.e., it has itself incoming contributing links) the other step hast to be executed first. This means that initially the composition steps which involve fragment instances that do not have incoming contributing links (i.e., they never act as receiving fragment) are identified and executed. After that, the receiving fragment of such a step has no incoming contributing links anymore since these have been processed. Therefore, it may now act as contributing fragment in other composition steps.

In Figure 6.15 two composition steps are found and executed. The first step (b) composes F3 with F2. It is identified because there is a contributing link between F3 and F2 and because

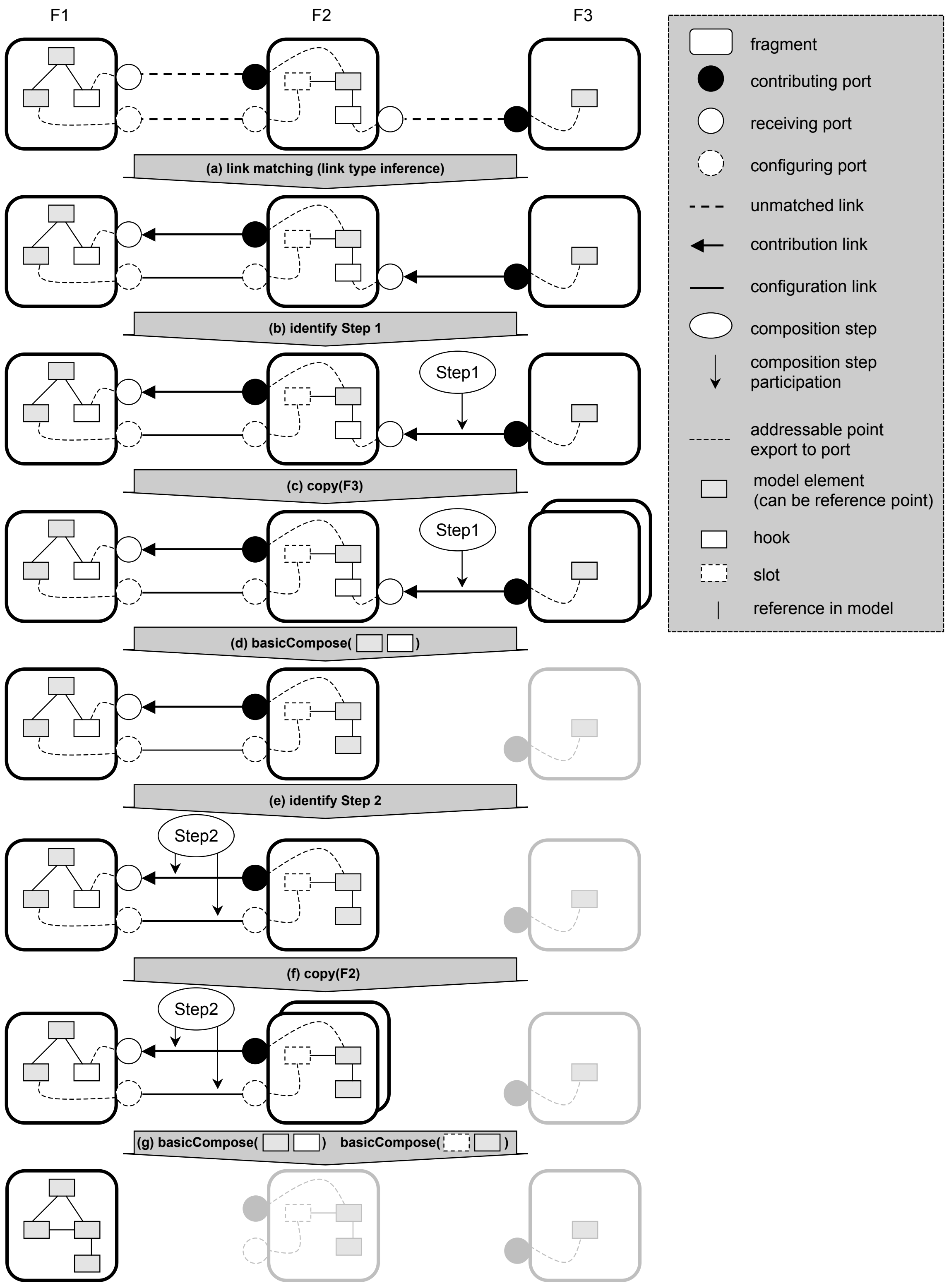


Figure 6.15: Stepwise execution of a UCL composition program.

F3 itself does not have incoming composition links. After the composition was completed, the second step is identified between (the now composed) F2 and F1, because of the contributing link between F2 and F1 and because the composed version of F2 does not have an incoming contributing link anymore. In addition to contributing links, configuring links (as the one between F1 and F2) and value settings of the receiving fragment are processed in each step.

6.3.3 Fragment Copying

Before a fragment is composed, it is copied. Copying fragments is the basis for invasive composition. It is necessary, since we need to modify graphs in invasive composition, without altering the original graphs. The copy operation for fragments was originally introduced as basic composition operator in [Aßm03, Section 4.3]. The use of this copy operator in COMPOST was demonstrated in Example 3.1. While copying had to be called explicitly in COMPOST, we implicitly trigger copying whenever a composition step is executed. In Figure 6.15, F3 is copied before it is composed with F2 (c) and the result of this composition is copied before it is composed with F1 (f).

Usually, at least one copy is created per fragment involved in a composition step. An exception are fragment instances that are set to `reference`. In such a case, cross-references to the original fragment are created during composition. To ensure that only cross-references are created, the port of that fragment may only contain anchors.

Another special case is the involvement of a homogeneous port. In that case, a composition step is triggered multiple times, leading to repetitive copying of the contributing fragment. One copy is created for each heterogeneous port behind the homogeneous port. This is necessary, because a prototype can only be bound once and such a separate copy is required for each hook, as illustrated in the following example.

Example 6.3.3. In Figure 6.16 (top), a composition that involves a homogeneous port is shown. It extends the file system UML model with the ID advice from Figure 5.24. Here, only one instance of the ID fragment is declared in the composition program. The receiving port `FileSystemPackage` however, is a homogeneous port that groups all three heterogeneous ports for the classes of the `FileSystemPackage` package (cf. Figure 5.24). Thus, since there are three heterogeneous ports with individual hooks, three copies of the ID fragment are produced during composition, as we can see in the composed model in Figure 6.16 (middle and bottom).

6.3.4 Basic Composition Operator Calls

During each composition step, the basic composition operators are called according to the link matches. It is important that the operators are called on the copies that were created for the step, and not the original graphs. Furthermore, a reference or a variation point in graph fragments can concern multiple nodes and edges (cf. Section 5.1.3). For example, a variation node can be connected by multiple edges.[1] All those edges need to be considered in a composition that involves the variation point. The basic composition operators introduced above only

[1]Note that the terms variation (reference) node and variation (reference) point are not to be confused: A variation (reference) node is exactly one node in a graph while a variation (reference) point describes a point of variation (for referencing) in a graph which may concern multiple nodes, edges, or attributes.

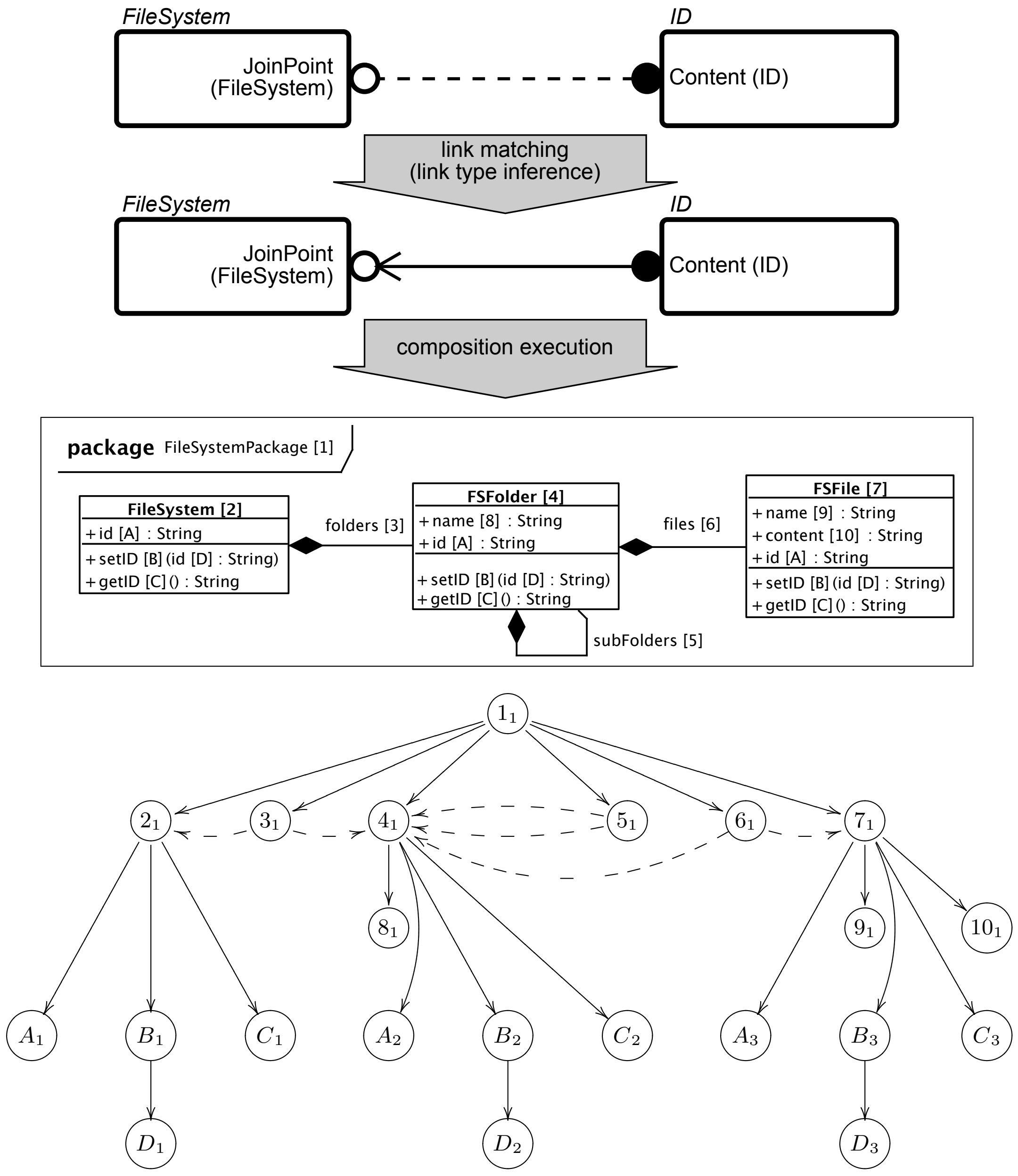


Figure 6.16: Composing file system (cf. Figure 5.22) and ID (cf. Figure 6.14) models.

modify one edge per call and are thus called for each edge individually. In Figure 6.15, BasicCompose is called once in the first composition step (d) and twice in the second composition step (g).

6.3.5 Variation Point Removal

After composition execution, it is useful to remove remaining unbound variation points, if the result of a composition is to be interpreted by other external tools (e.g., a compiler) that do not understand the semantics of the variation points. This concerns cases where variation nodes are involved (i.e., not cases where edge lists or attributes are modified) and where the nodes were not already removed by a `bind()` operator call. If composition is complete, removing variation points is harmless, since they are not required for further composition steps. Removal is performed by the basic operator Remove defined in Section 6.1.6.

With this, we conclude the description of the UCL composition semantics. We are now able to declaratively specify composition programs for graph fragments that are variability typed—as defined in Chapter 5—with the composition language concepts defined in Section 6.2. These composition programs can then be executed as explained in this section. To make this applicable in practice, means are required to define composition programs with UCL, which are discussed in the next two sections.

6.4 Using the Universal Composition Language

This section explains how UCL is integrated directly into a composition system for composition program definition. As we illustrated in Figure 6.2, no integration effort for the composition language is required by a composition system developer if UCL is used. In this case, UCL is the only composition language in the CB-MDSD composition system. Such composition systems are used for the ModelHiC architectural style that is utilised for a larger example in Chapter 9.

6.4.1 Fragment Instantiation

Developers using UCL have to do two things: select fragments to add them to a composition program and create composition links. The creation of fragment instances and their port instances, based on the fragments the developer selects, can be automated. This is a straight forward operation—one fragment instance is created per fragment and one port instance per port. The operation is defined in Appendix A.4. This tooling for UCL is part of Reuseware presented in Chapter 7.

Example 6.4.1. Consider again the REX_{CM} specification in Listing 5.5 and the UML fragments from Figures 5.22 and 5.23. The composition interfaces shown in these two figures are computed based on the REX_{CM} specification (cf. Section 5.3).

A developer can decide to add these fragments to a UCL composition program, which triggers the creation of a fragment instance for each usage of one of them. In the example shown in Figure 6.14, the file system fragment is instantiated once and the observer fragment twice.

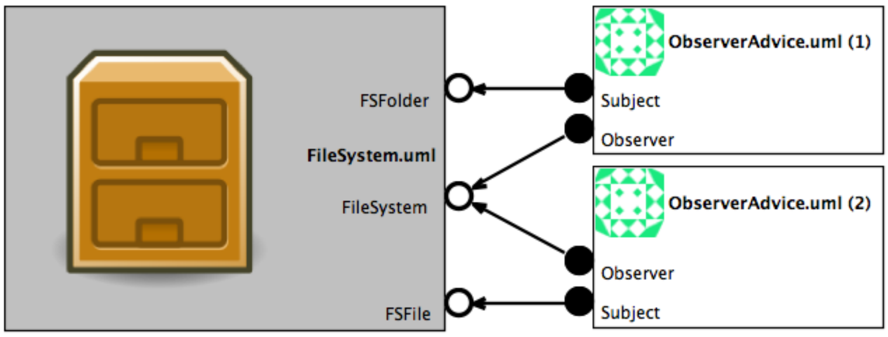


Figure 6.17: Composing file system and observer models using a composition program with customised syntax (cf. Figure 6.14).

The developer can now define composition links between the fragment instances as done in Figure 6.14 top. The composition execution (cf. Section 6.3) can then be performed, which analyses the composition program and automatically calls the corresponding composition operators. It can also give feedback to the developer if links are valid or not. The result of the composition shown in Figure 6.14 bottom is serialised to a file.

6.4.2 UCL Customisation

The advantage of using UCL in a composition system is that no integration effort is required. On the contrary, UCL does not contain and can not be extended with domain specific composition language constructs as dedicated composition languages (cf. [Hen09, Section 4.2]). Nevertheless, the UCL syntax—boxes for components and lines to connect them—is very intuitive for many composition systems. Building on this, the user experience can be further improved by customising the graphical syntax with dedicated icons for certain fragments or types of fragments.

This customisation does not change the core of the language—that is, the metamodel (cf. Figure 6.13) does not change but only the graphical syntax. Consequently, even if the concrete syntax changes, composition programs can still be executed without adjusting the language semantics. Such syntax adjustments can be done dynamically with the proper tool support. The UCL editor we provide in REUSEWARE allows the assignment of dedicated icons to fragments. As an example, Figure 6.17 shows the same composition program as Figure 6.14 with customised syntax.

To use UCL with customised syntax instead of defining new (domain-specific) composition languages proved applicable for applications in practice (one is presented in Chapter 9). This is a clear advantage, since reusing the one customisable language with its tool support relieves composition system developers from defining dedicated composition languages and tooling. In existing ISC composition languages, the universal parts of composition languages, which are the calls of basic operators, could only be used by ISC experts. With UCL, we defined a universal composition language that can be used by experts of a certain domain, since, despite of its universality, it hides low-level details of ISC.

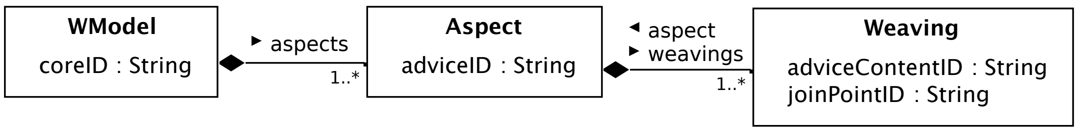


Figure 6.18: Metamodel of *SimpleWeave* composition language.

```
SYNTAXDEF simpleweave
FOR <http://www.reuseware.org/example/simpleweave>
START WModel

RULES{
    WModel::=   "core"  coreID['<','>']   "{" aspects+ "}"  ;
    Aspect::=   "weave" adviceID['<','>'] "{" weavings+ "}";
    Weaving ::=  adviceContentID[] "into" joinPointID[];
}
```

Listing 6.1: Text syntax of *SimpleWeave* composition language in EMFText CS (a variation of EBNF).

6.5 Composition Language Integration with REX$_{\text{CL}}$

Although we can use UCL directly as composition language in CB-MDSD, we showed in several examples that there are use cases for integrating arbitrary modelling languages as composition languages. For this, we introduce the *REuse eXtension language for Composition Language integration* (REX$_{\text{CL}}$). The language shares concepts with the component model configuration language REX$_{\text{CM}}$ (cf. Section 5.3). REX$_{\text{CM}}$ and REX$_{\text{CL}}$ together thus form the complete reuse extension (REX) language of U-ISC/Graph. As indicated in the beginning of this chapter, REX$_{\text{CL}}$ can be used by composition system developers to *integrate* DSMLs as composition languages into a CB-MDSD composition system, which in turn allows architects to develop composition programs in these DSMLs (this was illustrated in Figure 6.3).

REX$_{\text{CL}}$ specifications are also based on fragment collaborations. Instead of rules for fragment role binding (as specified with REX$_{\text{CM}}$) rules to establish collaborations between fragments (in terms of composition links that conform to composition associations) are defined with REX$_{\text{CL}}$. In a similar fashion to how variability types are identified in a model with REX$_{\text{CM}}$, UCL concepts are identified in a model with REX$_{\text{CL}}$. In contrast to a REX$_{\text{CM}}$ specification, a REX$_{\text{CL}}$ specification does not interpret a model as graph fragment, but as composition program.

Example 6.5. Imagine that we design a compact textual composition language for the FraCol ClassWeaving presented in earlier examples. We call the language *SimpleWeave* and define it in terms of an Ecore metamodel that consists of three metaclasses (Figure 6.18) and an EMFText syntax definition (Listing 6.1). What we need to do to use the language as composition language is to assign composition language semantics to the constructs of the language. This can be done with REX$_{\text{CL}}$. Then, instead of using UCL, we can defining a weaving in our own composition language as shown in Listing 6.2.

```
core <org/reuseware/example/uml/cores/FileSystem.uml> {
    weave <org/reuseware/example/uml/advices/ObserverAdvice.uml> {
        Subject into FSFolder
        Observer into FileSystem
    }
    weave <org/reuseware/example/uml/advices/ObserverAdvice.uml> {
        Observer into FileSystem
        Subject into FSFile
    }
}
```

Listing 6.2: A composition program defined in *SimpleWeave* with the same effect as the composition program in Figure 6.14.

6.5.1 REX_{CL} Concepts

In the following, we define the REX_{CL} language. Its syntax is similar to the one of REX_{CM} and, as REX_{CM}, it embeds OCL. The metamodel of REX_{CL} is shown in Figure 6.19. It reuses the abstract metclasses from the REX_{CM} metamodel (cf. Figure 5.27). Thus, it also contains relationships to the Ecore metamodel (cf. Figure 5.6) and our FraCol metamodel (cf. Figure 4.2). The concepts found in the metamodel are reflected in the textual syntax of REX_{CL} which we explain in the following on a schematic REX_{CL} specification shown in Listing 6.3. (The complete grammar REX_{CL} can be found in Section 7.2.)

The header in Lines 1–3 is similar as in the REX_{CM} syntax (cf. Section 5.3.2). To extract a composition program from different sources, it is important that also each composition program has a *Unique Composition Program Identifier (UCPI)*. Therefore, a REX_{CL} specification contains an *UCPI derivation rule* (Line 5) which is used to compute the UCPI of the composition program that is extracted when the specification is evaluated for a model. In this way, several models can contribute information to the same composition program. The rule has to compute an *ID* which is a special metaclass we introduce in Figure 6.20. An ID consists of a list of string segments and the ID metaclass offers a number of operations for modifying the segments. This allows a convenient derivation and modification of IDs with OCL. For OCL expressions in REX_{CL} (and also REX_{CM}) specifications, we make the UFI of the current model available as ID element in the variable `ufi`. Thus, expressions cannot only inspect elements of the model (via `self`), but also the model's UFI (via `ufi`).

Each REX_{CL} specification effectively defines rules that ensure that a certain fragment collaboration is established in the composition program identified by the UCPI. For that, it contains rules for *fragment instance role bindings* and *composition association bindings* explained in the following.

Lines 7–16 show a fragment instance role binding that ensures that a certain fragment instance exists in the identified composition program. The *roleName* identifies the role in the implemented FraCol. The `reference` keyword determines if the fragment instance should be a reference. The *eClass* specifies the `EClass` for which instances the rule applies. The optional condition can enforce further constraints on those instances. The *fragment*, *ufi*, and *target ufi* expressions (Lines 9–11) compute the name, UFI, and target UFI of the fragment instance, respectively. The target ufi expression is optional and only if it is set the `target` flag of the fragment instance is set to true. The fragment expression has to return a string and both ufi

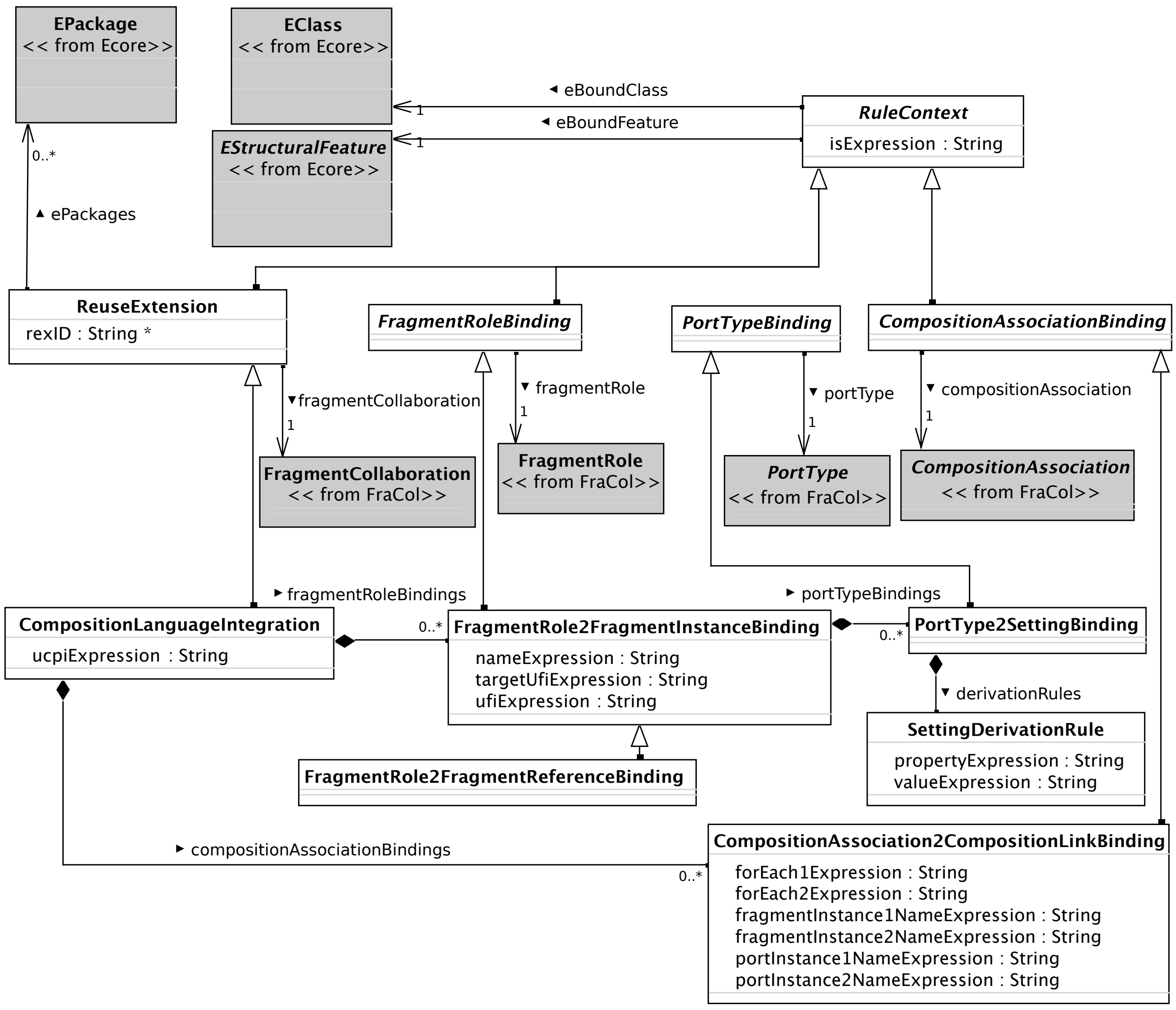


Figure 6.19: Composition language integration concepts.

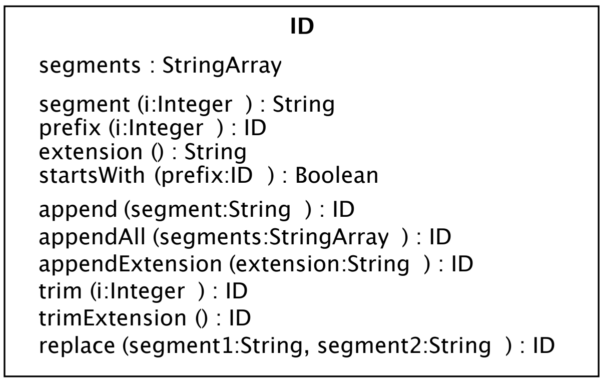

ID
segments : StringArray
segment (i:Integer) : String
prefix (i:Integer) : ID
extension () : String
startsWith (prefix:ID) : Boolean
append (segment:String) : ID
appendAll (segments:StringArray) : ID
appendExtension (extension:String) : ID
trim (i:Integer) : ID
trimExtension () : ID
replace (segment1:String, segment2:String) : ID

Figure 6.20: Metaclass for representing and manipulating IDs.

```
 1 compositionlanguage ID
 2 implements     fracolID
 3 epackages      <epackage> +
 4 rootclass      rootelementEClass if $OCLExpression$
 5 ucpi = $OCLExpression$ {
 6
 7   reference fragment role fragmentRoleName {
 8     eClass if $OCLExpression$ {
 9       fragment   = $OCLExpression$
10       ufi        = $OCLExpression$
11       target ufi = $OCLExpression$
12       port type portName {
13         $propertyOCLExpression$ = $valueOCLExpression$
14       }
15     }
16   }
17
18   association associationName {
19     eClass if $OCLExpression$ {
20       foreach $OCLExpression$
21       fragment = $OCLExpression$
22       port     = $OCLExpression$
23       -->
24       foreach $OCLExpression$
25       fragment = $OCLExpression$
26       port     = $OCLExpression$
27     }
28   }
29 }
```

Listing 6.3: Schematic REX$_{CL}$ specification.

expressions an ID. The rule ensures, that a fragment instance with the computed properties exists in the composition programs identified by the UCPI. In addition, settings of a port on the fragment with the name *portName* are derived by using a pair of OCL expressions (Line 13).

An association binding, shown in Lines 18–28, is used to ensure that a certain composition link exists. Again, *eClass* and its optional condition are used to identify the elements that trigger the association binding. Then, four rules, which are all expected to return a string, identify two fragments (*fragment name expressions*) and two ports (*port name expressions*) on these fragments. The association binding ensures that these two ports are linked with a composition link. If a linked port is static, the *port expressions* (Lines 22 and 26) can be omitted, since the name of the port is known from the static port type at the corresponding association end.

Optionally, the `foreach` expressions (Lines 20 and 24), which need to return a set of model elements, can be used to compute two sets of ports where each port of the first set is linked with each port of the second set.

Example 6.5.1. (1/2) To illustrate REX$_{CL}$, we continue the definition of the *SimpleWeave* composition language by defining composition semantics for the language with REX$_{CL}$ in Listing 6.4. We assign the following semantics to the three metaclases of *SimpleWeave*: `WModel` instantiates core fragments (Lines 7–13), `Aspect` instantiated advice fragments (Lines 15–20), and `Weaving` creates links between advice `Content` and core `JoinPoint` ports (Lines 22–30). When the expression defined in Lines 6–30 are interpreted, they extract UFIs of the core and

```
compositionlanguage org.reuseware.example.class_weaving.simpleweave
implements          org.reuseware.example.class_weaving
epackages           <http://www.reuseware.org/example/simpleweave>
rootclass           simpleweave::WModel
ucpi = $ufi.trimExtension().appendExtension('fc')$ {

  fragment role Core  {
    simpleweave::WModel {
      fragment   = $'CORE'$
      ufi        = $self.coreID.split('/')$
      target ufi = $ufi.trim(1).append('woven').append(ufi.segment(-1))$
    }
  }

  fragment role Advice {
    simpleweave::Aspect {
        fragment  = $self.adviceID$
        ufi       = $self.adviceID.split('/')$
      }
    }

  association weave {
    simpleweave::Weaving {
      fragment = $self.aspect.adviceID$
      port     = $self.adviceContentID$
      -->
      fragment = $'CORE'$
      port     = $self.joinPointID$
    }
  }
}
```

Listing 6.4: REX_{CL} specification of *SimpleWeave* composition language.

advice fragments as well as port names for the `JoinPoint` and `Content` ports from the `coreID`, `adviceID`, `adviceContentID`, and `joinPointID` attributes of *SimpleWeave* models.

Listing 6.2 shows a model defined in *SimpleWeave*. Interpreting it as composition program by using the REX_{CL} specification of Listing 6.4, derives the composition program shown in Figure 6.14. Compared to UCL, *SimpleWeave* introduces concepts specific to an aspect composition system (e.g., the distinction between core and aspect) and abstracts from details (e.g., a fixed target UFI is computed that cannot be changed by a user).

SimpleWeave can be combined with any component language, for which a REX_{CM} is defined that implements the FraCol ClassWeaving (cf. Listing 4.4), to a composition system that offers both a dedicated component model and a dedicated composition language for aspect weaving. A user of such a system does not need knowledge about U-ISC/Graph, since all its specifics are hidden behind the dedicated languages. So far, we defined two component models that can be combined with *SimpleWeave* to form an aspect composition system (UML in Listing 5.5 and Java in Listing 5.6). However, by providing a REX_{CM} specification that implements our FraCol ClassWeaving for another modelling language, *SimpleWeave* can be made usable for weaving aspects defined in that language.

This brings us back to the example in Section 3.1, where we defined a composition operator in COMPOST as Java method `weave()` (cf. Listing 3.3). Using the `weave` construct of *SimpleWeave* (metaclass `Weaving`) has the similar effect as the `weave()` operator. Using `weave`

```
compositionlanguage org.reuseware.lib.systems.participation.cl.odt
implements org.reuseware.lib.systems.participation.participation
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  fragment role Participant {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                          '_').concat(ufi.trimExtension().segment(-1))$
      ufi = $Sequence{'org','reuseware','lib','systems','participation','lib','Participant.'.concat(
                          variant)}$
      port type Name { $'name'$ = $mixed->at(1).getValue()$ }
    }
  }
  association Participation {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                          '_').concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'UseCase:'.concat(ufi.trimExtension().segment(-1))$
    }
  }
}
```

Listing 6.5: REX$_{CL}$ specification for OpenDocument as composition language that conforms to the FraCol Participation (cf. Listing 4.3).

in a *SimpleWeave* model is similar to calling `weave()` in a COMPOST composition program written in Java (cf. Listing 3.4). Therefore, in U-ISC/Graph, the complex composition operator concept has been replaced by the grouping of variation points in ports with REX$_{CM}$ and specification of composition languages in REX$_{CL}$. Through this, arbitrary Ecore-based languages can be integrated as composition languages into composition systems.

Example 6.5.1. (2/2) Finally, REX$_{CL}$ enables us to realise the extraction of composition information from models and collect it in a single composition program as required for the ticket shop example of Section 2.1. Listings 6.5 and 6.6 show two REX$_{CL}$ specifications that integrate OpenDocument and UML use cases as composition languages respectively. They implement the FraCols Participation and Exchange (Figures 4.4 and 4.7). Listing 6.5 defines how the Participation composition information is extracted from OpenDocument models. Listing 6.6 defines how the Exchange composition information is extracted from UML use case models.

6.5.2 REX$_{CL}$ Semantics

Above we explained the meaning of the REX$_{CL}$ concepts and showed how composition information is extracted from models and merged into a UCL composition program. The complete semantics of this composition program extraction are defined in in Appendix A.5.

```
compositionlanguage org.reuseware.lib.systems.exchange.cl.usecase_uml
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Value {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      ufi = $Sequence{'org','reuseware','lib','systems','exchange','lib','Value.'.concat(variant)}$
      port type Provider {
        $'value'$ = $body.split(' ')->at(1)$
        $'name'$  = $body.split(' ')->at(1).concat('_').concat(body.split(' ')->at(2))$
        $'ID'$    = $body.split(' ')->at(2)$
        $'inSet'$    = $body.contains('inSet')$
      }
    }
  }
  association Contribution {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
  association Provide {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      -->
      fragment=$'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(ufi.segment(-2))$
    }
  }
  fragment role Value {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      ufi = $Sequence{'org','reuseware','lib','systems','exchange','lib','Value.'.concat(variant)}$
      port type Consumer {
        $'value'$ = $body.split(' ')->at(1)$
        $'name'$  = $body.split(' ')->at(1).concat('_').concat(body.split(' ')->at(2))$
        $'ID'$    = $body.split(' ')->at(2)$
        $'inSet'$    = $body.contains('inSet')$
      }
    }
  }
  association Contribution {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
  association Consume {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      -->
      fragment=$'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(ufi.segment(-2))$
    }
  }
}
```

Listing 6.6: REX$_{\text{CL}}$ specification for UML use cases as composition language that conforms to the FraCol Exchange (cf. Listing 4.3).

6.6 Conclusion

Chapter Contributions This chapter contributes the following:

C1-4 **Basic invasive composition operators for graphs**
In this chapter, we defined the basic composition operators of ISC as graph rewritings. It is the first time that the basic composition operators have been defined in this form. It is not only a redefinition of the existing operators (which were defined informally for ISC in [Aßm03] and in a composition algebra for U-ISC/Tree in [Hen09]), but also an extension of them. This extension takes the requirements of the variability typing for graph fragments—the support for edge lists and attributes—into account. Furthermore, we introduced the new BasicCompose operator that automatically delegates to an appropriate operator.

C1-5 **Universal invasive composition language and algorithm**
This chapter introduced a set of basic composition language concepts for ISC—as an equivalent to having basic composition operators for the composition technique. Such a basic set of composition language concepts was not defined before. An invasive composition algorithm based on these concepts was introduced that works with declarative composition programs and automatically handles instantiation (i.e., copying) of fragments. This had to be done manually by developers in ISC before. So far, ISC relied on complex composition operators that were component language dependent. Our composition algorithm is language independent, because all language specifics are hidden behind composition interfaces which are configured with REX_{CM}. This contribution has the following sub-contributions:

- *Declarative universal composition language*
  This chapter introduced the first universal ISC composition language that abstracts from the basic composition operators but stays, thanks to the universal composition algorithm, universal. It therefore provides the first ISC composition language that is both (1) integrable into arbitrary ISC composition systems and (2) usable for developers who are not ISC experts.
- *Language for composition language integration*
  In this chapter we defined REX_{CL} to define rules to integrate arbitrary EMOF/Ecore-based languages into composition systems as composition languages. Such a language that is dedicated for defining composition languages for ISC systems was never defined before. The specification of composition language semantics in Embedded ISC [Hen09] required direct manipulation of a language's grammar and supported only the use of the same language as component and composition language.

Part Contributions Part I, which concludes with this chapter, has the following major contribution:

C1-6 **Declarative composition system specification formalism**
In combination, FraCols, REX_{CM}, and REX_{CL} form the first declarative formalism to specify complete invasive composition systems in which multiple component and composition languages can be chosen freely and combined. It is also the first formalism of this kind that relies on a standardised metalanguage (EMOF/Ecore) and does not require extension of language metamodels.

This concludes this part of the thesis that specified our composition technique U-ISC/Graph. The next part deals with using U-ISC/Graph for CB-MDSD and with that evaluates the practical relevance of U-ISC/Graph.

Part II

Component-Based Model-Driven Software Development with U-ISC/Graph

7 Reuseware Composition Framework

This chapter describes the Reuseware Composition Framework, which implements the concepts of U-ISC/Graph and thus can be used to develop composition systems with it.

This second part of the thesis deals with defining composition systems with U-ISC/Graph and using those for CB-MDSD. For that, we define and evaluate, in Chapters 8 and 9, two architectural styles for CB-MDSD called ModelSoC and ModelHiC. To use U-ISC/Graph in this practical context, we require tool support, which is introduced in this chapter.

We provide an implementation of U-ISC/Graph called Reuseware Composition Framework (short REUSEWARE), which we made available at `http://www.reuseware.org`. REUSEWARE is implemented in Java based on the Eclipse environment [Ecl06] and the Eclipse Modeling Framework (EMF) [SBPM09]. REUSEWARE effectively is a *generic CB-MDSD composition system* and therefore called a *framework*. The framework (i.e., the generic composition system) is instantiated by a composition system developer to a concrete CB-MDSD composition system. This is done by providing FraCols and REX definitions as defined in Part I.

Figure 7.1 gives an overview of the metamodels and tooling of REUSEWARE. The core of REUSEWARE is generated directly from the conceptual models of U-ISC/Graph defined in Part I and Appendix A. The EMF provides the EMOF implementation Ecore (cf. Section 5.1.1) and facilities to translate Ecore metamodels into Java code. Thus, this part of the implementation is directly derived from the metamodels for the different U-ISC/Graph aspects we presented in Part I. Story Diagrams (cf. Appendix A) can be translated into Java code as specified in [FNTZ00] by the Fujaba tool [NNZ00]. Thus, this part of the implementation is directly derived from the story diagrams defined in Appendix A.

On top of this core, we provide tooling for composition system developers and users. This tooling consists of textual editors—for composition system developers to write FraCols, REX$_{\mathrm{CM}}$, and REX$_{\mathrm{CL}}$ specifications—and of a graphical editor—for composition system users to define UCL composition programs. The textual editors were developed wit EMFText [HJK^{+}09] and the graphical editor was developed with GMF [Gro09].

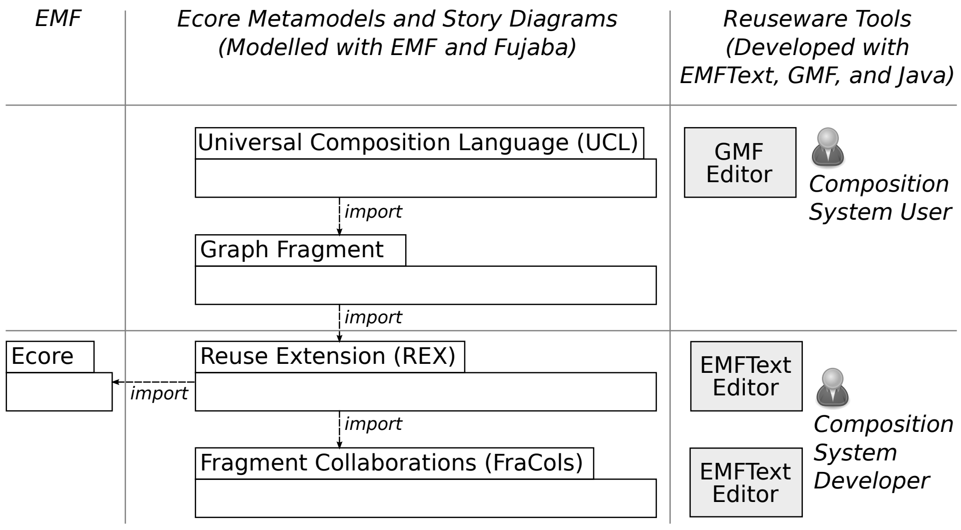

EMF
Ecore Metamodels and Story Diagrams (Modelled with EMF and Fujaba)
Reuseware Tools (Developed with EMFText, GMF, and Java)
Universal Composition Language (UCL)
import
Graph Fragment
import
Ecore
import
Reuse Extension (REX)
import
Fragment Collaborations (FraCols)
GMF Editor
Composition System User
EMFText Editor
EMFText Editor
Composition System Developer

Figure 7.1: Metamodels and tooling of Reuseware.

In the following, we describe the architecture of Reuseware and its integration with other modelling tools in Section 7.1 and give more details of the tools for developers (Section 7.2) and users (Section 7.3). In Section 7.4, we discuss extension possibilities for Reuseware.

7.1 Architecture of the Reuseware Composition Framework

In Part I, we introduced several software modelling concepts that are used by U-ISC/Graph. For each of these concepts, we selected one implementation that we used for developing Reuseware or that we integrated with Reuseware.

EMOF/Ecore (cf. Section 5.1) We use Ecore, the EMOF implementation in EMF, for metamodel definition. EMF was used for developing Reuseware and is also integrated with Reuseware. At development time, it was utilised to generate code from the metamodels for FraCols, REX, Graph Fragments, and UCL we define in Parts I (Figure 7.1 middle column). At runtime, EMF is used for loading, storing, and modifying models.

SDM (cf. Appendix A) Fujaba [NNZ00] is used to specify Ecore models and story diagrams at development time. Furthermore, it is used to generate code from story diagrams that integrates with the code generated by EMF. For that, Fujaba offers a special code generator [GBD07]. We utilised this code generator to generate code that implements the semantics of REX and UCL from the story diagrams we specified in Part I and Appendix A (Figure 7.1 middle column). Fujaba is not used at runtime and thus not integrated in Reuseware.

Concrete Syntax (cf. Section 5.1.2) We used EMFText [HJK$^+$09] and GMF [Gro09] to develop syntax for the languages introduced in Part I. From these syntax specifications, editors are generated (Figure 7.1 right column). EMFText is used at runtime for supporting textual languages (e.g., Java). The runtime part of GMF is also integrated in an optional extension of Reuseware for layout composition (cf. discussion in Section 12.1).

```
SYNTAXDEF fracol
FOR <http://www.reuseware.org/coconut/fracol>
START FragmentCollaboration
RULES {
  FragmentCollaboration ::= "fracol" fracolID[] ("." fracolID[])*
                            "{" (fragmentRoles | compositionAssociations)* "}";

  FragmentRole ::= "fragment" "role" name[] "{" portTypes* "}";

  StaticPortType  ::= "static"  "port" "type" name[] ";";
  DynamicPortType ::= "dynamic" "port" "type" name[] ";";

  Configuration ::= "configuring" "association" name[] "{"end1Role[]"."end1[] "-->" end2Role[]"."end2[]"}";
  Contribution  ::="contributing" "association" name[] "{"end1Role[]"."end1[] "-->" end2Role[]"."end2[]"}";
}
```

Listing 7.1: FraCols text syntax defined in EMFText CS (a variation of EBNF).

OCL (cf. Section 5.3.1) The EMF-based OCL interpreter (MDT-OCL) [Ecl10c] is used at runtime to interpret OCL expressions that are embedded in REX specifications. The interpreter is extended to support a richer string API (cf. Section 5.3.1) but otherwise it is a black-box integration. OCL was not used at development time.

Hand-written code (Java) At development time, the tool was completed by some hand-written code for user interfaces and integration into the Eclispe platform. This includes adjustments to the UCL editor generated with GMF to realise the syntax customisation described in Section 6.4. The main part of Reuseware, which corresponds to the conceptual models of the thesis, is generated as described above.

Potentially, all EMF-based tools (i.e., tools that understand Ecore metamodels) can integrate with Reuseware since they work on similar data structures. In Chapter 8 and 9, we mainly integrated with graphical and textual editors, but also integration with other modelling tools, for instance validation or transformation tools, is possible. This is an advantage gained by building on the EMOF-conformant Ecore and an important success criterion to transfer our approach into practice.

7.2 Tooling for Composition System Developers

Reuseware is a generic CB-MDSD composition system and thus can be configured by composition system developers with FraCols (Section 4.2), REX$_{\text{CM}}$ (Section 5.3), and REX$_{\text{CL}}$ (Section 6.5) specifications. For this, Reuseware provides text editors that support the syntax of these languages with features like syntax highlighting and auto-completion. These editors were generated with EMFText, a tool that was also partially developed in the research for this thesis. For this, the languages' text sytanxes were defined in the EBNF-like CS language of EMFText (cf. Section 5.1.2 and [HJK^{+}09] for CS language notation). These definitions for FraCols, REX$_{\text{CM}}$, and REX$_{\text{CL}}$ are shown in Listings 7.1, 7.2, and 7.3 respectively.

```
SYNTAXDEF rex
FOR <http://www.reuseware.org/coconut/reuseextension>
START ComponentModelConfiguration
RULES {
 ComponentModelConfiguration ::= "componentmodel" rexID[] ("." rexID[])*
      "implements" (fragmentCollaboration[] ".")* fragmentCollaboration[]
      "epackages"  ePackages['<','>']+
      "rootclass"  eBoundClass[] ("if" isExpression['$','$'])?
      "{" fragmentRoleBindings* "}";
  }
  FragmentRole2FragmentBinding ::= "fragment" "role" csFragmentRole[] ("if" isExpression['$','$'])?
                                                                      "{" portTypeBindings* "}";
  PortType2HeterogeneousPortBinding ::=       "port" "type" csPortType[] "{" derivationRules* "}";
  PortType2HomogenousPortBinding ::=   "homo" "port" "type" csPortType[] "{" derivationRules* "}";
  SlotDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "slot"
    ("if" isExpression['$','$'] )? "{" ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("mode"         "=" modeExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
      ("remove"       "=" removeExpression['$','$'])?
    "}";
  HookDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "hook"
    ("if" isExpression['$','$'] )? "{" ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("mode"         "=" modeExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
      ("remove"       "=" removeExpression['$','$'])?
    "}";
  AnchorDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "anchor"
    ("if" isExpression['$','$'] )? "{"  ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
    "}";
  PrototypeDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "prototype"
    ("if" isExpression['$','$'] )? "{" ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
    "}";
  ValueHookDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "value" "hook"
    ("if" isExpression['$','$'] )? "{" ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("mode"         "=" modeExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
      ("begin" "idx" "=" beginIndexExpression['$','$'])?
      ("end"   "idx" "=" endIndexExpression['$','$'])?
      ("list"  "idx" "=" endIndexExpression['$','$'])?
    "}";
  ValuePrototypeDerivationRule ::= eBoundClass[] ("." eBoundFeature[])? "is" "value" "prototype"
    ("if" isExpression['$','$'] )? "{" ("foreach" forEachExpression['$','$'])?
      ("foreach" forEachExpression['$','$'])?
      ("homo" "port" "=" homoPortNameExpression['$','$'])?
      ("port"         "=" portNameExpression['$','$'])?
      ("point"        "=" pointNameExpression['$','$'])?
      "value"         "=" valueExpression['$','$']
    "}";
}
```

Listing 7.2: $\mathrm{REX_{CM}}$ text syntax defined in EMFText CS (a variation of EBNF).

```
SYNTAXDEF rex
FOR <http://www.reuseware.org/coconut/reuseextension>
START CompositionLanguageIntegration
RULES {
  CompositionLanguageIntegration ::= "compositionlanguage" rexID[] ("." rexID[])*
      "implements" (fragmentCollaboration[] ".")* fragmentCollaboration[]
      "epackages"  ePackages['<','>']+
      "rootclass"  eBoundClass[] ("if" isExpression['$','$'])?
      "ucpi" "="   ucpiExpression['$','$'] !0
      "{" (fragmentRoleBindings | compositionAssociationBindings)* "}";
  }

  FragmentRole2FragmentInstanceBinding ::= "fragment" "role" csFragmentRole[] "{"
    eBoundClass[] ("if" isExpression['$','$'] )? "{"
      "fragment"      "=" nameExpression['$','$']
      "ufi"           "=" ufiExpression['$','$']
      ("target" "ufi" "=" targetUfiExpression['$','$'])?
      (portTypeBindings)*
    "}"
  "}";

  FragmentRole2FragmentReferenceBinding ::= "reference" "fragment" "role" csFragmentRole[] "{"
    eBoundClass[] ("if" isExpression['$','$'] )? "{"
      "fragment"      "=" nameExpression['$','$']
      "ufi"           "=" ufiExpression['$','$']
      ("target" "ufi" "=" targetUfiExpression['$','$'])?
      (portTypeBindings)*
    "}"
  "}";

  PortType2SettingBinding ::= "port" "type" csPortType[] "{" derivationRules* "}";

  SettingDerivationRule ::= propertyExpression['$','$'] "=" valueExpression['$','$'] ;

  CompositionAssociation2CompositionLinkBinding ::= "association" csCompositionAssociation[] "{"
    eBoundClass[] ("if" isExpression['$','$'] )? "{"
      ("foreach" forEach1Expression['$','$'])?
      "fragment" "=" fragmentInstance1NameExpression['$','$']
      "port"     "=" portInstance1NameExpression['$','$']
      "-->"
      ("foreach" forEach2Expression['$','$'])?
      "fragment" "=" fragmentInstance2NameExpression['$','$']
      "port"     "=" portInstance2NameExpression['$','$']
    "}"
  "}";
}
```

Listing 7.3: REX_{CL} text syntax defined in EMFText CS (a variation of EBNF).

With REX_{CM} and REX_{CL}, modelling languages defined with Ecore are integrated in Reuseware as component or composition languages. These can be existing languages, such as UML or Java, but also newly defined DSMLs, as demonstrated with the *SimpleWeave* language in Example 6.5. In the latter case, existing metamodelling tools can be used for language design. The only requirement is that the language metamodel is defined with Ecore. Since there are more and more tools emerging to develop Ecore-based languages and tooling for them, Reuseware integrates well into the current tool ecosystem for language development in MDSD. We discussed some of these tools in Section 5.1.2.

```
fracol org.reuseware.lib.systems.default {
  fragment role Default {
    dynamic port type Config;
    dynamic port type Contrib;
    dynamic port type Rec;
  }

  contributing association Contribution {
    Default.Contrib --> Default.Rec
  }

  configuring association Configuration {
    Default.Config --> Default.Config
  }
}
```

Listing 7.4: Default FraCol.

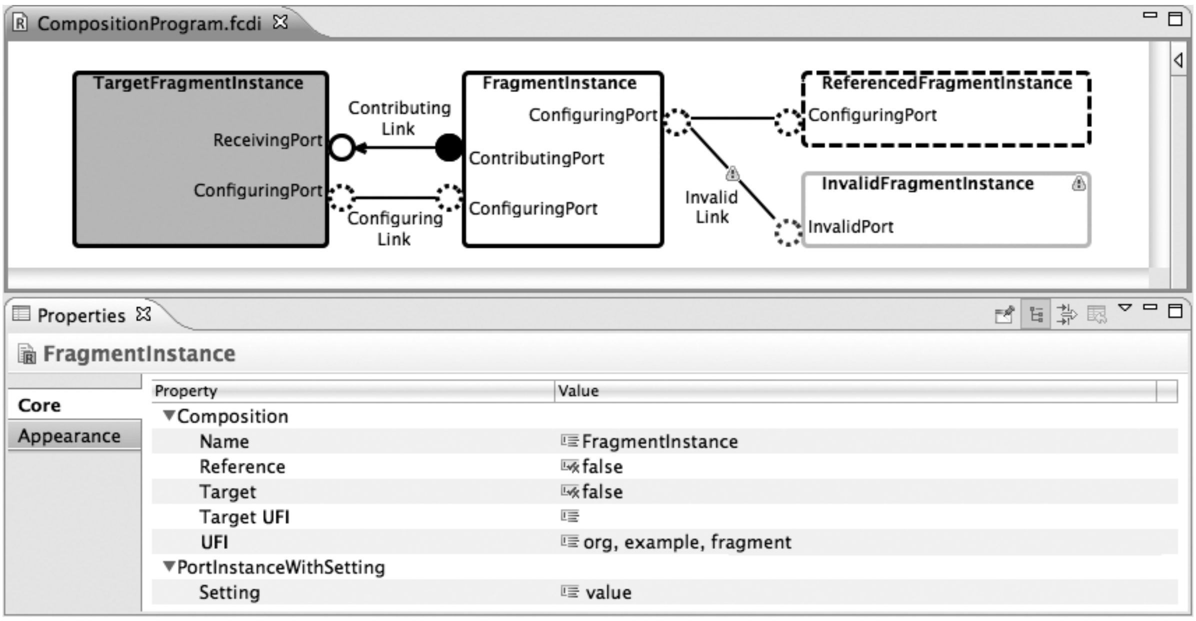
To improve composition system development for simple cases, REUSEWARE provides a *default* FraCol shown in Listing 7.4. The default FraCol defines fragment roles with dynamic port types in a way that all kinds of composition interfaces can be dynamically constructed by rules defined in a REX_{CM}. Thus, for simple composition systems, developers do not need to define their own FraCols but can reuse the default one. When UCL is reused as well, developers only need to define one REX_{CM} to obtain a composition system for a DSML.

7.3 Tooling for Composition System Users

A the composition system user can perform two kinds of activities: defining graph fragments and defining composition programs. Graph fragments can be modelled with existing and known model editors. REUSEWARE only delivers the tooling to compute the composition interfaces for these models. (This tooling is generated from the story diagrams of Appendix A.2.)

Composition programs are either directly defined with UCL, or, in cases other modelling languages were integrated with REX_{CL}, in the existing editors for those languages. For direct definition in UCL, REUSEWARE offers a graphical UCL editor that is oriented at the UCL notation we introduced in Section 6.2 and supports the syntax customisation for UCL discussed in Section 6.4.2. The editor with the graphical syntax is depicted in Figure 7.2. Fragment instances, port instances, and composition links use the same notation as introduced in Figure 6.12. In addition, a fragment instance that is set to `reference` is depicted by a dashed outline. A fragment can also be set to `target`, depicted by a grey background, which means that a composed copy of the fragment that is obtained after execution of the composition is stored into the repository. The `targetUFI` attribute is used to set the UFI for this new composed fragment. Value settings can be modified in a tabular properties view, individually for each fragment (shown in the bottom of Figure 6.12). Here, the settings that exist are listed grouped by the port instances they belong to.

The UCL editor also reports problems that are identified during link matching (cf. Section 6.3). If a fragment instance is invalid (i.e., if the fragment it represents does not does not exist), the box has a grey outline and is marked with a warning symbol. Invalid ports instances (i.e., port instances of ports that do not exist) have a grey dashed outline. If a link



Property	Value
▼Composition	
Name	FragmentInstance
Reference	false
Target	false
Target UFI	
UFI	org, example, fragment
▼PortInstanceWithSetting	
Setting	value

Figure 7.2: Graphical syntax of UCL shown in the UCL editor.

is invalid, it is marked with a warning symbol. A link is invalid if the connected port instances are invalid or if the matching of the link fails.

To support the integration of other composition languages with REX$_{\text{CL}}$, REUSEWARE offers the tooling to extract composition information from models defined in other editors and integrate them in derived UCL composition programs. (This tooling is generated from the story diagrams of Appendix A.5.) In this case, a user might be completely unaware of the fact that he or she uses U-ISC/Graph concepts for modelling, since REUSEWARE can work completely in the background. It depends on the concrete modelling task how much of a composition system, and hence of the REUSEWARE tooling, is exposed to the user—and it is up to the composition system developer to control this.

7.4 Extension Points

As explained above, the tooling for composition system users can be flexibly extended by integrating arbitrary modelling languages and their editors. On the other side, the tooling for composition system developers is fixed to the FraCols, REX$_{\text{CM}}$, and REX$_{\text{CL}}$ languages as defined in Part I. However, there are possible alternatives, in particular to REX$_{\text{CM}}$ and REX$_{\text{CL}}$, which we discuss in the following.

7.4.1 Alternative Expression Languages

REX$_{\text{CM}}$ and REX$_{\text{CL}}$ make heavy use of OCL to formulate conditions and queries over models. OCL expressions are defined as plain strings in the specifications (cf. metamodels in Figures 5.27 and 6.19) and are passed to an external tool, the MDT OCL interpreter [Ecl10c],

```java
public interface Evaluator {
    boolean eval(List<String> ID, EObject       context, String expression);
    boolean eval(List<String> ID, List<EObject> context, String expression);

    String        derive    (List<String> ID, EObject context, String expression);

    int           deriveInt (List<String> ID, EObject context, String expression);

    List<String>  deriveID  (List<String> ID, EObject context, String expression);

    List<EObject> deriveElementList(List<String> ID, EObject context, String expression);
}
```

Listing 7.5: The Evaluator interface.

which parses the string, evaluates the parsed expression, and returns the result of the evaluation. The external tool is hidden behind the Evaluator interface that is shown in Listing 7.5. This interface is used in the operations that interpret REX_{CM} or REX_{CL} specifications defined in Appendices A.2 and A.5.

If we look at the interface, we can observe that it does not depend on OCL or U-ISC/Graph. The types used are native Java types and `EObject`. The expression is passed to the tool unparsed as string. Consequently, there is nothing that hinders the implementation of the interface for a tool that can parse and evaluate expressions formulated in another language than OCL. Examples of such languages from the modelling domain are found in the Epsilon language family [Kol08]. Also languages from other domains such as SQL or web query languages could be used. In fact, any language that allows for querying a graph structure can be used. This includes all kinds of rule-based query languages [BEE+07].

Reuseware offers an extension point, where new Evaluator implementations can be plugged in. Each Evaluator has to be associated with a prefix. In REX_{CM} and REX_{CL}, we allow a prefix to be appended to each expression string separated by a colon. By default, Reuseware supports the `ocl` prefix for plain standard compliant OCL and the `ocl+` prefix for OCL with extended string API (cf. Section 5.3.1). If no prefix is given, `ocl+` is taken as default. When evaluating an expression, Reuseware selects an Evaluator based on the prefix, strips the prefix from the expression string, and calls the Evaluator. For example, in Listing 5.5, the expression `self.name.startsWith('advice')` (Line 24) can also be written as `ocl+:self.name.startsWith('advice')`. Therefore, any expression language might be integrated, if an interpreter exists (or can be written) that can be adapted to the Evaluator interface. Even languages that are limited in their expressiveness can be utilised. Since the prefix notation allows for selecting the expression language for each expression individually, a developer can always fall back to OCL for cases where another language is not expressive enough.

Also graphical query languages, as the matching part of story diagrams (cf. Appendix A), could be used. However, the graphical queries cannot be embedded as strings. For this, a mechanism is needed that allows for expressions to be specified externally and referenced from inside a REX_{CM} specification. Such a mechanism could well be implemented in an Evaluator that does not evaluate the expression string directly, but interprets it as a reference to an externally defined expression. Having such a mechanism built into Reuseware in general

```java
public interface CompositionInterfaceComputer {
    boolean canCompute(Fragment fragment);

    Fragment compute(Fragment fragment);
}
```

Listing 7.6: The CompositionInterfaceComputer interface.

```java
public interface CompositionProgramExtractor {
    boolean canExtract(Fragment fragment);

    DerivedCompositionProgram extract(Fragment fragment);
}
```

Listing 7.7: The CompositionProgramExtractor interface.

however, would also be useful since it would allow us to define libraries of queries that can then be reused at different positions (e.g., in Listing 5.5 there are several expressions that reoccur). This should be investigated in the future.

7.4.2 Alternative Composition Interface Calculation

Apart from varying the expression part of REX_{CM}, one can also compute the composition interface of a graph fragment by other means. In fact, the `computeCompositionInterface()` operation (defined in Apendix A.2) can be regarded as a model transformation that has an arbitrary model as input and a fixed type of output model—an instance of the fragment metamodel of Figure 5.21. There are two viewpoints on this from the model transformation perspective. First, `computeCompositionInterface()` can be seen as a transformation defined with SDM that takes a REX_{CM} specification as additional input model. Second, REX_{CM} itself can be seen as domain-specific model transformation language that takes some model as input and produces an instance of the fragment metamodel as output.

Generic model transformation languages such as QVT [OMG08] allow for input and output models of arbitrary metamodels. Thus, one can use QVT or another model transformation language to write a transformation computing the composition interface of some model. Compared to using REX_{CM}, the author of such a transformation requires detailed knowledge of the fragment metamodel (cf. Figure 5.21). Since such a transformation can also take additional input models, it is also possible to define alternative component model configuration languages. Because all our conceptual models are Ecore metamodels, any model transformation technology that works with Ecore can be used for that.

To support this from the tooling side, Reuseware offers an extension point to override the `computeCompositionInterface()` by another CompositionInterfaceComputer that implements the interface shown in Listing 7.6. An implementation of the method `compute()` can call transformation code or a transformation engine with a transformation script. The method `canCompute()` decides for a given fragment if the CompositionInterfaceComputer should be used instead of `Fragment.computeCompositionInterface()`. This decision can be based on the types of mode elements in the fragment or the UFI of the fragment.

7.4.3 Alternative Composition Program Extraction

Similar to composition interface computation, one can also perform composition program extraction, usually defined with REX_{CL}, in alternative ways, since composition program extraction can also be regarded as model transformation from an arbitrary input model to an instance of the UCL metamodel of Figure 6.13. For this, Reuseware supports overriding of the `DeriveCompositionProgram.extractCompositionProgram()` operation (defined in Appendix A.5) by implementing the CompositionProgramExtractor interface shown in Listing 7.7. This works similar as using the CompositionInterfaceComputer interface from Listing 7.6. We experimented with alternative CompositionProgramExtractors in [JZF^{+}09], which were realised with model transformations defined in languages of the Epsilon platform [Kol08]. This was done together with developers of Epsilon and is further discussed in Section 12.4. More integration possibilities with other modelling technologies should be addressed in future research.

7.5 Conclusion

Contribution The contribution of this chapter is the Reuseware Composition Framework that is an implementation of U-ISC/Graph and the generic CB-MDSD composition system.

C2-1 **Universal model composition framework**
We provide a complete implementation of U-ISC/Graph and the generic CB-MDSD composition system presented in Part I that is based on EMF and Ecore. Since EMF is a popular open-source modelling framework and Ecore is a metamodelling languages aligned with EMOF, our implementation integrates well into the current MDSD tool ecosystem. The implementation is the base for evaluating U-ISC/Graph in practical MDSD.

Reuseware is used in the following Chapters 8 and 9 to realise two different model-driven architectures with CB-MDSD composition systems.

8 CB-MDSD with Multi-Dimensional Separation of Concerns (ModelSoC)

In this chapter we introduce ModelSoC, which is an architectural style for CB-MDSD with multiple DSMLs. We realise ModelSoC with U-ISC/Graph and evaluate it, and with it U-ISC/Graph, on a an extension of the example presented in Section 2.1. This chapter is an extended version of the following publication:

- Jendrik Johannes, and Uwe Aßmann. *Concern-based (de)composition of Model-Driven Software Development Processes.* In Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010) – Part II, volume 6395 of LNCS, pages 47–62. Springer, October 2010.

In this chapter, we present the first of two component-based architectural styles for MDSD. The style supports a specifc type of component-based development which is *multi-dimensional separations of concens* [OT00]. Hence, the style is called ModelSoC[1]. Multi-dimensional separations of concerns follows the assumption that a system needs to be decomposable in different concern dimensions in parallel, although different concern dimensions make use of different modularity concepts. An example of two different concern dimensions are the decomposition of a system into use cases or into classes [JN04].

In this sense, each model-driven architecture with multiple DSMLs inherently performs separation of concerns in multiple dimensions, since each DSML defines another abstract viewpoint on the system with its own modularity concept(s) and therefore its own concern dimension(s). When model components originating from different DSMLs are composed, the concern dimensions meet. At that point it still has to be known, how the model components map to concern dimensions to support a simultaneous separation of concerns along the different dimensions. In contrast, for a model-driven architecture that uses only a single DSML, one concern-dimension

[1]Component-based model-driven architectures with multi-dimensional separation of concerns

may suffice, which allows for a strict hierarchical decomposition. We discuss a second architectural style based on hierarchical decomposition, called ModelHiC, in Chapter 9.

ModelSoC is an alternative style for designing model-driven architectures, which, as the OMG's transformation-based style (cf. Figure 1.1a), includes multiple modelling languages for modelling a system from different viewpoints and on different abstraction levels. ModelSoC defines a conceptual model for designing such architectures, which is an extension of the *hyperspace model* for multi-dimensional separation of concern defined by Ossher and Tarr [OT00]. As we will illustrate, the hyperspace model is well suited as a base for ModelSoC, since it (a) explicitly supports different dimensions for decomposition, which is needed because in different types of models information is decomposed along different dimensions and (b) is independent of a concrete language (i.e., not limited to e.g. Java) or a concrete language paradigm (i.e., not limited to e.g. object-oriented languages). We used ModelSoC to fully realise the *ticket shop* example introduced in Section 2.1.

The contribution of this chapter is thus twofold. First, we introduce ModelSoC which includes an extension of the hyperspace model that can handle replication of information in different formats and usage of DSMLs for composing information (Section 8.1). Second, we show how ModelSoC is realised with U-ISC/Graph (Section 8.2) and evaluated both by realising the example with Reuseware (Section 8.3).

8.1 Conceptual Model for ModelSoC

In this section, we introduce ModelSoC: an architectural sytyle for MDSD based on multi-dimensional separation of concerns. For this, we define and use an extension of the hyperspace model for separation of concerns of Ossher and Tarr [OT00].

The hyperspace model supports the decomposition of a system into *concerns* along several *concern dimensions*. These span up an n-dimensional space—a *hyperspace*—where n is the number of utilised concern dimensions. Implementation artefacts are perceived as consisting of *units*[2]. Each unit realises at most one concern in each concern dimension which is defined in a *concern mapping* that maps units to concerns. Units are composed into *hyperslices* and *hypermodules*. A hyperslice composes all units belonging to one concern. Hypermodules compose several hyperslices to a complete software module or program. The hyperspace model leaves it open, what exactly can be treated as unit, how concern mappings and hypermodules are specified, and how the actual composition of a system from units is performed. The implementation Ossher and Tarr provide is Hyper/J [OT00], which supports different types of Java fragments as units (e.g., statements). It contains a dedicated language for concern mapping as well as a language for hyperslice and hypermodule definitions. Such definitions do not only enumerate concerns to compose, but also include calls to operators for composing Java units by bytecode weaving.

Since the conceptual hyperspace model is independent of an implementation language, it can be used to separate concerns of models in a model-driven architecture as the one shown in Section 2.1. As concern, we regard a piece of information about the system which does not require further decomposition. Examples of concerns are: (a) *Customer participates in Book*

[2]In our case, a unit is always a graph fragment.

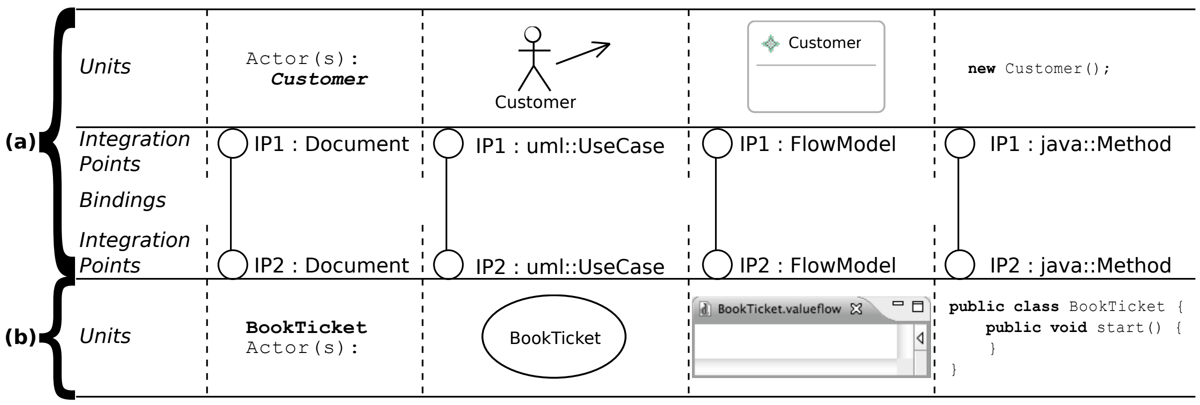


Figure 8.1: Multi-format units: (a) *Customer participates in Book Ticket* (b) *Book Ticket.*

Ticket, (b) *Bank participates in Book Ticket*, and (c) *Account is exchanged between Customer and Bank.* Concerns that follow the same mudularity concept belong to the same concern dimension. For instance, a new actor is composed into a use case by adding it to a use case model. This works similar for all actors, thus both concerns (a) and (b) belong to the concern dimension *Participation.* On the contrary, a new business value exchange is added by stating for a business value, which actor owns it before and which owns it after use case execution (cf. *value added invariant* in Section 2.1). Thus, (c) follows different (de)composition rules and belongs to another concern dimension (*Exchange*).

We identified three properties of MDSD that are difficult to map to the hyperspace model as it is, because they are either not considered by the model or further refine parts of the model that were left open [OT00]. First (Section 8.1.1), the hyperspace model does not consider that the same information may be present in multiple formats (e.g., same class in UML and Java). Second (Section 8.1.2), automated transformation of information is not covered by the hyperspace model. Third (Section 8.1.3), a refinement of the hyperspace model is that we want to forbid fixed concern mapping or hypermodule specification languages (as it is the case in Hyper/J). This is, because languages that are included in a model-driven architecture should be chosen based on the needs of the system's domain, which would not be given if a technology enforces the inclusion of a predefined mapping or module language. Therefore, ModelSoC introduces the following three concepts as extensions and refinements of the hyperspace model.

8.1.1 Multi-Format Units: Realisation of Concerns in Different Formats

A fragment of a model that represents a certain concern is a *unit* (in the sense of [OT00]). Since there may be different viewpoints on the same concern in MDSD (e.g., in Figures 2.4 and 2.5, information about actors and use cases is present in multiple views—OpenDocument, UML use cases, and Java) there can be several units representing the same concern in different formats. We introduce the *multi-format unit* concept that bundles such units. As an example, consider Figure 8.1 that shows the multi-format unit that realises the concerns *Book Ticket* and *Customer participates in Book Ticket.* To obtain a view that shows certain concerns, one has to select a viewpoint (e.g., UML use case) and if the format of the viewpoint is supported by all

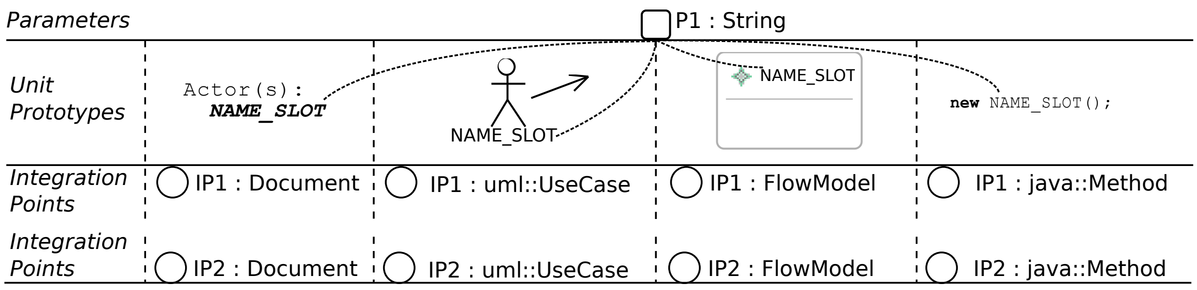


Figure 8.2: Multi-format unit prototype for *Participation* concern dimension.

multi-format units that realises the corresponding concerns, the view can be composed. Hence, each multi-format unit supports a set of viewpoints, but not each multi-format unit needs to support all viewpoints used in an model-driven architecture. The viewpoint of the final system (Java in the example), is most likely supported by all multi-format units. If support for a new viewpoint is needed for a multi-format unit, it can be added to it without altering the existing units (i.e., existing viewpoints) in the multi-format unit (see also Section 8.1.2 below).

A multi-format unit offers *integration points* and *bindings* between them to guide concern composition. Each unit has to support the integration points. Each unit offers its own version of the points typed by metaclasses of the metamodel for the unit (cf. Figure 8.1). Thus, integration points can be defined on the metamodel of a unit (cf. Section 8.1.3 below). Since a unit may need integration along several concern dimensions, the set of integration points is not fixed but can be extended, when a new concern dimension needs to be supported. For example, the concern *Book Ticket* in the concern dimension *UseCase* (Figure 8.1b) can exist on its own without the need for integration points because the concern dimension *UseCase* is independent of other concern dimensions. The concern *Customer participates in Book Ticket* in the concern dimension *Participation* (Figure 8.1a) requires integration with *Book Ticket* since the *Participation* depends on the *UseCase* dimension. Therefore, the multi-format unit for *Customer participates in Book Ticket* defines integration points for its own units and the units of *Book Ticket* and binds them.

8.1.2 Multi-Format Unit Prototypes: Automating Unit Creation

It is not practical to create each unit of a multi-format unit manually. Rather, following the MDSD idea, units holding the same information should be created automatically. We note that different concerns of the same dimension have similar structure. For example, the multi-format unit realising the concern *Bank participates in Book Ticket* looks similar as the multi-format unit of *Customer participates in Book Ticket* shown in Figure 8.1a (only the string `Customer` should be exchanged for `Bank` in each unit).

We can use this similarity to abstract a multi-format unit to a *multi-format unit prototype* that defines a common structure for all concerns of one concern dimension. For this, we create one *unit prototype* for each format supported by the multi-format unit prototype. A unit prototype is a small template, that offers parameters for the parts that differ between units of the same concern dimension.

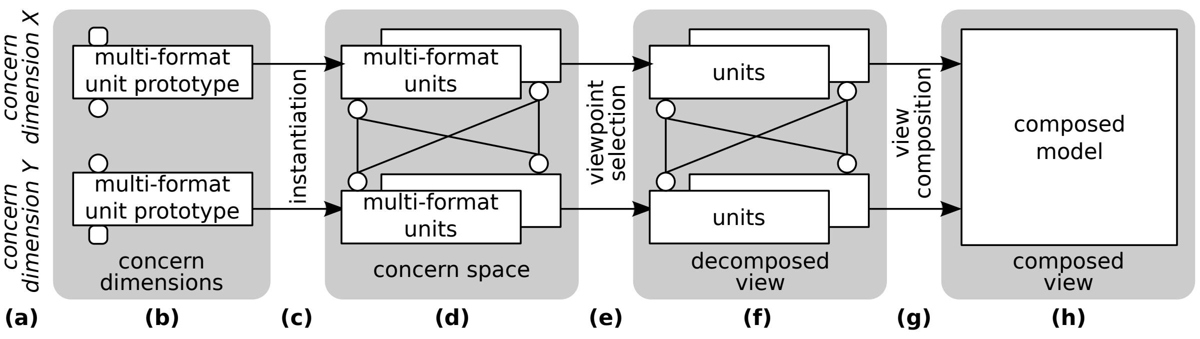


Figure 8.3: The concern management system of ModelSoC.

Figure 8.2 shows the multi-format unit prototype for the *Participation* concern dimension. It has one parameter *P1* for the actor name. A multi-format unit prototype can be instantiated to a multi-format unit by binding each parameter with a value (e.g., *P1* can be bound to `Customer` in Figure 8.2 to obtain Figure 8.1a) and integration points with each other. Integration points exist in a multi-format unit prototype, but not the concrete bindings, because these integrate individual concerns. Since the parameter *P1* is of the primitive type string, it is similar for all units. A parameter may also have a complex type that can differ for different views. In that case, different versions of a value are needed to bind the parameter. Integration points are always individually bound for each unit.

8.1.3 Meta-Level Concern Mappings and Compositions

We call the system supporting development with ModelSoC *concern management system*, depicted in Figure 8.3. The figure illustrates that each concern dimension (a) has a multi-format unit prototype (b). The instantiation of unit prototypes (c), which includes binding of parameters and integration points between individual concerns, spans up the concern space (d). Having this space available, a viewpoint can be selected (e) which reduces multi-format units to normal units (f). By interpreting the bindings, these units can be composed (g) to an integrated view in the selected viewpoint (h). Such a view corresponds to a hypermodule in the sense of [OT00].

Once the concern space has been established, steps (d) to (h) can be performed by a universal composition technology for any concern management system. Steps (a), (b), and (c) however require individual configuration for each model-driven architecture. Concretely, mechanisms are required to (a) define concern dimensions as well as integration points and parameters of the concerns in the dimensions, (b) define multi-format unit prototypes with integration points and parameters, and (c) define how parameter and integration point binding information is extracted.

(a) Concern dimensions, concern parameters, and concern integration points can be defined independently of models and metamodels for a model-driven architecture.

(b) Multi-format unit prototypes can be defined by modelling each unit prototype as a model fragment in the corresponding modelling language using an existing model editor (cf. Figure 8.2). Parameters and integration points can be specified for each unit prototype based on the prototype's metamodel. For example, the rules for *IP1* must state that in the UML use case format an actor is connected to a use case by adding it to the use case's `uml::Package` and for the Java format by adding the actor instantiation statement to the `java::Method` realising the use case execution. Rules for the parameter *P1* must state that `uml::Actor.name` represents the parameter in the UML format and `java::Variable.name` represents it in the Java format. These rules effectively define the concern mappings. By assigning the model fragments and rules that make up a multi-format unit prototype to the corresponding concern dimension defined in (a), we know to which dimension the instances of the unit prototype belong.

(c) Concern composition information is available in the user-defined models (e.g., in the textual use case description and the annotated UML use case diagram in Figure 2.6). In the example, the information that *Customer*, *Bank*, and *Clerk* participate in *BookTicket* is given in the BookTicket use case description and that *Hall* participates in *BookTicket* is given in the Hall UML use case model. This information can be extracted by rules based on the metamodels of the languages used. For instance, one rule must specify that each textual use case description, defined in OpenDocument format, instantiates the multi-format unit of the *UseCase* concern dimension and parameterises it with the name of the document. Furthermore, another rule must state that each mention of an actor in the document instantiates the multi-format unit prototype of the *Participation* dimension (cf. Figure 8.2) and composes it with the corresponding use case. This effectively forms a composition program for hypermodules. By assigning the rules for extracting the concern instantiation and composition information to a concern dimension defined in (a), we know which multi-format unit prototype to instantiate, which integration points to address, and which parameters to fill with the extracted information.

8.2 ModelSoC implemented with U-ISC/Graph

In the following, we use U-ISC/Graph to define a concern management system of an architecture based on ModelSoC. This is done by configuring the generic CB-MDSD composition system of U-ISC/Graph to define (cf. Section 8.1.3): (a) concern dimensions with integration points and parameters, (b) multi-format unit prototypes and (c) concern composition rule extraction. Reuseware, as our U-ISC/Graph implementation, then acts as concern management system for the architecture that can be used inside Eclipse in combination with Eclipse-based model editors. Figure 8.4 shows the ModelSoC configuration of the generic CB-MDSD composition system. All flexibility given for configuring the system is utilised. In the centre is an arbitrary large set of FraCols where each is implemented by potentially multiple REX_{CM} and REX_{CL} specifications. This maps to the conceptual model of ModelSoC as follows:

(a) We use FraCols (cf. Chapter 4) to define concern dimensions. In fact, we already defined some of the FraCols, which now represent concern dimensions, in Part I. For example, the FraCol for the *Participation* concern dimension was defined in Chapter 4 in Listing 4.2. Fragment roles are used to model the types of concerns of a dimension. In the FraCol Participation, these are `Participant` (which can be an actor) and `Collaboration` (which can be a use case). The port types of the fragment roles are used to declare integration points (`IP1` and `IP2` in Figure 8.2 that corresponds to Contrib and Rec in the FraCol)

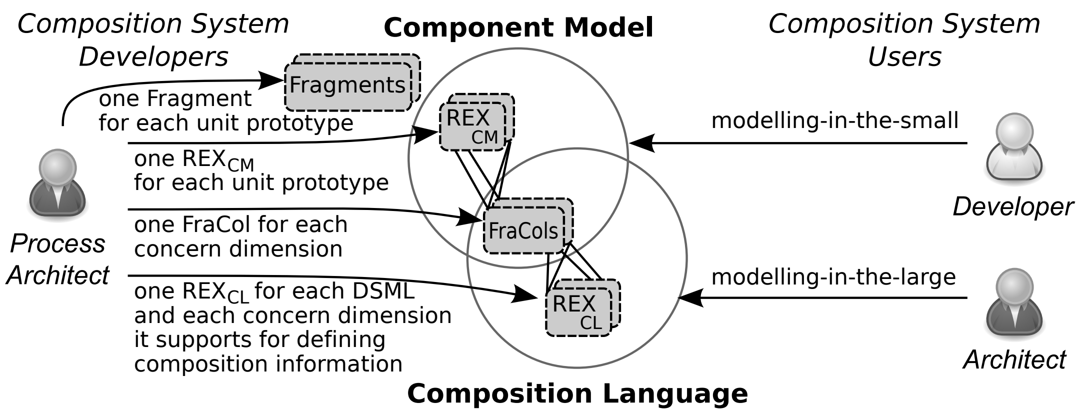

Composition System Developers
Component Model
Composition System Users
Fragments
one Fragment for each unit prototype
REX CM
one REX$_{CM}$ for each unit prototype
modelling-in-the-small
Developer
Process Architect
one FraCol for each concern dimension
FraCols
one REX$_{CL}$ for each DSML and each concern dimension it supports for defining composition information
REX CL
modelling-in-the-large
Architect
Composition Language

Figure 8.4: ModelSoC configuration of the generic CB-MDSD composition system.

and parameters (`P1` in Figure 8.2 that corresponds to Name in the FraCol). Composition associations are consequently used to define which integration points can be connected (Participation between Contrib and Rec in the FraCol).

(b) Multi-format unit prototypes are defined by a set of graph fragments with the help of REX_{CM} specifications (cf. Section 5.3). We define each unit prototype in its languages using a suitable existing model editor and then use REX_{CM} specifications to identify parameters and integration points and relate them to a FraCol and therewith to a concern dimension. The REX_{CM} specifications for UML and Java in the *Participation* dimension were already defined in Chapter 5 in Listings 5.8. and 5.9. When these specifications are applied to the multi-format unit prototypes of the *Participation* dimension (cf. Figure 8.2) and *UseCase* dimension, the parameters and integration points are identified.

(c) While REX_{CM} specifications define where fragments can be integrated, REX_{CL} specifications help with defining which fragments are integrated (cf. Section 6.5). Thus, we use REX_{CL} specifications to treat modelling languages in as composition languages to extract composition programs from models that contain the information which fragments are parameterised and composed. A REX_{CL} to perform this extraction for the Participation dimension from textual use case descriptions defined in OpenDocument was given in Chapter 6 in Listing 6.5. There, we defined that for each part of a textual use case description marked with the `OpenDocument::SpanType` *Actor*, a new actor fragment is instantiated with the `P1` parameter (the Name port type) bound to the name of the actor (extracted from the document using the OCL query `self.mixed->at(1).getValue()` in Line 13). Furthermore, we define that the binding of `IP1` and `IP2` (port types Rec and Contrib) is performed between the correct `Participants` and `Collaborations` by extracting the corresponding actor and use case names from the model and its UFI (Lines 18–20).

With FraCols, REX_{CM}, and REX_{CL} specifications given for all concern dimensions and viewpoints of a model-driven architecture, Reuseware acts as the concern management system (cf. Figure 8.3) for that architecture. For this, Reuseware interprets the specifications to extract a composition program that represents the complete concern space (d) by showing concerns and relations between them as parameterised and linked unit prototypes. Thus, the

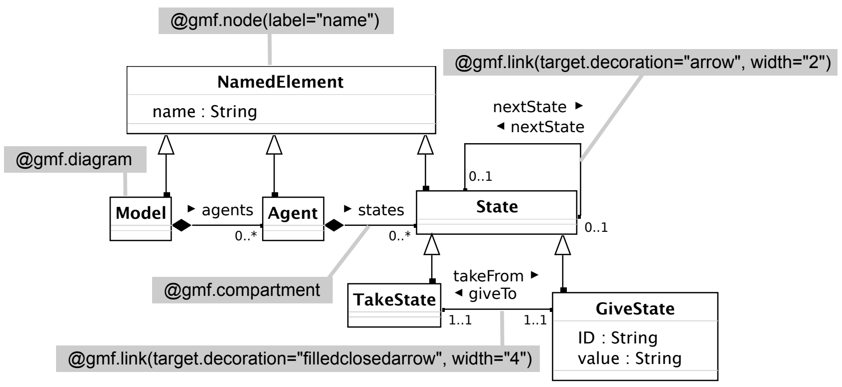


Figure 8.5: Ecore metamodel of ValueFlow with EuGENia annotations for graphical syntax.

concern space can be visualised graphically in the composition program editor for UCL (cf. Section 7.3). Since the composition is then executed automatically using the matching and composition algorithms of U-ISC/Graph (cf. Chapter 6), no further configuration is required for (e) to (h). Reuseware automatically executes the composition for all supported viewpoints by creating a composed model for each. The developer can decide at which view to look by opening a composed model in an editor of his or her choice.

8.3 Evaluation of ModelSoC

To evaluate that ModelSoC and U-ISC/Graph can be used to define component-based model-driven architectures with better separation of concerns than transformation-based architectures, we first (Section 8.3.1) specified the model-driven architecture introduced in Section 2.1 in Reuseware and used it to develop a first version of the ticket shop system with the features *book ticket* and *change seat* (Section 8.3.2). Afterwards (Section 8.3.3), we extended the architecture with a new viewpoint and concern dimension for security and used that to define security properties of the ticket shop system without changing the models defined before in other viewpoints.

8.3.1 Model-Driven Architecture Definition

Our model-driven architecture supports five different viewpoints. In Section 2.1 and other examples of Part I, we already used three of these viewpoints which are use case descriptions in OpenDocument, UML use case diagrams with business value annotations, and Java code.

As a fourth viewpoint, we use a graphical DSML—called ValueFlow—to define the order in which business values are exchanged between actors, since this is not known from the annotated UML use cases (which only define which business values are exchanged but not in which order). The Ecore metamodel of ValueFlow is shown in Figure 8.5. The graphical syntax for ValueFlow is defined with metamodel annotations defined in the EuGENia approach (also shown in Figure 8.5). The annotations should be self explaining (cf. [KRPP09] for more details). EuGENia uses GMF to generate a complete graphical editor from the annotations.

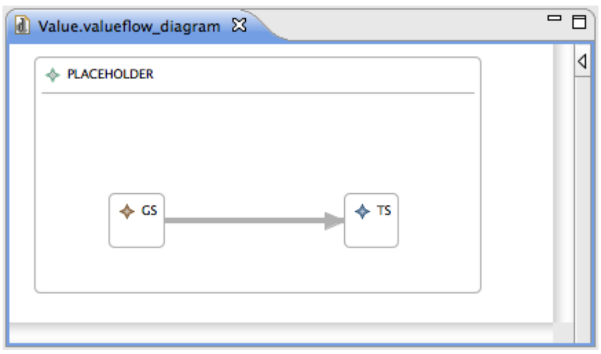


Figure 8.6: ValueFlow unit prototype for Exchange concern dimension.

We used this editor during the architecture definition phase to define unit prototypes for all concern dimensions that needed to be supported by ValueFlow. One example is the *Exchange* concern dimension unit prototype modelled in the generated editor shown in Figure 8.6. It consists of a placeholder Agent, a GiveState `GS` (representing a business value that is given away), a TakeState `TS` (representing a business value that is received) and a link between those states. The two states and the link between them represent a business value exchange in ValueFlow. When used as unit prototype, the states are composed into other Agents and their names are changed. This is defined in a REX_{CM} (cf. Appendix B.3.4).

As a fifth viewpoint, we introduced UML class diagrams for analysis only (i.e., it gives an overview of all classes that appear in Java code, but does not support modification).

All specifications for the model-driven architecture are collected in Appendix B. We identified 11 concern dimensions that we defined as FraCols. The average size of these specifications is 16 LOC. Each viewpoint can show concerns of certain concern dimensions as presented on the left side of Figure 8.7. To add support for a concern dimension to a viewpoint, a unit prototype (created with a normal model editor) and one REX_{CM} was defined (23 in total; average size 26 LOC). Four viewpoints support editing of concerns (i.e., instantiation of unit prototypes) shown on the right side of Figure 8.7. To add editing support for a concern dimension to a viewpoint, a REX_{CL} specification was written (15 in total; average size 37 LOC). Certain concerns are created automatically. For instance, a class is created for an actor as soon as it appears in some use case, and therefore the OpenDocument viewpoint influences the class dimension. All marks on the right side of Figure 8.7 that have no counterpart on the left side identify such situations. Also, some concerns are shown but can not be edited in the corresponding viewpoint (e.g., actors cannot be changed in Java). All marks on the left side that have no counterpart on the right side identify these.

If one wants to use ModelSoC for its own model-driven architecture, one has to write FraCols, REX_{CM}, and REX_{CL} specifications as discussed above. This is a metamodelling task which is done instead of writing model transformations by the process architect of a model-driven architecture (cf. Figure 1.2). Compared to model transformations, our specifications are highly modular as indicated by the small number of LOCs of the specifications in this example. This is not because the demonstrator system we developed with the architecture (discussed next) is relatively small—the architecture itself can be used to develop larger systems.

	usecase	particip.	exchange	flow	trigger	factory	class	dataclass	associate	typebind.	app	security	usecase	particip.	exchange	flow	trigger	factory	class	dataclass	associate	typebind.	app	security
OpenDocument	x	x											x	x				x	x			x	x	
UML use case	x	x	x											x	x			x	x	x	x	x		
Value Flow	x	x	x	x												x	x							
UML class							x	x	x															
Java	x	x	x	x	x	x[1]	x	x	x	x	x	x[2]												
SecProp	x	x	x									x[2]												x

Left side: viewpoints (y-axis) supported by concern dimension (x-axis); Right side: modelling languages used to model (y-axis) in concern dimension (x-axis) ([1]*factory concerns* have complex parameters defined for Java format only; [2]*security concerns* have complex parameters defined individually for SecProp and Java formats)

Figure 8.7: Concern dimensions and viewpoints of the example component-based model-driven architecture.

8.3.2 System Modelling

Once the a model-driven architecture is defined with ModelSoC and set up with Reuseware, developers can use existing model editors or editors that have been specifically generated for the architecture (as the ValueFlow editor shown above) to edit and view different viewpoints. For composed views that are graphical, Reuseware also performs layout composition (cf. Section 12.1) in addition to composing graph fragments. Preserving layout between views helps developers to relate views to each other. Because of traceability issues, such layout preservation is often not well supported in transformation-based MDSD.

Furthermore, developers can make mistakes which lead to inconsistencies that are discovered by Reuseware (e.g., use UML to add an actor to a use case for which no use case description document exists). These errors are discovered, because they lead to invalid fragment instances, port instances, or composition links that are discovered when Reuseware analyses the composition program that represents the concern space or when a type or link matching fails (cf. Section 6.3.1). The errors are annotated to the source models of the error using Eclipse's error marking mechanism. If the model editor used supports this mechanism well, the developer most likely understands the error. However, this is not the case for all editors and sometimes external editors are used (e.g., OpenOffice to edit documents in OpenDocument format). Therefore, improving tool support and integration for error reporting and debugging is part of future work.

Realising a model-driven architecture with ModelSoC, enables us to trace information that is replicated in, scattered over, and tangled in different integrated views. All information about how unit prototypes are composed to the system is collected in one unifying composition program that can be inspected with the UCL editor. Although this composition program quickly becomes large, it still gives a good overview due to the minimal set of concepts in UCL. Anyhow, it is an improvement since transformation-based architectures usually do not give such an overview at all. In the future, the tooling for UCL should be extended to improve the navigation in large composition programs.

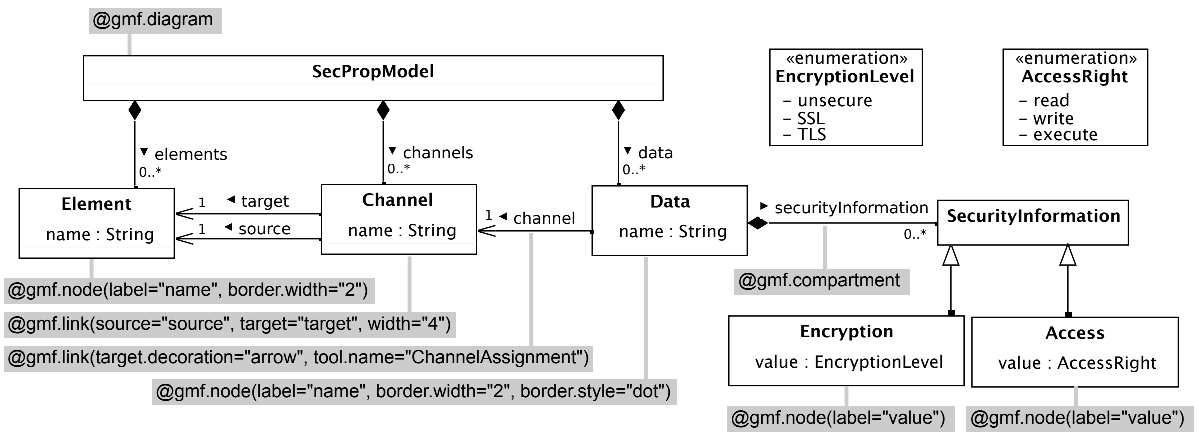


Figure 8.8: Ecore metamodel of SecProp with EuGENia annotations for graphical syntax.

The biggest drawback is currently that the integrated views cannot be edited directly. Rather, small models are edited and the integrated views are created immediately for inspection. Since tracing—which is already used for layout preservation—is simple with the explicit concern space representation, we believe that editable views can be realised by using a round-trip mechanism that propagates changes from the integrated views back. Such a mechanism could even allow editing information in a different viewpoint as it was defined in. We discuss first successful results in this direction in Section 12.2.

While we implemented only two features of a demonstrator system with the model-driven architecture defined above, this architecture can be used to continue development on this or other (possible much larger) systems. For this, no modification of the architecture itself is required. Also, for new model-driven architectures, parts of the specification for this architecture can be reused due to the high modularity—each concern dimensions can be reused individually. An architecture can also be flexibly extended as discussed next.

8.3.3 Model-Driven Architecture Extension

In ModelSoC, model-driven architectures are extensible by adding new concern dimensions, which is an advantage over transformation-based architectures which are often difficult to extend because many transformations interact and have to be adjusted in parallel.

We show this by extending the architecture defined in Section 8.3.1 with a new concern dimension for security—after the functional features were developed and the system, without security, was already running. Security is usually a cross-cutting concern that effects several places in a system. For modelling security information, we developed a small graphical DSML called SecProp. The DSML was motivated by a DSML that was developed by Thales Information Systems in a case study in Modelplex [MOD08]. There we used U-ISC/Graph to add a security viewpoint to a system otherwise modelled with UML only.

The metamodel of SecProp with EuGENia annotations is shown in Figure 8.8. SecProp can be used to define access rights and encryption needs of the business values in the system.

To allow the security modeller to see existing business values that need security properties, the information from the concern dimensions *Usecase*, *Participation*, and *Exchange* needed to be transported to the SecProp viewpoint. This was done by adding new unit prototypes (defined in SecProp) to the corresponding concern dimensions (cf. Figure 8.7 bottom). A multi-format unit prototype for the security dimension was introduced supporting the SecProp and Java viewpoints. To allow integration with other dimensions, new integration points were added to the *Exchange* unit prototypes in SecProp and Java (by providing a new REX_{CM} specification; no changes to the existing specifications or Java fragments were required).

8.4 Conclusion

Chapter Contribution The main contribution of this chapter is the ModelSoC architectural style for the development of component-based model-driven architectures:

C3-1 **Architectural style for MDSD with multi-dimensional separation of concerns** This chapter defined the ModelSoC architectural style for CB-MDSD. In ModelSoC, the complete architecture of a system is reflected in one UCL composition program and concerns are separated in a unified way throughout all abstraction levels and viewpoints. This ensures that all models are kept synchronised and eases traceability, while it preserves the property of MDSD to integrate arbitrary DSMLs into a model-driven architecture. This contribution has the following sub-contributions:

- *Hyperspace model extension for MDSD*
  This chapter presented an extension of the hyperspace model for multi-dimensional separation of concerns that takes the specifics of MDSD into account—replication of information in different formats and the usage of DSMLs as composition (i.e., hypermodule definition) languages.
- *Evaluation of ModelSoC and U-ISC/Graph*
  In this chapter, we successfully evaluated the applicability of ModelSoC, and therewith of U-ISC/Graph in this context, by setting up a model-driven architecture with ModelSoC and developing a small demonstrator system with it. Although we realised only one concrete model-driven architecture with ModelSoC, we can generalise our observations, since the process we chose has properties found in many model-driven architectures: (1) with UML it includes a large standard modelling language, (2) with Java it includes a large standard programming language, (2) with OpenDocument it includes a large standard document language, (3) with ValueFlow, it includes a DSML, (4) the standard languages were not altered for the benefit of ModelSoC or U-ISC/Graph, and (5) we performed an extension of the process by including a new DSML to model security, which is commonly recognised as a cross-cutting concern in system modelling, without altering the previously defined architecture or models. Therefore, we conclude that ModelSoC and U-ISC/Graph is applicable for a number of similar MDSD scenarios.

In the next chapter, we present another architectural style for CB-MDSD which is based on hierarchical composition rather than multi-dimensional separation of concerns.

9 CB-MDSD with Hierarchical Composition (ModelHiC)

In this chapter we introduce ModelHiC, which is an lightweight architectural style for CB-MDSD with a single DSML and UCL. We introduce a process to develop a modelling environment with ModelHiC and perform an evaluation that shows that ModelHiC, and therefore U-ISC/Graph and in particular UCL, can be used to realise component support for a large modelling DSML. The evaluation was carried out in collaboration with Telefónica using a case study they defined in the Modelplex *project. This chapter is an extended version of the following publication:*

- Jendrik Johannes, and Miguel A. Fernández. *Adding Abstraction and Reuse to a Network Modelling Tool using the Reuseware Composition Framework.* In Proceedings of 6th European Conference on Modelling Foundations and Applications (ECMFA 2010), volume 6138 of LNCS, pages 132–143. Springer, June 2010.

In this chapter we introduce the ModelHiC[1] architectural style for CB-MDSD that, in contrast to ModelSoC, is used for developing model-driven architectures that centre around a single DSML to define components and use UCL as composition language. The tools for such architectures are also referred to as domain-specific modelling environments, since their focus is on domain abstraction and not on automatic stepwise refinement by model transformations. However, we enrich such environments with automatic stepwise refinement by *hierarchical composition.* There, different abstraction levels are supported by UCL composition programs that represent abstract views on the system. We allow for an hierarchy of composition programs (i.e., the result of one composition can be input to another composition) and with that realise multiple abstraction levels. Hence, the properties of MDSD, domain-specific abstractions and automatic refinement, are present and thus domain-specific modelling environments with component support fall into our definition of model-driven architectures. Although these architectures use a single DSML for modelling, generators or interpreters to translate the DSML

[1]Component-based model-driven architectures with hierarchical composition

into code or interpret it are still required. In this case, however, the DSML is the lowest level of abstraction the developer works with (which we call Level 0).

In the following, we first introduce a process for developing architectures with ModelHiC in Section 9.1 and shortly describe how ModelHiC is realised with U-ISC/Graph in Section 9.2. We then evaluate ModelHiC in Section 9.3 by adding reuse and abstraction facilities on top of an existing DSML for telecommunication networks and conclude in Section 9.4.

9.1 Development Process for ModelHiC

In this section, we first discuss, why abstraction and reuse is needed for architectures with a single DSML. Afterwards, we introduce a process to developed such architectures.

9.1.1 Abstraction and Reuse support for a DSML

A DSML is used to reduce the complexity arising when developing software systems using a general-purpose language (GPL) such as UML or Java. Unlike a GPL, a DSML focuses on a particular problem domain and contains a relatively small number of constructs that are immediately identifiable by domain experts and allow them to construct concise models capturing the design of the system at an appropriate level of abstraction. While typical DSMLs are small languages with a manageable number of concepts, a DSML that embodies a standard vocabulary of a larger domain may grow large in its number of concepts. This eventually compromises the very aims against which the DSML was built in the first place: domain focus and conciseness.

One example of such a complex DSML is the telcommunications DSML presented in [EFM09], which we extend using ModelHiC in Section 9.3. While this language is a DSML in the sense that it provides dedicated constructs for the telecommunication domain, its size in terms of the number of constructs and features is comparable to that of a GPL such as the UML—the Common Information Model (CIM) [DMT10] standard, on which the DSML is based, defines more than 1500 concepts.

However, the domain for which CIM is designed can be split into more specialised domains where not all details of CIM are required in each of them. The classical MDSD approach would be to construct new DSMLs that provide abstractions over and above the constructs provided by CIM. This means that different DSMLs, all in the telecommunication domain, are created for different abstraction levels and are combined in a transformation-based model-driven architecture.

To employ this approach, one has to identify the abstraction levels and decide which DSMLs, and with which constructs, have to be created. This is an iterative process, since a DSML has to be tested and used by the domain experts to evaluate its usefulness and improve it. Updating one DSML alone can be costly when the associated tooling, which includes the model transformations, needs to be adapted manually, which is often the case with today's DSML development technology as experienced in the development of tooling for the original CIM-based DSML [EFM09]. This cost would even increase if multiple DSMLs, which are connected in a model-driven architecture, are updated and co-evolved.

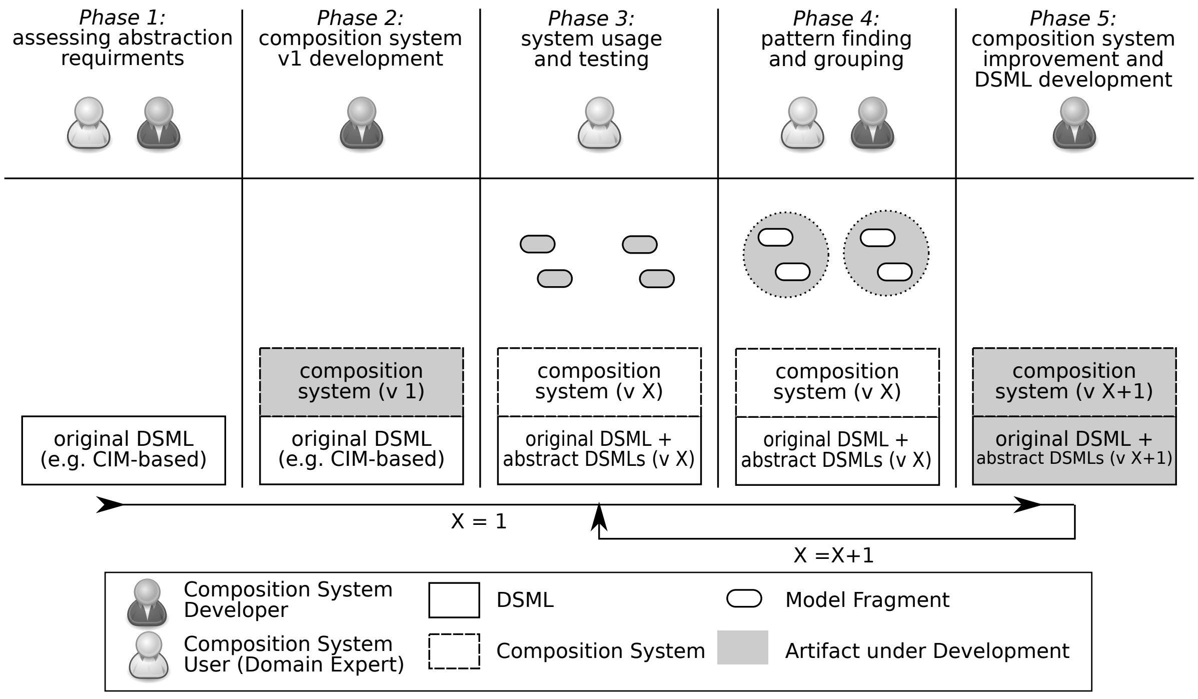

Phase 1:
assessing abstraction requirments
Phase 2:
composition system v1 development
Phase 3:
system usage and testing
Phase 4:
pattern finding and grouping
Phase 5:
composition system improvement and DSML development
original DSML (e.g. CIM-based)
composition system (v 1)
original DSML (e.g. CIM-based)
composition system (v X)
original DSML + abstract DSMLs (v X)
composition system (v X)
original DSML + abstract DSMLs (v X)
composition system (v X+1)
original DSML + abstract DSMLs (v X+1)
X = 1
X =X+1
Composition System Developer
Composition System User (Domain Expert)
DSML
Composition System
Model Fragment
Artifact under Development

Figure 9.1: Process for developing component support for DSML wit ModelHiC.

Instead of using a classical MDSD approach with a transformation-based architecture, as described above, we propose ModelHiC to develop abstraction, and provide associated tooling, for a DSML. As we discuss next, this solution can be used as (1) an alternative for the transformation-bases approach with multiple DSMLs, (2) prototyping for finding the DSMLs in the transformation-bases approach, and (3) basis for implementing the transformation-bases approach.

9.1.2 Process to develop Reuse and Abstraction Support for a DSML

Designing a DSML and tooling for it requires feedback from the domain experts. Therefore, rapid prototyping and continuous updating of the DSML tooling, based on that feedback, is desirable. Therefore, we propose an iterative process for the development of abstraction and reuse features for a DSML based on ModelHiC. The process is supported by rapid prototyping with REUSEWARE which allows for continuous adjustment of modularity concepts with FraCols and REX_{CM}. The process, consisting of five phases, is visualised in Figure 9.1:

1. In the first phase, the process architect, collects initial information about desired abstraction levels from the domain experts who are already familiar with the complex existing DSML. A method to do this is to let the domain experts define models with the DSML and explain to the process developer where in these models they see possibilities for abstraction and reuse. In the ModelHiC evaluation we present in Section 9.3, this was done by telecommunications experts at Telefónica who marked parts in a model which they consider separate components.

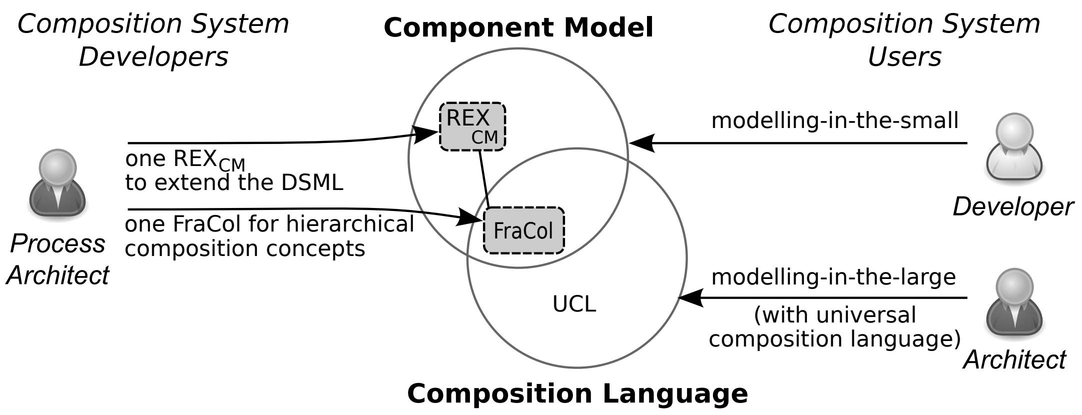


Figure 9.2: ModelHiC configuration of the generic CB-MDSD composition system.

2. In the second phase, the developer designs the first composition system version that is flexible enough to cover all abstraction and reuse requirements identified in phase one, but is customised enough such that it can be used by domain experts. We, in the role of the process architect, developed such a composition system with Reuseware using ModelHiC. That is, we configured the the generic CB-MDSD system with FraCols and REX_{CM} specifications for the telecommunications DSML based on the feedback we got in phase one.
3. In the next phase, the domain experts uses the composition system, creates fragments, and gives feedback. The developer can give support in this phase.
4. In phase four, the fragments and composition programs are analysed to find common patterns and to group the fragments following these patterns. From these patterns, the process architect can derive restrictions and default behaviour for the composition system or identify constructs for new abstract DSMLs and refine the abstraction levels.
5. In the last phase, the developer improves the composition system based on the results from the previous phase or builds new DSMLs that replace (parts of) the composition system. The result is then given to the domain-experts. It is either the final development environment or a next prototype and phase three to five are repeated.

9.2 ModelHiC implemented with U-ISC/Graph

ModelHiC can be realised with U-ISC/Graph by a specific configuration of the generic CB-MDSD composition system shown in Figure 9.2. Instead of doing a rich configuration as done in ModelSoC (cf. Figure 8.4), we limit ourselves to a REX_{CM} specifications for the Ecore metamodel of the central DSML and do not use REX_{CL}. Instead, UCL is used directly as composition language. One FraCol is defined to support one modularity concepts (i.e., concern dimensions) for hierarchical composition. The FraCol is then mapped only once to the DSML with a REX_{CM} specification.

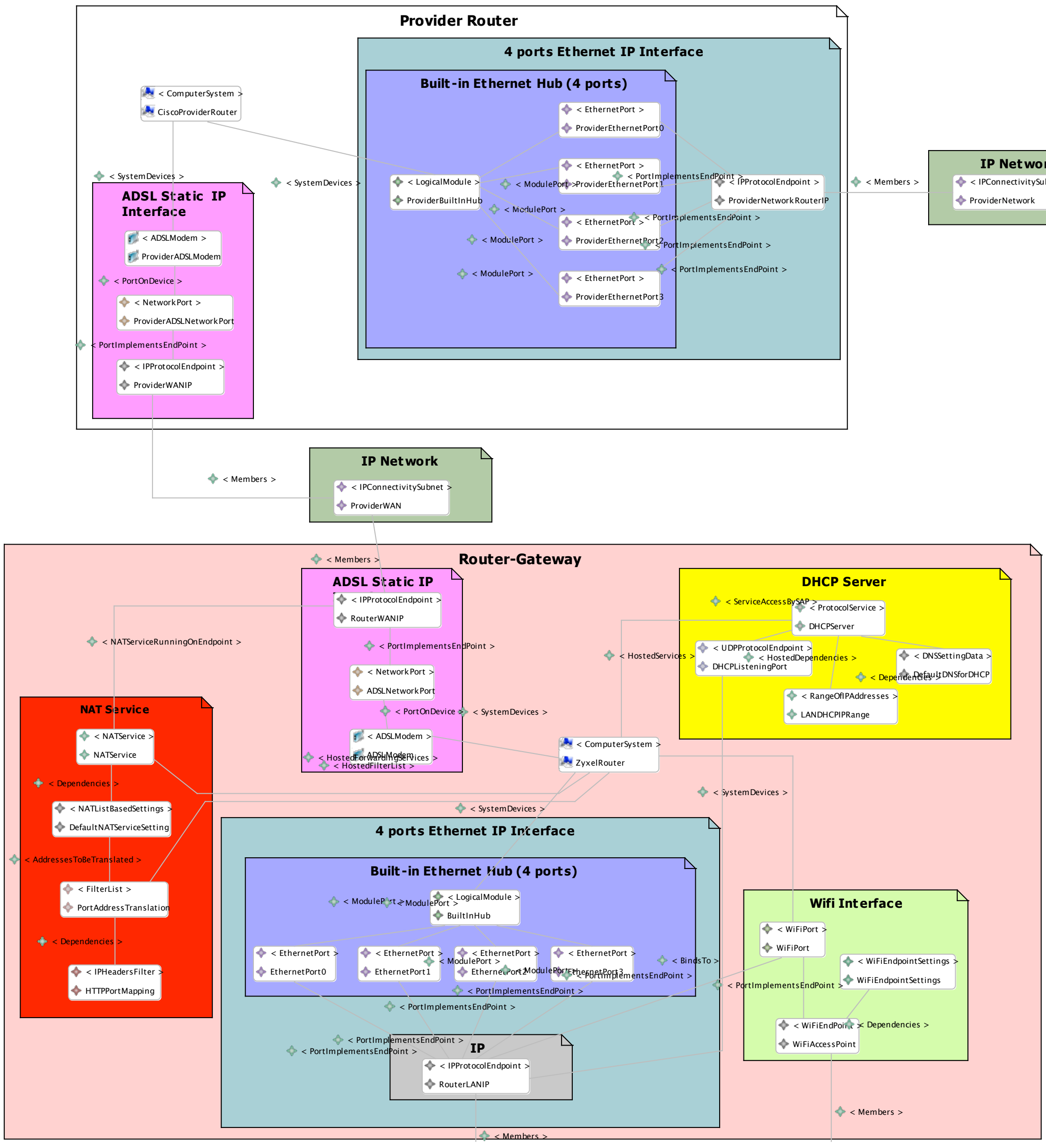


Figure 9.3: Excerpt from CIM model of an ADSL service network configuration (provided by Telefónica R&D for the MODELPLEX project [MOD08]).

9.3 Evaluation of ModelHiC: ReuseCIM

In this section we evaluate the applicability of ModelHiC and usability of UCL by presenting how we develop abstraction and reuse support for a large telecommunications DSML. For this, we started the development process of Section 9.1, executing the first two phases of it.

We continued the work of [EFM09] who present experiences gained in developing a DSML for telecommunication experts at Telefónica. There, the Graphical Modeling Framework (GMF) [Gro09] (cf. Section 5.1.2) was used to develop a graphical editor as core of the DSML tooling. [EFM09] identified challenges for technologies that were not met by the tooling used so far. One of the identified challenges is *abstraction and reuse*. That is, supporting domain experts to create abstract views of complex models and to develop reuseable model components. We realised this abstraction and reuse support as a composition system with ModelHiC and therewith integrated Reuseware into the DSML tooling.

The telecommunications DSML is based on the Common Information Model (CIM) [DMT10] that is a Distributed Management Task Force (DMTF)[2] standard for system, network, application, and service definitions. The graphical editor, which was the main part of the tooling for the DSML as presented in [EFM09], is directly based on an Ecore metamodel that represents a large part of the CIM standard.

In the Modelplex project, Telefónica defined a case study in which they not only use the CIM-based DSML for telecommunication network modelling, but also formulate abstraction and reuse concerns [MOD08]. Driven by this case study, we developed abstraction and reuse tooling with U-ISC/Graph using ModelHiC. In the spirit of [Hen09], who called languages extended with component support *reuse languages*, we name the such extended CIM-based DSML *ReuseCIM*.

9.3.1 Development Process Utilisation

Phase one of the process (cf. Figure 9.1) was performed by domain experts at Telefónica who defined a model of a typical ADSL service network configuration for their customers consisting of 52 model elements. The major part of the model, displayed in the graphical editor of the DSML, is shown in Figure 9.3. After the model was defined, the domain experts at Telefónica marked and named parts of the model that can be abstracted into a single concept on a higher abstraction level and reused at different places in the model. For the marking, they used notes with different colours. These are only parts of the graphical syntax and do not change the meaning of the underlying model. In addition, for each concept they marked, they provided a list of attributes that need to be visible on a higher abstraction level.

The concepts were then grouped into seven specialised domains (Protocols, Physical Interfaces, Logical Interfaces, Systems, Network Devices, Network Links, and Network Topologies) that reside on different abstraction levels (Levels 1–4) as summarised in Figure 9.4. For each specialised domain and/or each abstraction level, a separate DSML could potentially be defined. Between the different specialised domains, dependencies can be identified (arrows in Figure 9.4). They express which domains from a lower abstraction level are used to express concepts of a higher abstraction level. In principle, concepts of Level x can be represented by

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

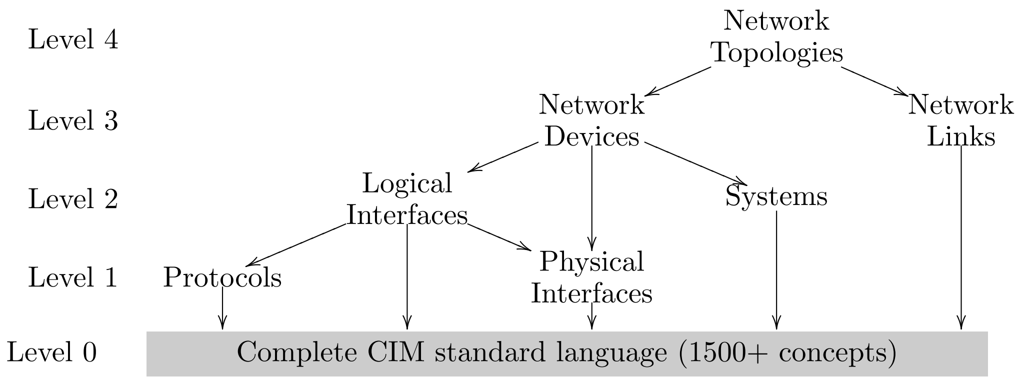


Figure 9.4: Abstraction levels and component types for network models.

concepts of Level $x-1$. Certain concepts of higher abstraction levels (Level 2, 3 or 4) are also expressed directly with concepts of Level 0. This is, because the CIM standard (Level 0) itself offers constructs of low (e.g., `EthernetPort`) but also high (e.g., `System`) abstraction.

In phase two of the process, we configured our generic CB-MDSD composition system for the CIM-based DSML driven by the requirements specified in the example model (cf. Figure 9.3). First, we decomposed the model into fragments following the decomposition suggestions marked in the model. Second, we defined UCL composition programs using the UCL editor. For each set of fragments that makes up one of the components marked in the example model, one composition program was defined and the corresponding fragments were inserted. Composition links, however, were not created at this point, because the fragments did not yet provide composition interfaces. Third, we extended CIM to ReuseCIM by configuring the generic composition system to support the composition of CIM fragments. Finally, we adjusted the fragments to ReuseCIM and completed the UCL composition programs such that they recompose the original example model.

Figure 9.5 shows the fragments and composition programs that are the decomposed version of the upper part of the example model shown in Figure 9.3. The three rows in the figure correspond to the abstraction Levels 1–3 (from bottom to top). On Level 1, we have the fragment `BuiltInEthernetHub`, which is a Physical Interface modelled in the CIM-based DSML, and the fragment `IP`, which is a Protocol also modelled in the CIM-based DSML. On Level 2, three CIM models are defined: two Logical Interfaces and one System. The first Logical Interface (`EthernetIPInterface`) is a composition program that contains the two Level 1 fragments. On the contrary, the second (`ADSLStaticIPinterface`) is directly modelled in the CIM-based DSML. The fragment `System` is also modelled in the CIM-based DSML. On Level 3, we then have one composition program (`ProviderRouter`) that composes the three Level 2 fragments.

An overview of all fragments and composition programs we obtained by decomposing the complete example model that consists of 52 elements, is given in Figure 9.6. In total, 12 fragments and 8 composition programs were defined. In average, each fragment contains 2.42 model elements which means that a total of 29 model elements were created in the CIM-based DSML. In the original model, 52 elements were modelled, which means that 44% of the example model can be created by reusing fragments instead of modelling in the CIM-based DSML.

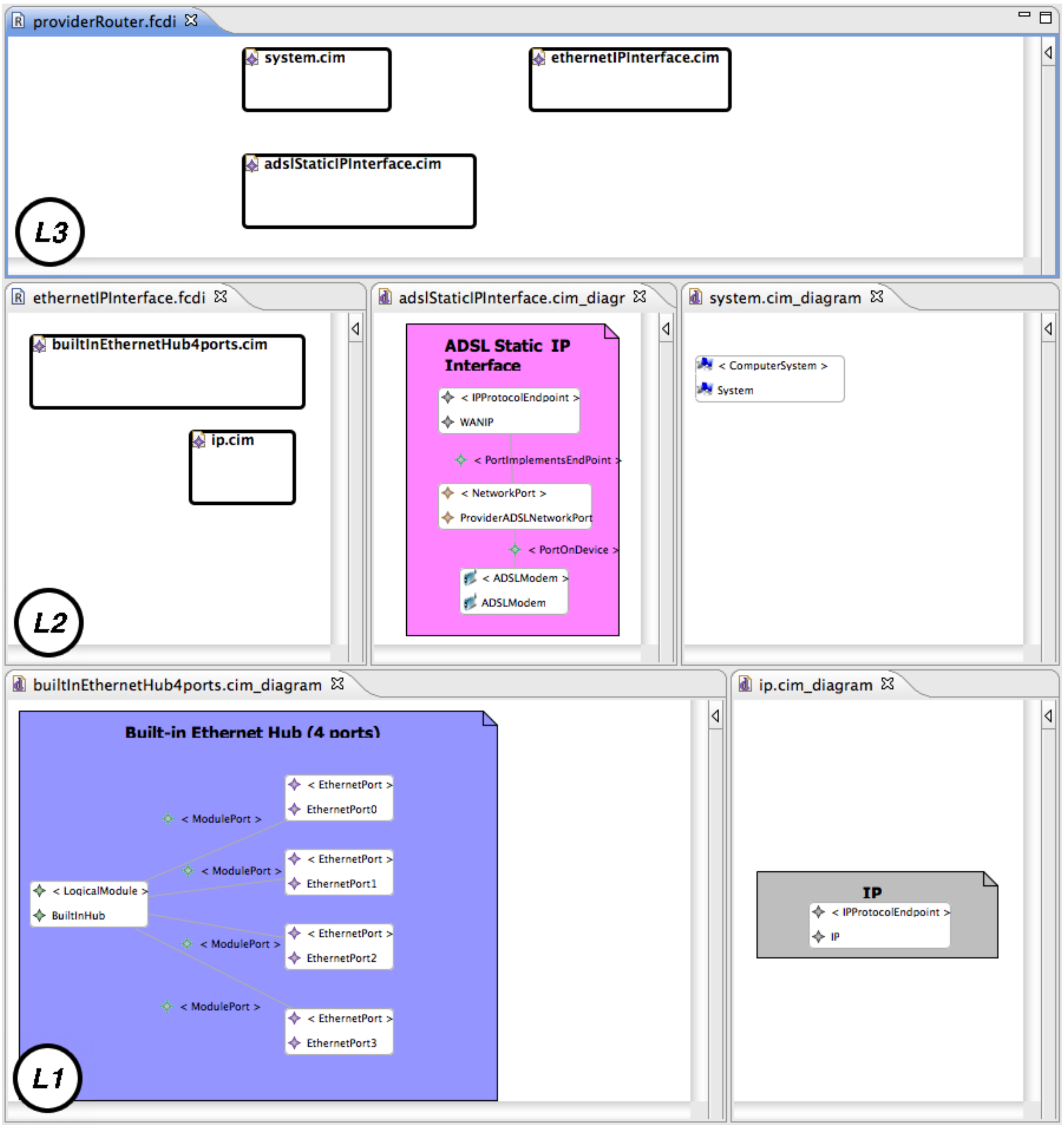


Figure 9.5: Upper part of Figure 9.3 decomposed into fragments and composition programs on Levels 1–3 (bottom to top).

		fragments	avg. no. of model elements	comp. programs	avg. no. of fragments in comp. prgr.	reused
Level 4	NW Topologies	0	n.a.	1	8.00	0
Level 3	NW Devices	0	n.a.	4	3.25	4
	Network Links	2	1.00	0	n.a.	4
Level 2	Logical Interfaces	1	3.00	3	2.67	6
	Systems	1	1.00	0	n.a.	4
Level 1	Physical Interfaces	3	3.00	0	n.a.	4
	Protocols	5	2.60	0	n.a.	7
Total		12	2.42	8	3.63	29

Figure 9.6: Fragments of the example model.

9.3.2 ReuseCIM Composition System

Since ReuseCIM should extend the existing tooling (the graphical editor of the CIM-based DSML and the UCL editor of REUSEWARE) we did not alter or extend the DSML's metamodel. Instead, we introduce five prefixes (`+`, `%`, `?`, `*`, `-`) that can be prepended to attribute values of model elements to define the composition interface of a CIM model fragment. The prefixes are summarised in Figure 9.7 and are explained in the following on the example of Figure 9.8.

Looking at the two Level 1 models (Figure 9.8; bottom row), we can see that the element `BuiltInHub` in the fragment `BuiltInEthernetHub` is prefixed with `+` and named `+BuiltInHub`. `+` exports the element to the composition interface and lets it appear with the name of the element (in this case `BuiltInHub`). This can be seen in the composition program `EthernetIP-Interface` on Level 2 (left in middle row). Similar is done with the element `IP` in the fragment `IP`. Furthermore, we add a new element (depicted in orange) to the fragment `BuiltInEthernet-Hub` that we name `?PrototcolEndpoint` and connect it to other elements in the fragment. `?` is used to define a variation point. That is, this is not an element with meaning, but only a placeholder. It is replaced or removed during composition.

The composition program `EthernetIPInterface` (1st in middle row) on Level 2 can now make use of the composition interface. Concretely, the exported element `IP` is linked to the variation point `?PrototcolEndpoint`. Executing the invasive composition yields a fragment `EthernetIPInterface` that is equal to the corresponding part of the original use case model (cf. Figure 9.3). In the fragment `ADSLStaticIPInterface`, we declare two elements to be exported using `+` (`WANIP` and `ADSLModem`). In the fragment *System*, we also export the element `System`, add three variation points (using `?`) and add the extension point **Modules* (using `*`). In contrast to a variation point, an extension point defined by `*` allows for multiple extensions and needs to be explicitly removed by using `-` in a composition program.

Using the exports, variation points, and extension points defined in the fragments of Level 1 and 2, the composition program `ProviderRouter` on Level 3 (Figure 9.8; top) can now be enriched with the composition links that are required to compose a CIM model `ProviderRouter` that is equal to the corresponding part of the original example model (cf. Figure 9.3 top).

Furthermore, the `%` prefix is used in all fragments to export attributes to the composition interface. Figure 9.9 shows this exemplarily for the element `System` in the fragment `System` (middle row on the right in Figure 9.8). Here, we export the Description attribute by setting

Prefix	Type	CIM Level 0 Fragments	CIM Level 1+ Composition Programs
+	Element Export	If used in *elementName*, the corresponding element is *exported*	If used in an attribute that changes the element name, the element is re-exported; if not used in an attribute that changes the name of an exported element, the element is not re-exported
%	Attribute Export	Use this to export an attribute for modification; if the prefix is followed by the same value in multiple places, the attributes are merged and are always set to the same value	Use this to re-export an attributed
?	Variation Point Declaration	Create an element of any kind and use this in *elementName* to declare a *variation point*: a point that can be connected to one exported element and is automatically removed if not connected	n.a.
*	Extension Point Declaration	Create an element of any kind and use this in *elementName* to declare an *extension point*: a point that can be connected to multiple exported element	n.a.
-	Extension Point Removal	n.a.	Use this to remove an existing extension point

Figure 9.7: Naming conventions to define composition interfaces for CIM fragments.

it to `%Description`. The value followed after `%` defines the name of the attribute on the next abstraction level (here `Description`). Furthermore, the Name attribute is exported to the attribute `System` (by using `%System`).

The exported attributes can be modified in the properties of a corresponding fragment instance in a composition program. Figure 9.10 shows these attributes for the instance of the fragment `System` in the composition program `ProviderRouter` (top row in Figure 9.8). The properties also list all extension points, which is only `*Modules` in this case. An extension point can be removed if it should be no longer visible on the next abstraction level by setting it to `-`, which we do in the case of `*Modules` here.

Another feature to improve the user experience is the specification of icons that are then shown on fragments in composition programs. In ReuseCIM, domain experts can specify icons themselves by placing them next to the fragments they develop.

The composition system for ReuseCIM was defined in terms of a FraCol, a UCL syntax customisation (cf. Section 6.4.2), and three REX_{CM} specifications. Furthermore, it reused the default FraCol presented in Section 7.2. The FraCol specifies the central concepts of ReuseCIM which is used for the implicit parts of composition interfaces of CIM fragments. It is used by the UCL syntax customisation and one REX_{CM} specification. The other two REX_{CM} specifications, which implement the default FraCol, deal with declared interfaces which composition system users (i.e., CIM domain experts) define by using the prefixes described above.

We used two FraCols and three REX_{CM} specifications for clarity. Conceptually, the specifications can be merged to one FraCol and one REX_{CM} as required by the ModelHiC style (cf. Section 9.2).

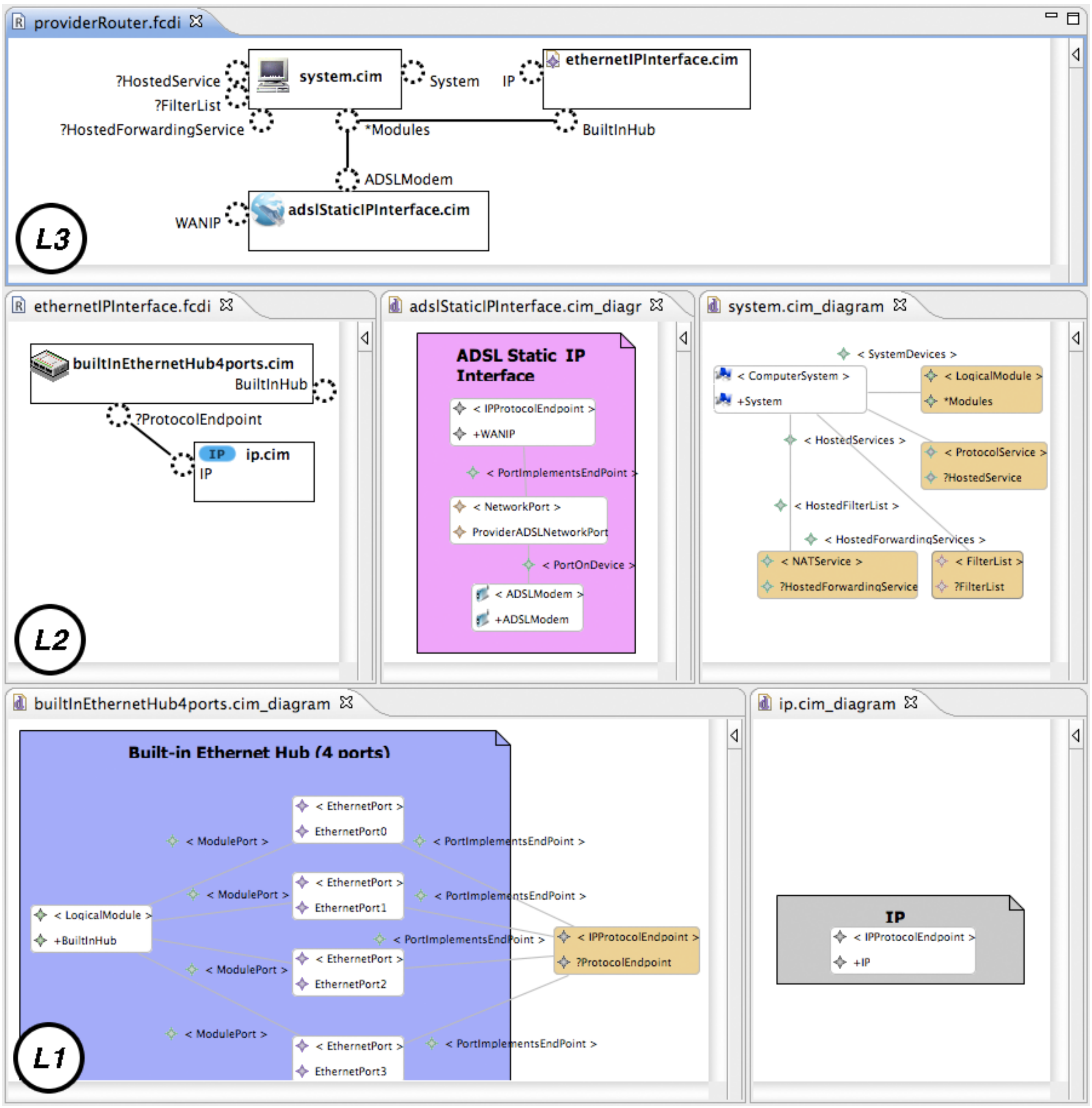
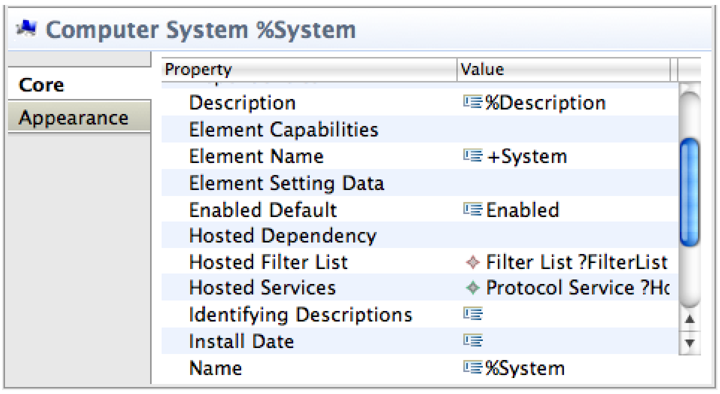


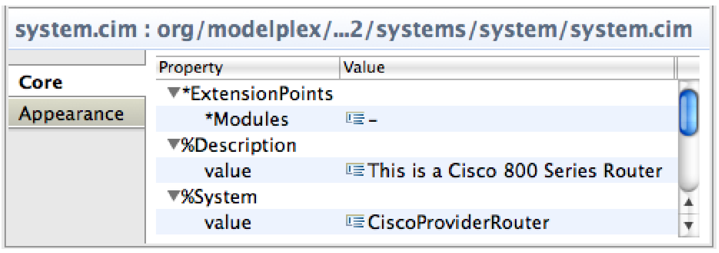
Figure 9.8: Figure 9.3 decomposed into fragments and composition programs on Levels 1–3.


Computer System %System

Core
Appearance

Property	Value
Description	%Description
Element Capabilities	
Element Name	+System
Element Setting Data	
Enabled Default	Enabled
Hosted Dependency	
Hosted Filter List	Filter List ?FilterList
Hosted Services	Protocol Service ?Hc
Identifying Descriptions	
Install Date	
Name	%System

Figure 9.9: Properties of the model element `System` (cf. Figure 9.8 middle).


system.cim : org/modelplex/...2/systems/system/system.cim

Core
Appearance

Property	Value
▼*ExtensionPoints	
*Modules	-
▼%Description	
value	This is a Cisco 800 Series Router
▼%System	
value	CiscoProviderRouter

Figure 9.10: Properties of the Level 2 fragment *system.cim* (cf. Figure 9.8 top).

Implicit Composition Interfaces

The core of ReuseCIM is hidden to the composition system users. To explain it, we have to understand the structure of the CIM models. Each model has a `CIM_Model` as its root node. All the other nodes, which are visible as boxes in the diagrams (cf. Figures 9.3, 9.5, and 9.8) are directly contained in the `CIM_Model` via the containment reference **`elements`**. Consequently, the **`elements`** containment reference always needs to be considered in a composition (this is the implicit part) and all other references only need consideration if they are connected with an addressable point (this is the declared part).

We capture the implicit part of ReuseCIM in a FraCol shown in Listing 9.1 and bind it to the CIM metamodel with the REX$_{\mathrm{CM}}$ specification in Listing 9.2. We define two fragment roles—`Core` and `Element`—with one static port each—**`extensions`** and **`contents`**. Between these two ports we allow contributing composition links through the contributing association **`extension`**. In the binding to CIM (cf. Listing 9.2) we define an empty `CIM_Model` as `Core` (OCL expression ***`elements->isEmpty()`***) that can be extended by making the **`elements`** reference a hook. A non-empty `CIM_Model` is considered as `Element` (OCL expression ***`not elements->isEmpty()`***) by making all nodes of the **`elements`** reference prototypes.

```
fracol org.modelplex.cim.ReuseCIMCore {
    fragment role Core {
        static port type extension;
    }
    fragment role Element {
        static port type contents;
    }

    contributing association extension {
        Element.contents --> Core.extension
    }
}
```

Listing 9.1: CIM FraCol specification.

```
componentmodel org.modelplex.cim.rex.ReuseCIMCore
implements     org.modelplex.cim.ReuseCIMCore
epackages <http://www.tid.es/cim>
rootclass CIM_Model {
    fragment role Core if $elements->isEmpty()$ {
        port type extension  {
            CIM_Model.elements is hook {}
        }
    }
    fragment role Element if $not elements->isEmpty()$ {
        port type contents {
            CIM_Model.elements is prototype {}
        }
    }
}
```

Listing 9.2: CIM REX_{CM} core specification.

Composition Program Syntax Customisation

The implicit part of the ReuseCIM should be hidden to the composition system user and used automatically. For that, we define a customised composition language syntax as discussed in Section 6.4.2. To define such a customisation, Reuseware contains a small specification language that is in its structure similar to the REX languages. It allows the composition system developer to specify syntax properties for fragment roles and port types. When a fragment instance with its port instances is rendered in the composition program editor, the syntax properties of the corresponding fragment roles and port types are taken into account. The syntax customisation for ReuseCIM is shown in Listing 9.3.

At first, we define that *EmptyCore.cim* (Lines 4 and 5) is added automatically to each composition program. *EmptyCore.cim* is thus part of ReuseCIM and delivered with the specifications. It contains one single `CIM_Model` node and is therefore a `Core` fragment (cf. Listing 9.2). The target UFI (Line 6) for the *EmptyCore.cim* fragment instances is derived from the UCPI of the composition program (by replacing the extension with `cim`). Furthermore, we define that `Core` fragments are hidden (Line 7), which hides the *EmptyCore.cim*, and that the `contents` port of all `Element` fragments is hidden (Line 14), which hides the `contents` port of each CIM fragment that is added (since each is an `Element`; cf. Listing 9.2). Note that the hidden `Element.contents` and `Core.extension` ports are always connected automatically by

```
 1 compositionlanguagesyntax org.modelplex.cim.rex.ReuseCIMSyntax
 2 implements                org.modelplex.cim.ReuseCIMCore {
 3     fragment role Core {
 4         fragment   = $'EmptyCore.cim'$
 5         ufi        = $'org/modelplex/cim/lib/EmptyCore.cim'$
 6         target ufi = $ucpi.trimExtension().appendExtension('cim')$
 7         visible    = $'false'$
 8     }
 9     fragment role Element {
10         icon   = $ufi.trimExtension().appendExtension('gif')$
11         width  = $'100'$
12         height = $'50'$
13         port type contents {
14             visible = $'false'$
15         }
16     }
17 }
```

Listing 9.3: CIM composition program syntax customisation.

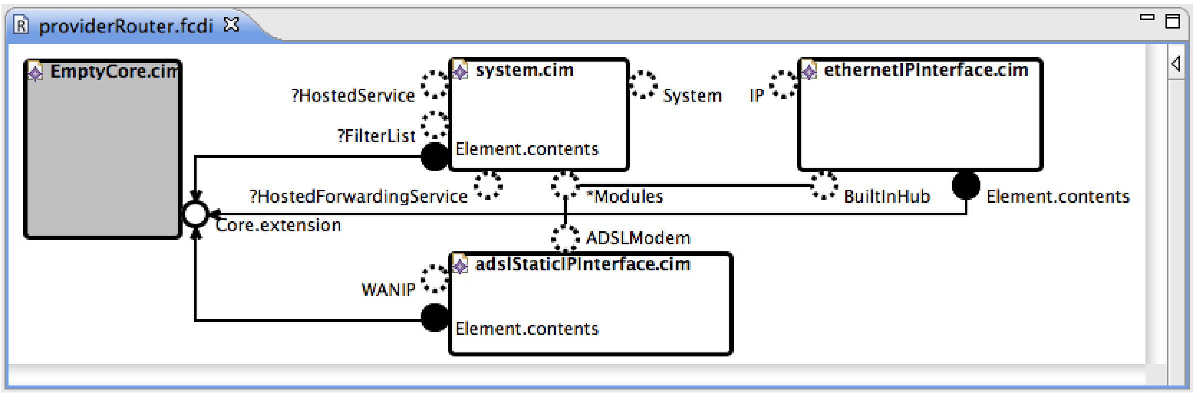


Figure 9.11: *ProviderRouter* composition program without custom syntax (cf. Figure 9.8 top).

the composition program editor, since it tries to connect hidden ports automatically following the specified associations (which is the `extensions` association in this case; cf. Listing 9.1). To illustrate the hiding of the core through the syntax customisation, Figure 9.11 shows the *ProviderRouter* composition program without the custom syntax (cf. top of Figure 9.8). Lines 10–12 of Listing 9.3 defines the look of visible `Element` fragment instances in composition programs by defining an initial size and that a `gif` icon is used that is placed next to the fragment in the repository.

Declared Composition Interfaces

In ReuseCIM, composition interfaces are declared by domain experts using naming conventions. Naming conventions, which were already used in COMPOST (cf. Section 3.1), are a convenient way to support composition interface declaration without changing a language's syntax. This support for declared composition interfaces is specified in Listings 9.4 and 9.5. Since we do not require specific fragment roles for this, we use the *default* FraCol (cf. Section 7.2) in both

```
 1 componentmodel org.modelplex.cim.rex.ReuseCIM
 2 implements     org.reuseware.lib.systems.default
 3 epackages <http://www.tid.es/cim>
 4 rootclass CIM_Model {
 5     fragment role Default {
 6         port type Config {
 7             ManagedElement is anchor if $elementName.startsWith('+')$ {
 8                 port = $elementName.substring(2,elementName.length())$
 9             }
10         }
11         port type Config {
12             ManagedElement is slot if $elementName.startsWith('?')$ {
13                 mode   = $'bind'$
14                 port   = $elementName$
15                 remove = $'true'$
16             }
17         }
18         port type Config {
19             ManagedElement is slot if $elementName.startsWith('*')$ {
20                 mode = $'extend'$
21                 port = $elementName$
22             }
23         }
24         port type Config {
25             ManagedElement is slot if $elementName.startsWith('-')$ {
26                 port   = $elementName$
27                 remove = $'true'$
28             }
29             ManagedElement.elementName is value hook if $elementName.startsWith('*')$ {
30                 port  = $'*ExtensionPoints'$
31                 point = $elementName$
32             }
33         }
34     }
35 }
```

Listing 9.4: CIM REX$_{\text{CM}}$ for element export, variation, and extension.

REX$_{\text{CM}}$ specifications. Listing 9.4 defines a REX$_{\text{CM}}$ that interprets the prefixes that concern element export, variation, and extension (`+`, `?`, `*`, and `-`; cf. Section 9.3.2)

- `+` (Lines 8–10): Any element that has an `elementName` starting with `+` is exported by making it an anchor. The name of the port for the anchor is constructed by removing the `+` prefix.
- `?` (Lines 11–17): Any element that has an `elementName` starting with `?` is recognised as a variation point by making it a slot operating in *bind* mode. The name of the port for the slot is taken from `elementName`. Furthermore, setting the `remove` property to *true* ensures that the slot is also removed if it is not bound.
- `*` (Lines 18–23): Any element that has an `elementName` starting with `*` is recognised as an extension point by making it a slot operating in *extend* mode. The name of the port for the slot is taken from `elementName`.
- `-` (Lines 24–33): Any element that has an `elementName` starting with `-` is removed (`remove` property set to *true*). We enable composition system users to set `elementName` of an existing extension point to `-` by defining a value hook (Lines 30–33) for each extension point (prefix `*`) that is accessible through the `*ExtensionPoint` port.

```
 1 componentmodel org.modelplex.cim.rex.ReuseCIMAttributes
 2 implements     org.reuseware.lib.systems.default
 3 epackages <http://www.tid.es/cim>
 4 rootclass CIM_Model {
 5     fragment role Default {
 6         homo port type Config {
 7             /* a similar rule is defined for each EAttribute in the CIM metamodel */
 8             ManagedElement.description is value hook if $description.startsWith('%')
 9                     and not elementName.startsWith('*') and not elementName.startsWith('?')$ {
10                 homo port = $description$
11                 port      = $'description'$
12                 point     = $'value'$
13             }
14 
15             ManagedElement.elementName is value hook if $elementName.startsWith('+')$ {
16                 homo port = $'%'.concat(elementName.substring(2,elementName.length()))$
17                 port      = $'elementName'$
18                 point     = $'value'$
19             }
20         }
21     }
22 }
```

Listing 9.5: CIM REX_{CM} for attribute export.

Listing 9.5 defines a REX_{CM} that interprets the prefixes that concern attribute export (`?` and `+`; cf. Section 9.3.2). Since we want to give composition system users the possibility to export any attribute, we required one value hook rule for each `EAttribute` in the CIM metamodel. Lines 8–13 shows this rule for the attribute `description`.[3] This rule makes a `description` that starts with `%` a value hook except those that are parts of variation or extension points (`?` and `*` prefixes). Such an export of the `description` was shown in Figure 9.9. Because the `elementName` of an exported element (`+` prefix) should also be exported, we added one additional value hook rule for that (Lines 15–19).

If a CIM fragment is added to a composition program, a value setting is automatically created for each value hook (cf. Section 6.4). These settings are then shown in the properties view of the composition program editor (e.g., Figure 9.10) and can be edited by composition users in a similar way as they edit attributes of CIM models in the CIM-based DSML editor (e.g., Figure 9.9).

9.3.3 Discussion

In this section, we have introduced ReuseCIM, a complex DSML that supports domain experts to define reuseable models on different abstraction levels. ReuseCIM uses a ModelHiC configuration of our generic CB-MDSD composition system as a lightweight alternative to developing a set of DSMLs for different abstraction levels and connect them via transformations, which would have been the traditional MDSD approach. ReuseCIM was developed with ModelHiC with relatively little effort—the complete system definition consists of only 103 Lines specification code (Listings 9.1–9.5). Nevertheless, the system, described in Section 9.3.2, can be

[3]Instead of writing the rules for each attribute manually, we used a generator that produces a rule similar to the one shown in Lines 8–13 of Listing 9.5 for each `EAttribute` in the CIM metamodel.

directly used by domain experts without knowledge about graph fragment composition. Abstraction levels can be introduced and used by the domain experts as required with the UCL editor.

Using ReuseCIM, we were able to recompose the complete original example model from the fragments (cf. Figure 9.6) that were created from it based on the decomposition proposed by the domain experts (cf. Figure 9.3). As mentioned, 44% of the model consists of reused fragments compared to complete manual modelling.

The features of this first version of ReuseCIM give the domain experts a lot of freedom. They can introduce new fragments and design their composition interface and their look in composition programs individually. They can also introduce new abstraction levels without modifying any language, tooling, or ReuseCIM specifications, since all CIM models on abstraction levels higher than Level 0 are UCL composition programs. Thus, the presented ReuseCIM is useful in particular in the early stages of building component support for the DSML to find appropriate abstraction levels.

Still, there are also (potential) drawbacks in using the developed composition system over the classic MDSD approach. First, the flexibility we gave to the composition system inherently comes with the danger that it again threatens the simplicity and abstraction we wanted to introduce with the composition system in the first place. Since the domain experts control the composition interfaces themselves to a large degree, they might overload the interface of components or design them too restrictive, which makes fragments hard or impossible to reuse. Second, the tooling (in particular the user interface), which is only a thin layer on top of Reuseware, can never be as highly customised or adjusted to other platforms and technologies as individual DSMLs can be.

These drawbacks, however, only apply in certain scenarios. For example, when new users that only work on one particular abstraction level are introduced to the DSML often, which justifies the costs of developing customised tools for them; or when people have to work on specific platforms with resource restrictions that cannot be met by the Reuseware tooling. To answer such questions for the CIM case, we need to perform more case studies and, most importantly, get feedback from the domain experts on these questions.

In any case, feedback from domain experts is of high importance for the whole idea of DSML building. We claim that creating a flexible composition system for an existing complex DSML with ModelHiC is a good first step to build abstraction and reuse facilities on top of the existing complex DSML. Even if we switch to a classic MDSD approach later, or a ModelSoC based one, ModelHiC is a lightweight way to obtain a first prototype that can then be used and tested by the domain experts to collect feedback on what the correct abstraction levels are.

9.4 Conclusion

Chapter Contribution The main contribution of this chapter is the ModelHiC architectural style for the development of component-based model-driven architectures:

C3-2 **Architectural style for MDSD with hierarchical composition**
With ModelHiC, we introduced a lightweight style to use U-ISC/Graph to define additional abstraction levels on top of a complex DSML following a hierarchical composition. This was shown on a large DSML based on an industry standard (the CIM-based DSML). This can be used both for building a productive development environment or for initial prototyping. This contribution has the following sub-contributions:

- *Iterative process for developing abstraction and reuse support for a DSML*
  We defined an iterative process for developing abstraction and reuse facilities for existing DSMLs. The process allows for rapid prototyping thanks to the flexibility of U-ISC/Graph and Reuseware, which is fundamental to involve domain experts in the process.
- *Evaluation of ModelSoC and U-ISC/Graph*
  In this chapter, we successfully evaluated the applicability of ModelHiC, and therewith of U-ISC/Graph in this context, for one complex DSML from the telecommunications domain. We can generalise our observations, since the DSML we chose has three properties which are typical conditions in practice: (1) the DSML is large, (2) the DSML is based on an industry standard, and (3) the DSML was developed before we considered using ModelHiC for adding component support and was not modified in any way for the benefit of ModelHiC or U-ISC/Graph. Therefore, we conclude that ModelHiC and U-ISC/Graph are applicable for a number of similar MDSD scenarios.

This concludes this part of the thesis on using U-ISC/Graph for CB-MDSD. We introduced two architectural styles, ModelSoC and ModelHiC, for CB-MDSD. While the two presented architectural styles are the major application scenarios we currently see for CB-MDSD with U-ISC/Graph, there might well be other useful ways to configure the generic CB-MDSD composition systems. This should be subject to future research.

Part III

Related Work, Conclusion, Outlook

10 Related Work

In this chapter, we look at works that relate to ours. We consider works from the fields of Aspect-Oriented Software Development (AOSD) and MDSD. To structure this, the body of work is separated into different categories. The following gives an overview of what these categories cover and why they relate to our work.

- **Aspect-Oriented Programming (AOP)** (Section 10.1) Aspect-Oriented Programming (AOP) [KLM+97] is one of the roots of AOSD. AOP systems extend object-oriented systems. In AOP, the term *aspect* is usually used as we used it in the aspect weaving example (cf. Example 2.2) throughout this thesis. As we have seen from this example, AOP is one possible application for U-ISC/Graph. Therefore, we compare the modularity concepts of AOP with ours in Section 10.1.
- **Aspect-Oriented Software Development(AOSD)** (Section 10.2) Aspect-Oriented Software Development (AOSD) [FECA05] is, in a more general sense than AOP, concerned with separation of concerns and powerful decomposition and composition of software. In AOSD the term *aspect* is used broader as in AOP and may refer to any kind of component that uses a novel modularity concept, which is usually not the dominating modularity concept of the original method or technology extended by AOSD means. CB-MDSD can thus be classified as an AOSD approach and therefore we compare it to other popular AOSD approaches (which are not AOP approaches).
- **AOP-like Approaches for Modelling** (Section 10.3) Works that, similar to ours, bring together ideas from AOSD and MDSD are summarised under the term Aspect-Oriented Modelling (AOM) [AOM10]. Most approaches are published in context of the AOM workshop series [AOM10] and more literature can be found there. AOM approaches can be separated into AOP-like approaches and more general AOSD approaches for modelling. The former, which are treated in Section 10.3, transfer the *aspect* concept of AOP, sometimes with slight extensions, to modelling languages.
- **Other AOSD Approaches for Modelling** (Section 10.4) AOM approaches that are concerned with AOSD in the general sense are analysed in Section 10.4.

- **Generic Model Composition Approaches** (Section 10.5) There are model composition approaches, which are often used for AOM, but which are, similar to U-ISC/Graph, applicable for a variety of composition scenarios and modularity concepts. We look at these approaches in Section 10.5.
- **Model Manipulation in MDSD** (Section 10.6) In Section 10.6, we discuss how MDSD approaches that are not specifically concerned with component-based development, but in a more general sense with the manipulation and integration of models, relate to our work.
- **Process Management in MDSD** (Section 10.7) In this thesis, we also presented two architectural styles for model-driven architectures. In Section 10.7, we discuss related work that is concerned with managing model-driven processes and architectures.

10.1 Aspect-Oriented Programming (AOP)

In this section we look at AOP [KLM+97] in its original form as implemented in Aspect/J [KHH+01].

AOP and AOP-like approaches are focused on AOP component models. That is, they always relay on core and aspect component types and a joinpoint model as component model. This means, the base language has to have object-oriented features as basis for AOP features. Thus, a joinpoint component model of an AOP approach is less flexible as the basic component model of U-ISC/Graph. Nevertheless, the central AOP concepts can be found in U-ISC/Graph in the basic concepts. These parts of U-ISC/Graph were to a certain degree inspired by AOP ideas. We discuss this in the following.

Heterogeneous grouping is found in AOP aspects, which group several advices that are to be applied together. This is similar to grouping several prototypes into a heterogeneous port. Homogeneous grouping is performed by pointcuts, which define where advices are woven into a core. Pointcuts are always defined by the users of aspect systems, while in U-ISC/Graph groupings are defined by the composition system developers in REX_{CM} specifications—although, a developer can give control about the grouping to a user.

Another important property of AOP is obliviousness of the core, which in our terminology means that a core component has only implicit and no declared interfaces. In U-ISC/Graph, a composition system developer can design components with implicit interfaces, declared interfaces, or both.

In AOP, the core is always the target of a weaving (i.e., the extended component) and the aspect is the source of a weaving (i.e., the extending component). This is also called an *asymmetric*, in contrast to a *symmetric*, composition where all components play an equal role. For example, the merge of two components in Hyper/J [OT00] is seen as a symmetric composition. We believe that if a composition is broken down into basic composition operations, for each of these operations one can identify one component as *extending* and one as *extended* component—although these roles can switch in the next composition operation. Thus, also *symmetric* approaches are *asymmetric* in their most basic operations. We follow this in our composition algorithm, where in one composition step we always identify a *contributing* (i.e., extending) and *receiving* (i.e., extended) fragment. Still we used our composition algorithm to perform symmetric composition composition in the Hyper/J sense in ModelSoC.

The most popular AOP tool is Aspect/J [KHH⁺01] that is limited to Java as component language. There is a family of Aspect/J-like AOP tools that transfer the Aspect/J concepts to other component languages. One example is Aspect/C++ [SGSP02]. This tools are, however, all individually hand-crafted. That is, there is no framework supporting the creation of an Aspect/J-like tool for arbitrary object-oriented languages. With U-ISC/Graph, and Reuseware as the implementing framework, we can define AOP-like systems for arbitrary languages.

10.2 Other AOSD Approaches from Programming

This section treats other flexible composition approaches that are not based on the AOP paradigm.

Composition Filters [AWB⁺94] extends the behaviour of objects by message manipulation and is, as AOP, limited to object-oriented languages. However, with Compose* [Sof10], there is a framework as base for composition filter implementations for different languages such as Compose*/Java for Java and StarLight for several .NET languages. Composition filters contains a kind of heterogeneous grouping because a filter can extend the complete object behaviour and not only the behaviour of a single operation. Composition filters was also recently extended with AOP-like pointcuts for homogeneous grouping [BA01].

Role modelling [RG98] extends object-oriented programming and can be used to encapsulate information into roles which are attached to objects to add functionality—making objects extensible with roles. There are different approaches to role modelling that differ in the details (see [Ste00] for an overview). One of the approaches is ObjectTeams [Her02] which is currently only implemented for Java. Thus, role modelling in general is limited to object-oriented languages and ObjectTeams in particular only gives tool support for Java and no framework to support other languages. In contrast to AOP, role modelling uses declared interfaces and is more concerned with heterogeneous grouping in collaborations than with homogeneous grouping.

Compared to the other AOSD approaches, the hyperspace approach [OT00], which we discussed in Chapter 8, is not limited to a specific language or language paradigm. However, it is a conceptual model and the implementation (Hyper/J) is limited to Java and makes other trade-offs such as introducing a dedicated language to define modules (which are called hypermodules in the hyperspace approach).

Since U-ISC/Graph allows us to define implicit and declared interfaces we can use it to construct composition systems that follow role modelling or composition filter principles but are not limited to that. What we cannot provide is flexible composition at runtime as the corresponding implementations ObjectTeams/J [Her02] and Compose* [Sof10] provide, since we only perform static composition.

10.3 AOP-like Approaches for Modelling

In this section, we look at approaches that transfer the AOP concepts to modelling. If not noted differently below, the comparison of AOP with U-ISC/Graph in Section 10.1 also applies to these approaches.

A prominent UML-based AOP-like approach is the Theme approach [CB05] for aspect-oriented analysis and design that works with UML and requirement specifications. However, it is limited to these. Furthermore, it comes with its own specification language for *Themes* (i.e., aspects), which is not exchangeable. Other approaches that are limited to UML are Aspect-Oriented Design Model (AODM) [SHU06], the JAC Design Notation [PDF+02], AOSD with Use Cases [JN04], Motorola Weavr [CvdBE07], AOSD Profile [AEB03], and Aspect-Oriented Architecture Models (AAM) [RGR+06]. A detailed comparison of Theme and these approaches can be found in [SRK+07].

MATA [WJE+09] is an approach that in principle can be applied to any modelling language. It uses graph rewriting with pattern matching as a base for aspect weaving. The distinct feature of MATA, compared to ordinary graph rewriting systems, is, that it allows for specification of patterns in concrete syntax of the languages it supports (e.g., UML sequence diagrams with dedicated stereotypes). Thus, this part of MATA is language dependent and has to be implemented for each supported language manually. So far, the implementation that exists is an extensions of Rational Software Modeler [IBM10b] that supports UML only.

The tool GeKo [MKBJ08] (also used in RAM [KAAK09]; Section 10.4) can be configured by metamodels and thus can be used with arbitrary DSMLs. GeKo is based on Kermeta [Tri10]. In GeKo, the metamodel of the language that should be supported is extended to allow pointcuts to be defined as partial models in the extended language. This is similar to the definition of graph rewriting rules in concrete syntax in MATA [WJE+09] but does not rely on dedicated UML stereotypes. It is also comparable to the grammar extension formalism of U-ISC/Tree (cf. Section 3.2) that is also used to relax the restrictions of the original language grammar to allow the definition of fragments. Furthermore, GeKo has similarities to how new composition language concepts are integrated into languages in U-ISC/Tree, because it individually extends the component language to obtain a pattern definition language for pointcuts (pointcuts are composition language concepts). Thus, in GeKo, the base language is extended with new constructs and composition semantics. It is however not possible to combine different component and composition languages as in U-ISC/Graph.

Both AOP interpretations of MATA [WJE+09] and GeKo [MKBJ08] are more flexible as in traditional AOP in the sense that pointcuts are defined as patterns that match on models. Therefore, pointcuts are not limited to object-oriented language concepts and can consequently be defined for any modelling language. These patterns allow for both heterogeneous and homogeneous grouping of elements. However, the user is confronted with defining the patterns directly on the model structure. Effectively, there is no composition interface. All nodes of the model are exposed to the user. In U-ISC/Graph, the composition system developer can decide how much to expose to the user and can hide parts of a model to ensure that it is not altered during composition. The composition system developer also defines the patterns (which are OCL expressions in REX_{CM} specifications). The user does not have to (but can) be confronted with complex pattern definition.

GenAWeave [Roy08] is a framework for model-driven construction of aspect weavers. It focuses on creating aspect weavers for existing textual programming languages such as Pascal or FORTRAN to support evolution of legacy systems. Nevertheless, it uses metamodellig and model transformation technology to achieve its goal. GenAWeave allows for the construction of dedicated aspect languages for different base languages by designing a metamodel as an extension of a generic aspect metamodel that is provided. Instances of these metamodels are transformed into rewriting rules, using ATL [Ecl10a] model transformations, which can be executed by the Design Maintenance System [Bax92] to perform the actual weaving. GenAWeave thus combines technologies from different areas. It specifically supports aspects in the AOP sense, and is thus only applicable to languages with structures where notions such as *before*, *after*, and *around* advice make sense.

10.4 Other AOSD Approaches for Modelling

In the Reuseable Aspect Models (RAM) [KAAK09] approach, different views on a system are modelled in *aspects* where each aspect contains three views (structural, state, message) modelled with UML class, state, and sequence models. Thus, in contrast to the other UML-based approaches, RAM allows for combined modelling in different UML sublanguages, covering different language paradigms, but is still limited to the languages it supports. RAM is implemented with Geko, Kompose (cf. Section 10.5), and Kermeta [Tri10]. An aspect in RAM can be regarded as a multi-format unit in ModelSoC that supports three different viewpoints. The difference is, that no new viewpoints can be added which hinders the integration of DSMLs or GPLs such as Java.

Another application of AOSD in MDSD is “AOSD with use cases” [JN04]. While it promotes the idea of using aspect technology to decompose system according to UML use cases, it does not offer direct tool support for that and is limited to UML. “AOSD with use cases” is related to our example ModelSoC architecture in Chapter 8 where use case decomposition is one concern dimension. However, ModelSoC supports arbitrary dimensions and the implementation in REUSEWARE gives tool support for automation (not given by [JN04]).

Hovsepyan et al. [HSVB^{+}10] discuss in a case study whether aspect weaving should be performed on models or code. This is motivated by the fact that some approach perform model weaving ([FBFG08, MKBJ08]), while others offer translations to aspect (e.g., Aspect/J) code ([CB05, JN04]). With U-ISC/Graph, weaving can be performed on both models and code since U-ISC/Graph supports any language, graphical or textual, defined with EMOF/Ecore—which does not exclude programming languages. In particular in ModelSoC, weaving is performed with REUSEWARE for any supported viewpoint. While weaving on code level is mandatory to obtain the final system, weaving for other viewpoints can be supported if it aids development.

In all discussed AOP and AOSD approaches, we have not encountered a concept similar to FraCols. While in any composition approach, a connection between component model and composition language is required, this connection is often hard-wired into the approach and cannot be configured for individual composition systems. Only the conceptual hyperspaces approach [OT00] does not give such a limitation. However, the relation between composition operators and components is also hard-coded in the Hyper/J tool.

10.5 Generic Model Composition Approaches

In this section, we look at model composition approaches that, as U-ISC/Graph, are not limited to a specific modelling language and a specific modularity concept.

Kompose [FBFG08] (also used in RAM [KAAK09]) can be configured by metamodels and thus can be used with arbitrary DSMLs. Kompose is, similar to GeKo (cf. Section 10.3), based on Kermeta [Tri10]. It is, in contrast to GeKo which focuses on AOP-like aspects by explicitly supporting pointcut definitions, a generic composition tool for different types of model components. In Kompose, for each language that should be supported, *composers* have to be defined in Kermeta's model transformation language that define what the composition semantics for that language are. For the user of a such defined composition system, Kompose offers the generic composition language *komp* in which compositions can be triggered and additional pre and post composition operations can be defined on the input and the composed models.

Kompose has similarities to our framework approach, which separates the composition system developer and user roles. In Kompose, composition systems specific to a metamodel can be defined by implementing composers in Kermeta's [Tri10] model transformation language. Here, it can be controlled which part of a model is taken into account during composition. Thus, parts of models can be left untouched in a composition, which is effectively information hiding. A kind of composition interface is therefore established. However, this is not as explicit as in U-ISC/Graph, because a generic model transformation language is used instead of a dedicated component model configuration language as we offer with REX_{CM}.

In Kompose, models are always matched as a whole and then composed. Thus, there are no concepts similar to ports or links in *komp*—the complete models are always matched and composed. This is similar to defining graph fragments that group everything in one port. Instead, *komp*, as an imperative model manipulation language, allows for controlling details of the composition through arbitrary adjustment of models. Despite *komp*, there are no other composition languages in Kompose and integration of new composition languages is not supported.

10.6 Model Manipulation in MDSD

A number of model manipulation approaches exist in MDSD which can be used for composition and which are not bound to a specific modelling language. These approaches can, as U-ISC/Graph, be configured by a metamodel of a language.

One important discipline in MDSD is model transformation, as, for instance, offered by the the Query View Transformation (QVT) standard [OMG08] that is implemented, for example, in the Eclipse M2M project [Ecl10b] or the medini QVT tool [ikv10] (cf. [CH03] for an overview of other prominent model transformation approaches). These tools work language independent and are configured by metamodels. With model transformations, models can also be composed by taking separated models as input and creating a new integrated model. Therefore, model transformations can be used to build component support for modelling languages, but with more effort as when using a dedicated approach, like U-ISC/Graph, for that. However, model transformations can be used as a base to implement model composition tools. Kompose [FBFG08], for example, builds on the model transformation language of Kermeta

[Tri10]. We defined U-ISC/Graph with SDM (cf. Appendix A), which can also be regarded as a model transformation technology.

Model weaving, as realised in the Atlas Model Weaver (AMW) [Fab07], is concerned with establishing relationships between models and model elements. It is a generic approach that can be used to represent all kinds of relationships. These can then be interpreted (e.g. by model transformations). Thus, model weaving can also be used to describe composition relationships and execute them by a model transformation. Again, the shortcoming in our context is that it is not specialised for composition and thus building a composition system requires extra effort.

Epsilon [Kol08] for model management is a representative for a framework in MDSD that can be extended with new task-specific languages on the meta level. That is, Epsilon offers dedicated languages for model management and manipulation tasks such as transformation, merging, or validation. In contrast to the above MDSD approaches, the language family of Epsilon can be extended with new languages for special purposes. Thus, it can also act as a platform for realising composition systems and the effort for creating individual systems could be reduced by introducing new task-specific languages dealing with composition.

In summary, generic model transformation, weaving, and management technologies may be used by a composition system developer to build composition systems. Since these technologies and their formalisms are not dedicated for that, as REX_{CM} or REX_{CL} are, it takes more effort to do so as when using U-ISC/Graph directly.

Model transformations also relate to U-ISC/Graph in another way. As discussed already in Section 7.4, REX_{CM} and REX_{CL} can be regarded as domain-specific model transformation languages. This is best observed, when analysing how U-ISC/Graph is used to transfer information from one format into another, which is a task usually done by model transformation in transformation-based model-driven architectures. All model transformation approaches named above, give possibilities to declare rules that consist of three parts: (1) a pattern to match, (2) a template-like structure to produce, and (3) a mapping to insert matched data into the template. These three functionalities are also found in U-ISC/Graph: (1) is the extraction of composition programs, (2) are fragments with variation points, and (3) is the mapping between component and composition language via FraCols. U-ISC/Graph allows, compared to the other approaches, for independent reuse of (1) and the specification of (2) in concrete syntax. We illustrate this on the following example.

Example 10.6. A typical transformation in MDSD is to generate Java classes from UML classes. For example, we could generate the Java version of our file system example (cf. Listing 3.1) from the UML version (cf. Figure 5.14). For this, a kind of template is needed that defines the structure of classes in Java and contains variation points for information that is extracted from the UML model. Traditionally, a code generation engine like MOFScript [Ecl10f] or Xpand [Ecl10e] is used for that, but also model transformations as discussed above can be used if they treat Java, as we do, as textual modelling language.

Templates can be defined as graph fragments with variation points and the code generation can be realised as composition of these fragments. Listings 10.1 and 10.2 show fragments containing a Java Class and a Java Field. The two fragments contain value hooks and slots to fill in information about names and type relationships. By instantiating these fragments

```java
public class CLASS_NAME_VALUE_HOOK {
    // MEMBER_LIST_HOOK
}
```

Listing 10.1: Class template Java fragment.

```java
    protected TYPE_SLOT FIELD_NAME_VALUE_HOOK;
```

Listing 10.2: Field template Java fragment.

multiple times (3x class fragment and 6x field fragment), we can define a composition program that composes Listing 3.1.

The information we capture in this composition program is contained in the UML model of Figure 5.14. If we interpret each UML class as an instantiation of the class fragment (Listing 10.1) and each UML property as well as each UML association as an instantiation of the field fragment (Listing 10.2), we can extract the composition program from the UML model.

With U-ISC/Graph, we can define a composition system that realises this. Listing 10.3 shows the FraCol for this composition system. Based on that, we define composition semantics for the UML metaclasses `Class` and `Association` with REX_{CL} in Listing 10.5 and with that treat UML in this context as composition language. For Java, we define a REX_{CM} in Listing 10.4 to obtain the required composition interfaces of the class (Listing 10.1) and field (Listing 10.2) fragments. Interpreting the UML model in Figure 5.14 as composition program derives a UCL composition program (shown in Figure 10.1) that, when executed, produces the Java classes of Listing 3.1.

10.7 MDSD Process Management

A number of approaches to define MDSD processes and architectures exist. We compare those to our architectural styles for MDSD.

Approaches such as UniTI [VAB$^+$07], MCC [Kle06], TraCo [HKA10], and Megamodelling [BJV04] organise MDSD processes by defining relations between models, metamodels, and transformations. In all these approaches, transformations are a central concept and thus they are suited to realise transformation-based model-driven architectures. ModelSoC and ModelHiC, in contrast, rely completely on composition with U-ISC/Graph as refinement method. Still, ModelSoC and especially ModelHiC could be used as part of a larger MDSD process and integrated with other automatic refinement techniques by one of these approaches. For example, the telecommunication models composed with ModelHiC in Chapter 9 require further processing (e.g., for generating device configurations from them). For this, U-ISC/Graph would act as one tool in a chain of refinement tools. Such a chain can be orchestrated with the approaches named above.

In the introduction, we mentioned an architectural style for MDSD presented by Atkinson et. al [AS08]. Their approach is called *orthographic modelling* and uses, as explained in the introduction, a single underlying model that holds all information about the system and

```
fracol org.reuseware.example.uml2java {
  fragment role Class {
    static port type Name;
    static port type Hook;
    static port type Self;
  }
  fragment role Field {
    static port type Name;
    static port type Content;
    static port type TypeSlot;
  }
  contributing association Extend {
    Field.Content --> Class.Hook
  }
  configuring association Type {
    Class.Self --> Field.TypeSlot
  }
}
```

Listing 10.3: FraCol for a composition system to transform UML to Java.

```
componentmodel org.reuseware.example.uml2java.java
implements     org.reuseware.example.uml2java
epackages      <http://www.emftext.org/java>
rootclass      java::containers::CompilationUnit {
    fragment role Field {
        port type Name {
            java::members::Field.name is value hook { point = $'name'$ }
        }
        port type Content {
            java::members::Field is prototype {}
        }
        port type TypeSlot {
            java::classifiers::Class is slot {}
        }
    }

    fragment role Class {
        port type Name {
            java::classifiers::Class.name is value hook { point = $'name'$ }
        }
        port type Hook {
            java::classifiers::Class.members is hook {}
        }
        port type Self {
            java::classifiers::Class is anchor {}
        }
    }
}
```

Listing 10.4: REX_{CM} for Java as part of a composition system to transform UML to Java.

```
 1 compositionlanguage org.reuseware.example.uml2java.uml
 2 implements          org.reuseware.example.uml2java
 3 epackages           <http://www.eclipse.org/uml2/3.0.0/UML>
 4 rootclass           uml::Model
 5 ucpi = $ufi.trimExtension().appendExtension('fc')$ {
 6   fragment role Class {
 7     uml::Class {
 8       fragment   = $self.name$
 9       ufi        = $Sequence{'UML2Java','java','Class.java'}$
10       target ufi = $ufi.trimExtension().append(self.name.concat('.java'))$
11       port type Name  { $'name'$ = $self.name$ }
12     }
13   }
14
15   fragment role Field {
16     uml::Association {
17       fragment   = $self.ownedEnd->at(1).type.name.concat('.').concat(self.ownedEnd->at(2).name)$
18       ufi        = $Sequence{'UML2Java','java','Field.java'}$
19       port type Name  { $'name'$ = $self.ownedEnd->at(2).name$ }
20     }
21   }
22   association Extend {
23     uml::Association {
24       fragment = $self.ownedEnd->at(1).type.name.concat('.').concat(self.ownedEnd->at(2).name)$
25       -->
26       fragment = $self.ownedEnd->at(1).type.name$
27     }
28   }
29   association Type {
30     uml::Association {
31       fragment = $self.ownedEnd->at(2).type.name$
32       -->
33       fragment = $self.ownedEnd->at(1).type.name.concat('.').concat(self.ownedEnd->at(2).name)$
34     }
35   }
36
37   // repeat Lines 15-35 with 'self.ownedEnd->at(1)' and 'self.ownedEnd->at(2)' exchanged
38 }
```

Listing 10.5: REX$_{\mathrm{CL}}$ for UML as part of a composition system to transform UML to Java.

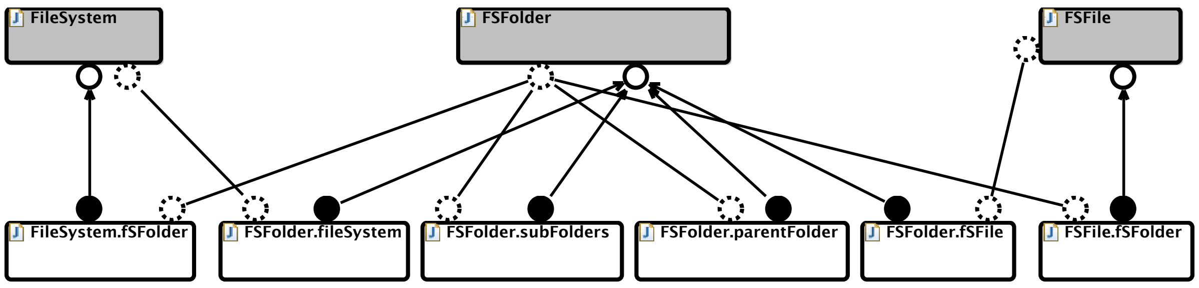


Figure 10.1: The composition program derived from Figure 5.14 using the REX$_{\mathrm{CL}}$ from Listing 10.5.

treats the models developers work with—including the implementation code—as views on the underlying model. They use bi-directional transformations for creating models as views and reintegrating models into the single underlying model. From the user viewpoint, orthographic modelling is not so different to ModelSoC. However, setting up an architecture with orthographic modelling is complex because the bi-directional transformations have to be defined in model-transformation technologies as the ones discussed in Section 10.6. In particular, defining the transformations from the views to the single underlying model is challenging since all these transformations work on the same target model and potentially need to know what all other transformations do with it. In ModelSoC, we attempt to avoid exactly this complexity by explicitly splitting the system into different concern dimensions and using UCL as a minimal language to present the system architecture.

11 Conclusion

The main goal of this thesis is to introduce concepts from component-based development into model-driven software development to improve separation of concerns and therewith consistency and tracing in model-driven architectures. To achieve this, we presented the universal model composition technique U-ISC/Graph and a generic composition system for MDSD that can be configured to work with arbitrary modelling languages (Part I). U-ISC/Graph and the generic composition system were implemented in our model composition tool REUSEWARE (Chapter 7). As guidance for using the technique and the tool in MDSD, we proposed two architectural styles for setting up component-based model-driven architectures by configuring the generic composition system. Both styles are based on recognised component-based development paradigms: ModelSoC (Chapter 8) realises multi-dimensional separation of concerns for MDSD and ModelHiC (Chapter 9) realises hierarchical composition for MDSD.

In the introduction we claimed three major contributions (C1, C2, and C3). These major contributions entail nine contributions we formulated in the conclusion sections of Chapters 4–9. An overview of these contributions as well as their relationships between each other and to existing work is given in Figure 11.1. The figure has the same structure as Figure 1.4 that gave an overview of the document structure in the introduction. In the following, we summarise the contributions and group them according to the major contributions.

Contribution 1 (C1): Composition Technique for Typed Graphs

C1-1 **Contracts between component and composition languages** Our solution for a model composition technique is based on the concept of *composition systems* and thus our aim, which we motivated in Chapter 3, was to develop a generic composition system for MDSD that can be configured with multiple component and composition languages. For this, we identified the need to define contracts that have to be fulfilled by component and composition languages when they are integrated into a composition system. These contracts needed to support aggregation to allow for the extension of a composition system with new contracts.

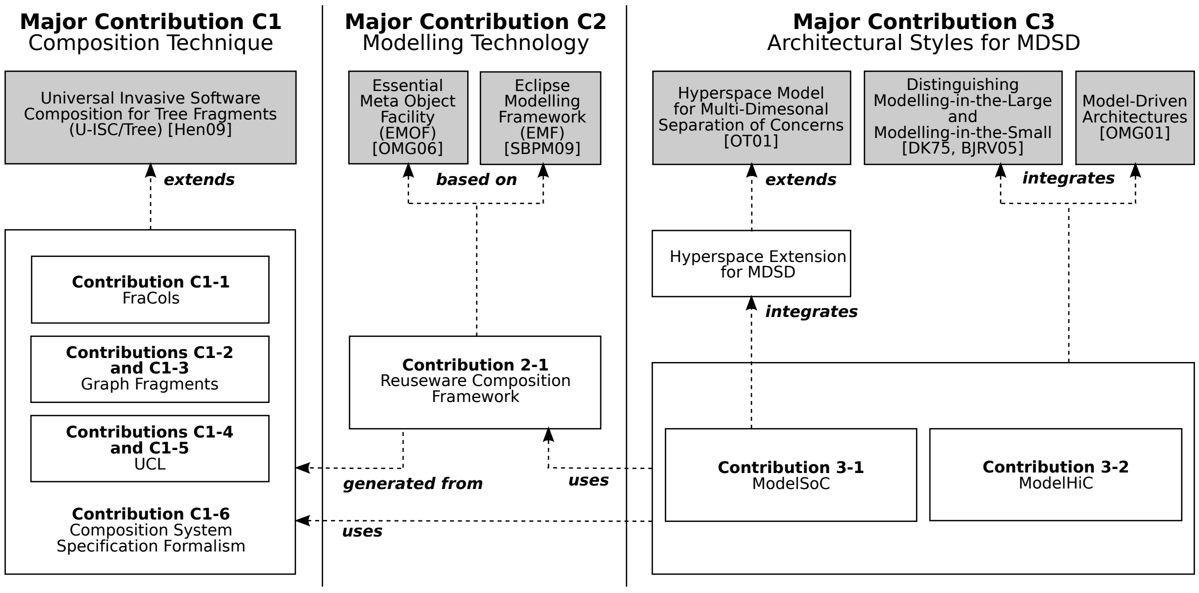

Major Contribution C1
Composition Technique
Universal Invasive Software Composition for Tree Fragments (U-ISC/Tree) [Hen09]
extends
Contribution C1-1
FraCols
Contributions C1-2 and C1-3
Graph Fragments
Contributions C1-4 and C1-5
UCL
Contribution C1-6
Composition System Specification Formalism
Major Contribution C2
Modelling Technology
Essential Meta Object Facility (EMOF) [OMG06]
Eclipse Modelling Framework (EMF) [SBPM09]
based on
Contribution 2-1
Reuseware Composition Framework
generated from
uses
uses
Major Contribution C3
Architectural Styles for MDSD
Hyperspace Model for Multi-Dimesonal Separation of Concerns [OT01]
Distinguishing Modelling-in-the-Large and Modelling-in-the-Small [DK75, BJRV05]
Model-Driven Architectures [OMG01]
extends
integrates
Hyperspace Extension for MDSD
integrates
Contribution 3-1
ModelSoC
Contribution 3-2
ModelHiC

Figure 11.1: Overview of our contributions.

Our contribution here is the *fragment collaborations* (FraCols) concept defined in Chapter 4. The concept is based on role modelling which ensures the aggregatability of FraCols. With FraCols, properties of modularity concepts can be defined independent of concrete component or composition languages. It is the first concept of this kind for invasive software composition systems. It is one of the three foundations of U-ISC/Graph (the other two being C1-2 and C1-4). And it is a prerequisite for C1-3, C1-5, and C1-6.

C1-2 **Type-safe invasive composition interfaces for graphs** Models in MDSD are graphs with certain properties that we discussed in Section 5.1. To treat models as components, we thus required a composition technique that works with these kinds of graphs. In Chapter 3, we decided to extend the tree merging composition technique of U-ISC/Tree, because this technique is language-independent and type-safe, which are both properties we required. The first concept we needed to transfer to graphs was the variation point concept for composition interfaces.

In Section 5.2, we therefore introduced *variability typing* for *graph fragments* that extends the variation point concept of U-ISC/Tree. The extension covers the properties of model graphs that trees do not have. Our graph fragment variability typing is the first concept for invasive composition interfaces that give access to graph structures. It is the second foundation of U-ISC/Graph and a prerequisite for C1-3.

C1-3 **Abstraction of physical composition interfaces** To make a connection between the physical composition interfaces of graph fragments and FraCols (C1-1), we needed to abstract from the physical interfaces available through the variability typing (C1-2).

We therefore introduced *grouping concepts* for variation points in Section 5.2. These allow to form groups of variation points that concern one modularity concept which has been defined in a FraCol. Furthermore, we defined the component model configuration language REX_{CM} in Section 5.3, which allows to bind modularity concepts defined in FraCols to the metamodel of a modelling language that should act as component language. Through these bindings, variability typings of models are defined as well as mappings of composition interfaces to FraCols. The grouping concepts for variation points are the first abstraction mechanisms of this kind for invasive composition interfaces. Furthermore, before REX_{CM}, ISC did not offer any method for extending languages to component languages without modifying a language's metamodel. This contribution is a prerequisite for C1-6.

C1-4 **Basic invasive composition operators for graphs** In addition to the variability typing, the basic composition operators of U-ISC/Tree needed to be extended for graph composition, because they are the basis of the composition technique. Our extension to the basic operators needed to take the properties of graphs and the new graph variability concepts (C1-2) into account.

We defined the operators in terms of graph rewriting rules with story driven modelling in Section 6.1. We introduced a new basic delegating operator that decides which concrete operator to call based on the nature of the variation point it is applied to. This way, operators do not have to be called explicitly anymore in composition programs, as it was the case in ISC so far, but can be automatically selected, which is a prerequisite for C1-5. The basic composition operators are the third foundation of U-ISC/Graph.

C1-5 **Universal invasive composition language and algorithm** We needed a mechanism to integrate arbitrary languages as composition languages into composition systems. For that, we required a common format for composition programs as bridge between arbitrary composition languages, modularity concepts defined in FraCols (C1-1), and basic composition operator calls (C1-4).

As such common format, we defined the declarative *Universal Composition Language* (UCL) with a minimal set of concepts in Section 6.2. Composition programs formulated in UCL can be translated to basic operator calls by the universal composition algorithm defined in Section 6.3. Both UCL and the composition algorithm are independent of a concrete component or composition language. Component languages are integrated by using REX_{CM} (C1-3) to define how a concrete model is accessed and which parts of such a model are passed to the basic operators for modification. Furthermore, to treat an arbitrary modelling language as composition language, we define the REX_{CL} language for composition language integration in Section 6.5. REX_{CL} maps a metamodel of a modelling language to modularity concepts defined in FraCols and with that expresses how a UCL composition program is extracted from models defined in that modelling language. Thanks to the declarative nature of UCL, which does not define an order in which elements of a composition program needs to be interpreted, UCL programs can be combined by extraction from multiple sources. UCL is the first declarative composition language for ISC and REX_{CL} the first method to integrate arbitrary composition languages into composition systems. This contribution is the last prerequisite for C1-6.

C1-6 **Declarative composition system specification formalism** Our goal was to develop a formalism to configure a generic composition system that can be used for MDSD.

FraCols (C1-1), REX_{CM} (C1-3), and REX_{CL} (C1-5) in combination with EMOF/Ecore yield this formalism. Modelling languages are defined in EMOF/Ecore and modularity concepts are defined in FraCols. With REX_{CM} and REX_{CL}, the modelling languages are mapped to the FraCols, either as component or composition languages. Since FraCols can be combined with very few restrictions, such composition systems can be extended, even if they are already in use, with new modularity concepts as well as new component and composition languages. This is the first formalism for defining invasive composition system with this degree of flexibility.

Contribution 2 (C2): Composition Framework for Models

C2-1 **Universal model composition framework** to transfer our results into practice and evaluate them, a tool is required that implements U-ISC/Graph with the generic composition system for MDSD (C1) and its configuration formalism (C1-6).

We provide this tool, which is called Reuseware, and described it in Chapter 7. The core of the tool is generated from the semantics specifications of U-ISC/Graph that we provide as story diagrams in Appendix A. Reuseware is the first ISC tool that is based on a standardised metalanguage (EMOF/Ecore) and offers the high flexibility of integrating arbitrary languages as both component and composition languages.

Contribution 3 (C3): Component-Based Model-Driven Architectures

C3-1 **Architectural style for MDSD with multi-dimensional separation of concerns** To fully combine the advantages of MDSD and component-based development, we needed an architectural style for MDSD that supports the advantages of both.

We developed such a style, called *ModelSoC*, based on the multi-dimensional separation of concerns paradigm. For this, we extended the hyperspace model for multi-dimensional separation of concerns to meet the requirements of MDSD. ModelSoC uses the generic composition system for MDSD to its full extend and thus can be used to define model-driven architectures with arbitrary many modelling languages and modularity concepts (called concern dimensions in multi-dimensional separation of concerns). With ModelSoC, we contributed a style to define model-driven architectures in which the complete architecture of a system is reflected in one UCL composition program and where concerns are separated in a unified way throughout all abstraction levels and viewpoints. This ensures that all models are kept synchronised and eases traceability, while it preserves the property of MDSD to integrate arbitrary DSMLs into a model-driven architecture.

We evaluated ModelSoC and with it U-ISC/Graph. For this we used Reuseware to set up a model-driven architecture with the ModelSoC style and developed a demonstrator system with it. Since, the architecture was designed to reflect properties found in many model-driven architectures, we can generalise from it and conclude that ModelSoC and therewith U-ISC/Graph is applicable for a number of similar MDSD scenarios.

C3-2 **Architectural style for MDSD with hierarchical composition** While ModelSoC is powerful, it can also be complex to use when many DSMLs and modularity concepts are involved. We also identified model-driven architectures for which a simple hierarchical composition suffices.

For such architectures, we defined a lightweight component-based architectural style for MDSD called *ModelHiC*. This style can be applied to realise abstraction in a model-driven architecture that is centred around a single DSML and in which abstract modelling is performed by defining a hierarchy of UCL composition programs. This spares the effort of integrating different DSMLs for different abstraction levels. ModelHiC thus provides a restricted, but lightweight, way for model-driven software development.

We evaluated ModelHiC and with it U-ISC/Graph. For this we used REUSEWARE to develop abstraction support for one complex DSML with ModelHiC. Since the DSML we chose had properties which are typical conditions in practice, we can generalise from it and conclude that U-ISC and ModelHiC are applicable in a number of similar MDSD scenarios.

12 Outlook

In this chapter, we discuss open issues and give directions of future work. This includes issues that we researched during the work on this thesis but did not present so far.

There are important issues related to our work on invasive composition systems for MDSD that are not part of U-ISC/Graph, but rather build on top of U-ISC/Graph. Still, these issues are motivated by requirements for the practical use of components in MDSD and thus we were aware of them from the beginning of our work. In particular to perform the evaluations in Chapters 8 and 9, we needed to address some of these issues at least partially and integrate them in Reuseware to provide user friendly tooling. In the following sections, we introduce the different issues, summarise the initial work done so far, point at our publications about this work, and propose future directions for continuing this work. If a part of work was done in collaboration with others, we indicate this.

12.1 Layout Composition Support

We discussed in Section 5.1.2 that models with graphical syntax are augmented with layout information that has to be managed and stored in addition to the models. The layout contains information about where shapes and lines that represent parts of a model are positioned on the screen. If we compose models that have additional layout information with U-ISC/Graph, this information is lost in the composed model, because it is not part of the original graph fragments. This is an undesirable situation, since preserving the layout information is vital to understand a composed model and relate it mentally to the fragments it was composed from. Furthermore, layout information of the involved composition programs can also be of interest. Thus not only preserving, but also merging the layout from different sources is an issue.

We addressed this issue together with Karsten Gaul in [JG09] and [Gau10]. In [JG09] we considered layout preservation and composition in U-ISC/Graph composition systems that directly integrate UCL. There, the layout information created in the GMF editor for UCL (cf. Section 7.3) was taken into account. The tool support that resulted from this work was used in the evaluation of ModelHiC in Chapter 9. In [Gau10], the layout composition work was extended to support composition systems with arbitrary composition languages. That is, combining layout information from models defined in different component and composition languages using different graphical and textual syntaxes. The result of this work was used in the evaluation in Chapter 8.

In summary, in [JG09] and [Gau10] we developed an extensible layout composition framework called *LaCoMe* (Layout Composition Framework). Reuseware calls into LaCoMe after executing a composition step (cf. Section 6.3) and passes the model fragments with their layout information, the composition program with its layout information, and the composed model to LaCoMe. Furthermore, it gives LaCoMe access to the trace information of the `SyncEcoreUtil` tool (cf. Appendix A.3.3) which allows LaCoMe to identify certain nodes as copies of others. LaCoMe uses this information to create a layout for the composed model that preserves as much information as possible of the original layouts. It also can apply adjustment algorithms to remove overlaps of shapes that result from the merge of several layouts. Since there is no established standard for presenting layout information (cf. discussion in Section 5.1.2) LaCoMe is extensible to support new layout formats. Currently the formats used by GMF [Gro09] and TOPCASED [TOP10] are supported which are the formats we used in the evaluations. LaCoMe offers another extension point for adjustment algorithms since it turned out that there are different possibilities to remove overlaps and that it highly depends on the concrete kinds of diagrams and models which one to select. For more details please consult [JG09] and [Gau10].

Although it improved the invasive composition of graphical models a lot, our work on layout preservation and composition is still only a first step. During the work, it became clear that different adjustment strategies are required for different types of diagrams which makes the development of language independent solutions challenging. Although [Gau10] provides a classification of diagram layouts, this classification has yet to be put into practice to automate the selection of adjustment algorithms. So far, a user has to do manual configurations to select the best suited adjustment for the task at hand. This becomes a tedious task if many different languages and models are involved in a development process as in Chapter 8. The results of a user study on the usefulness of LaCoMe in the model-driven architecture of Chapter 8, which was performed as part of [Gau10], underpin this. Since there is limited work on layout preservation in MDSD (in the context of model transformations, only Pilgrim et al. addressed the issue in [vP07, vPVSGB08]), we belief that continuing this work is important not only for the usability of U-ISC/Graph but for MDSD in general.

12.2 Round-trip Support

Composed models are in many cases meant to be inspected by users and are not, as for instance compiled code, only processed by machines. That is the reason why layout preservation, discussed above, is an important issue. Another issue that is also motivated by the fact that users work with composed models, is round-trip support for editing these models. That is, users should be able to modify composed models, for example when they find a mistake that was not recognised in the fragments from which the model was composed. Such modifications should be propagated back into the original fragments or the composition program in a way that recomposing the composed model preserves the changes. If this is not possible, the user should at least be informed about that when he or she tries to modify a composed model. Currently, without round-trip support, users can open and edit composed models, but changes will soon be lost when the model is recomposed after one of the underlying fragments or composition programs changes.

We started to work on round-trip support for U-ISC/Graph together with Mirko Seifert [JSS09], who works on round-trip systems in general, to understand how round-trip can be supported in U-ISC/Graph and which concepts particular to U-ISC/Graph can be exploited for that. In this context, a prototype was developed by Mirko Seifert which was used to extend the security modelling in the ModelSoC architecture of Chapter 8 with round-trip features.

In general, we learnt that round-trip support can be provided easily for the parts of graph fragments that are *hidden* behind composition interfaces (cf. Section 5.1.3). These parts are not modified during composition and appear in the composed model as exact copy of the original graph fragment part. If a user modifies the copy, the same modification can be automatically applied to the original. Problems can occur when a fragment is reused in multiple places. Then it is not clear if the user intents to modify only one or all copies of the reused fragment. For this, more information about the user's intent is required. This information can be obtained by asking the user interactively each time such a situation occurs or by providing strategies that guess the user's intention. In one concrete composition system, the users might always expect a certain behaviour given by the nature of the system. For example, for an aspect fragment, as used in examples in this thesis, it can be expected that all occurrences of the aspect are always intended to be modified together. Certain intents might require the modification of composition programs and the fragment repository. For instance, if only one occurrence of a fragment is modified, a duplicate of the fragment for that one occurrence needs to be created and the composition program needs to be changed to use the new duplicate in the future. If changes to elements that influence the composition interface are made, this can change the whole composition and have unexpected impact on the composed model. The current solution produces a preview of the impact of the change and allows the user to undo the change. So far we only consider U-ISC/Graph composition systems that use UCL as composition language directly. For more details please consult [JSS09].

In the work so far, the focus was on fragments and composition interfaces and only partially on composition programs. In the future, it should be investigated how more changes can be automatically back propagated to composition programs and how the UCL (cf. Chapter 6) and FraCols (cf. Chapter 4) concepts can help with that. To take this a step further, extending the round-trip support for models that are interpreted as composition programs, is of high interest. In particular, ModelSoC, as presented in Chapter 8, would profit from this immensely since it would allow editing of the generated views.

12.3 Fragment Library System

Once a composition system is in use and fragments are developed and reused, it is important to organise the fragments in a library and provide means to search for suitable fragments in it. Also composition programs and even composition systems themselves should be organised and searchable. In the end, only the ability to find fragments that are useful for the task at hand makes these fragments reuseable for a user.

We developed together with Matthias Schmidt a library system for models in [SPJF10]. The system is based on faceted classification and browsing, which is a popular method for searching in many online systems (e.g., in Ebay). The library system was implemented as an extension of REUSEWARE that includes a visual fragment browser for Eclipse. We evaluated the tooling in [SPJF10] using the fragments from the evaluation in Chapter 9.

Faceted classification is a concept from book libraries which was introduced by Ranganathan in the 1930s. Príeto-Diaz [PDF87, PD91] was the first who proposed its usage for software component libraries. In a faceted classification, not an object as a whole, but different aspects (i.e., facets) of an object are described. For example, well-known facets used in book libraries are author, topic, and publisher. A set of facets define a controlled vocabulary that is used for classification. In our approach, a facet developer, who is possibly also composition system developer, defines facets that are suitable to describe fragments of a certain composition system. The composition system user, who defines fragments, can classify them by using the vocabulary defined by the facets. Another user can use the facet-based browser in Eclipse to interactively browse the library of classified fragments. Instead of classifying all fragments manually, a facet can also come with a query defined in OCL that derives the value for that facet for a fragment automatically by inspecting the fragment. All details are described in [SPJF10].

The critical point of the faceted browsing system is that all fragments need to be sufficiently classified. If this is done manually, there is danger of incomplete, wrong, or outdated classification. Therefore, the possibility of automatic classification is important and should be further investigated. One way to ensure sufficient classification would be to integrate it deeper into the composition system development process. On the one hand, the composition system developer could be urged to define facets with automatic classification rules. On the other hand, the composition system users could be forced by a tool to define certain facets when they register fragments with the repository. Apart from fragments, also FraCols, REX_{CM}, and REX_{CL} definitions could be classified. This is in particular important for highly customised composition systems with many small specifications as used in ModelSoC (e.g., the specifications of the architecture defined in Chapter 8 collected in Appendix B). If one wants to set up a new model-driven architecture with ModelSoC, it is important to search for existing specifications that can be integrated instead of defining everything from scratch. For this, a dedicated set of facets could be defined. In the future, larger case studies need to be performed to evaluate and improve facets and classification methods further.

12.4 Easing Composition System Development

Developing a composition system for MDSD is a language engineering, or metamodelling, task. It is a complex task, since the developer has to understand the concepts of U-ISC/Graph and learn the specification languages FraCol, REX_{CM}, and REX_{CL}. In other language engineering disciplines such complexity also exists. For example, developing a graphical syntax with GMF [Gro09] (cf. Section 5.1.2) is difficult to learn because GMF also has rich and complex specification languages. The EuGENia [KRPP09] approach simplifies that by abstracting from the specification languages in GMF. Instead, it offers a much more compact formalism based on metamodel annotations. Instead of forcing the user to define all details, EuGENia assumes default values for places where nothing has been specified. Although GMF allows much more configurability, EuGENia is considerably easier to learn and use and still allows for most functionality. In a similar way, we could abstract from our specification languages to offer a limited but easier to learn way to specify composition systems.

In [JZF^{+}09] we developed, together with the developers of EuGENia, an approach to define composition languages by means of metamodel annotations (instead of using REX$_{\mathrm{CL}}$). Although this approach is more limited, it is simpler to learn and use and is particular suitable to combine DSMLs in a composition system that are located in the same domain as we discussed in [JZF^{+}09].

Technically, we realised the annotation based approach by generating an ETL model transformation [Kol08] from annotations that can transform models into composition programs. Thus, instead of using REX$_{\mathrm{CL}}$ with the `extractCompositionProgram()` operation (cf. Appendix A.5), we directly transform models into composition programs with the generated transformation. For this the transformation is registered with Reuseware using the CompositionProgramExtractor extension point defined in Section 7.4, where we already discussed such a way of using model transformations instead of REX$_{\mathrm{CM}}$ or REX$_{\mathrm{CL}}$ specifications in more detail.

In the future, further possibilities for easing development of invasive composition systems should be investigated. New formalisms and languages can be developed with metamodelling and model transformation technologies. In particular, we expect that abstract formalisms can be found for specific categories of composition system—for instance, composition systems used for ModelSoC (cf. Chapter 8). Tools more specialised for composition system development could also be imagined. Instead of defining a textual specification, one could also think of graphical tools or a tool based on wizards.

12.5 Combining Composition with Rewriting

Another interesting direction of future work we have not yet worked on is the combination of composition with rewriting. U-ISC/Graph is only concerned with composition of graphs, but not with rewriting them internally. However, at the point in a model-driven architecture where platform specifics need to be taken into account, rewriting can become important to do optimisation (e.g., tuning source code fragments). An interesting question is how composition and rewriting can be combined.

In fact, the layout adjustment discussed in Section 12.1 can be seen as such a rewriting. After a composition step, information that is connected with the graph but does not influence the composition is rewritten to meet some optimisation goal—in that case an optimal diagram layout. Similar, internals of a composed fragment can be rewritten—for example to optimise execution speed. Thus, we could generalise the layout composition (which is performed in addition after a composition step) to general *composition post processing* that may perform other kinds of rewriting after a composition step. If such a rewriting can ensure that it only changes internals but not the composition interface of the composed fragment, it would not interfere with the composition algorithm. An interesting point is that rewritings cannot only be performed on single fragments or on the complete composition result, but also after intermediate composition steps on intermediate composition results. It is an interesting research question if, and for which kinds of optimisations, this brings advantages.

12.6 Integrating U-ISC/Graph into other Tools

In Part I of this thesis we conceptually developed U-ISC/Graph and in Chapter 7 we presented Reuseware as an implementation of it. The concepts of U-ISC/Graph could be transferred to other tools as well. For example, one could imagine composition support based on U-ISC/Graph as a new component in the mentioned Epsilon model management platform [Kol08].

To realise this, the U-ISC/Graph specifications in EMOF models and story diagrams (cf. Appendix A) could be used by alternative code generators to generate a U-ISC/Graph composition engine that respects the APIs of the tool into which U-ISC/Graph should be integrated. Another implementation possibility is to formulate the U-ISC/Graph semantics in other model transformation languages or to generate model transformation scripts from the story diagrams.

Part IV

Appendix

A U-ISC/Graph Semantics defined with Story Driven Modelling

A.1 Introduction to Story-Driven Modelling (SDM)

This appendix contains the semantics definitions of U-ISC/Graph that are specified with the Story-Driven Modelling (SDM) formalism, to which we give a short introduction in the following. SDM [FNTZ00] extends UML activities [OMG10b, Chapter 12] to *story diagrams.* Story diagrams can be seen from the modelling as well as from the graph rewriting perspective. From the modelling perspective, they introduce a specific kind of action—the *story pattern action*—into activity diagrams. From the graph rewriting perspective, the SDM formalism is a graph matching and rewriting language following the ideas of graph grammars.

As stated in [FNTZ00], story diagrams were developed to improve the integration of graph grammars with design and implementation languages such as UML and Java. Consequently, the story diagram implementation found in Fujaba integrates into UML activity diagrams. From such activity diagrams executable Java code is generated by Fujaba.

We will not discuss in detail how story diagrams are translated into Java here, but describe the semantics of story diagrams informally. We limit this description to the elements of story diagrams we use in this thesis.

A.1.1 SDM Concepts

SDM is an extension of UML activities [OMG10b, Chapter 12]. A UML activity can be used to define the behaviour of an operation—in the Ecore case an `EOperation` (cf. Figure 5.6)—with a workflow. Hereby, a set of *actions* and a flow between them is modelled. In an abstract model, an action might only contain a description in natural language that describes what is done inside the action. In a more concrete model, an action needs to be linked to some other model or code that exactly describes the semantics of the action—usually in terms of reading and modifying the state of the modelled/implemented system.

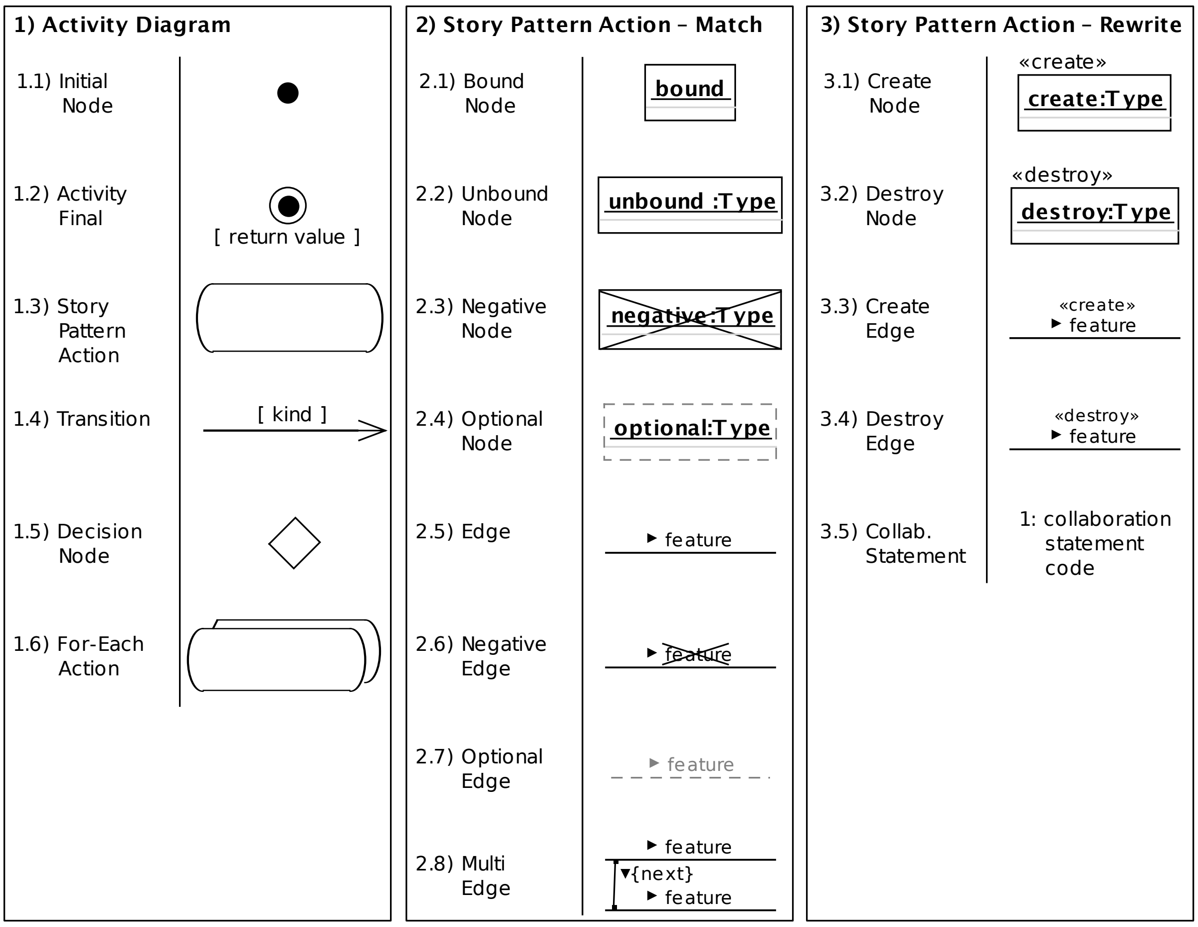


Figure A.1: Graphical story diagram notation.

In the case of the metamodelling language Ecore, where the `EOperations` and consequently the actions within describe language semantics, an exact description of semantics is desired. Ecore itself does not contain a formalism for this by default. Although UML does offer formalisms for behaviour modelling, it also allows adding additional formalisms to describe action semantics if needed.

SDM is such a formalism that allows the specification of action semantics in terms of graph rewriting rules—called *story patterns*. Since we describe semantics for graph modifications and compositions, the SDM formalism suits our purpose. In the following, we describe the details of UML activities and SDMs directly on their graphical notation which we will use throughout this thesis.

A.1.2 Graphical Notation

The elements of story diagrams and their graphical notations are shown in Figure A.1. They can be separated into three parts: (1) The activity diagram part that is UML standard notation [OMG10b, Section 12.4]. (2) The part of story pattern actions that assemble the left side (matching part) of a graph rewriting rule. (3) The part of story pattern actions that assemble the right side (rewrite) part of a graph rewriting rule.

In SDM, the activity diagram notation acts as *explicit rule control language* that controls the order in which graph rewriting rules are applied. An activity is always invoked on a node of a model that is an instance of the `EClass` containing the corresponding `EOperation`. Furthermore, if the `EOperation` declares `EParameters`, other nodes are passed as arguments to the activity. All rules operate—match and rewrite—on the graphs that the invoked-on node and the argument nodes are nodes of. Other nodes of these graphs can be *bound* to named variables that can be referenced by other rules. Thus, all rules (story patterns) of a story diagram operate in a common context and build upon each other.

Activity Diagram (Figure A.1 (1)) The following activity diagram concepts are used in this thesis: Each activity starts with an *initial node* (1.1) and ends with one or more *activity finals* (1.2). If the `EOperation` has a return type, the activity finals have to be annotated with a node of the corresponding type that is returned as a result of the activity. The main parts of an activity are the *story pattern actions* (1.3). The order in which the actions are executed is specified by *transitions* (1.4) between the actions. An action will either succeed (pattern matches at least once) or fail (pattern does not match). Therefore, an action can have two outgoing transitions where one is annotated with `[succeed]` and one with `[failure]`. In addition to story pattern actions, *decision nodes* (1.5) are supported. A decision node always has two outgoing transitions, where one is annotated with a boolean expression over the nodes currently bound in the activity and one with `[else]`. A special action is the *for-each action* (1.6) that is executed for every match (and not only for one match). A for-each action always has an outgoing transition `[end]`. It may also have an outgoing transition `[each time]` that points to a follow-up pattern that is executed for each match.

Story Pattern Action – Match Part (Figure A.1 (2)) Inside a normal or an for-each story pattern action, the following elements are used for graph matching: In general, a story pattern consists of a set of nodes[1] notated in UML object diagram notation [OMG10b, Chapter 7], which is `node:EClass` (where `node` is an instance of `EClass`). *Bound nodes* (2.1) are the starting points of each match. They need to be already bound in the context of the activity. Always available is the special bound node `this` that is the node the activity is invoked on. The arguments of the operation are also available as bound nodes. *Unbound nodes* (2.2) are the parts of a rule that are bound during matching. After a successful match, these nodes are available as bound nodes in successive rules. Unbound nodes can be negative (2.3), which means that the rule only succeeds if the node cannot be bound. They can also be optional (2.4), which means that the node is bound in case of a match, but would not let the rule fail if the binding is not possible.

Nodes are connected via *edges* (2.5) that are instances of `EReferences` (cf. Figure 5.6). An edge can therefore only be drawn between nodes that are instances of `EClasses` that are connected by the corresponding `EReference`. A set of connected nodes needs to contain at least one bound node as a starting point for the matching. Similar to nodes, edges can be negative (2.6), or optional (2.7). Another required construct is the *multi edge* (2.8). Edges that are instances of the same `EReference` can be part of an edge list—if the `upperBound` of

[1]In the original UML and SDM terminology *nodes* are called *objects* and *edges* are called *links*. We stick to the graph terminology and therefore use the terms *node* and *edge* consistently.

that `EReference` is greater than one. With the multi edge concept, we can take the order of edges in an edge list into account for matching and rewriting.

Story Pattern Action – Rewrite Part (Figure A.1 (3)) After a rule matches, rewrite operations can be performed. These are creations or destructions of matched nodes or links (3.1–3.3). Additionally, so called *collaboration statements* (3.5) can be formulated. In general, these statements can be used to call into arbitrary existing code. In our case, we use collaboration statements for two purposes. First, they are used to call other story patterns. Second, they are used to call standard library operations on primitive data types such as string. The syntax of these statements is close to Java statement syntax. The statements are executed in an order defined by the leading number (`1:` in Figure A.1 (3.5)). The collaboration statements of a story pattern action are only executed if the matching succeeds.

We gave a brief overview of the SDM graph rewriting formalism and its syntax above. This may be used as a reference to support the understanding of the diagrams which are presented in this appendix. For further details about SDM please refer to [FNTZ00] and the Fujaba website `http://www.fujaba.de`.

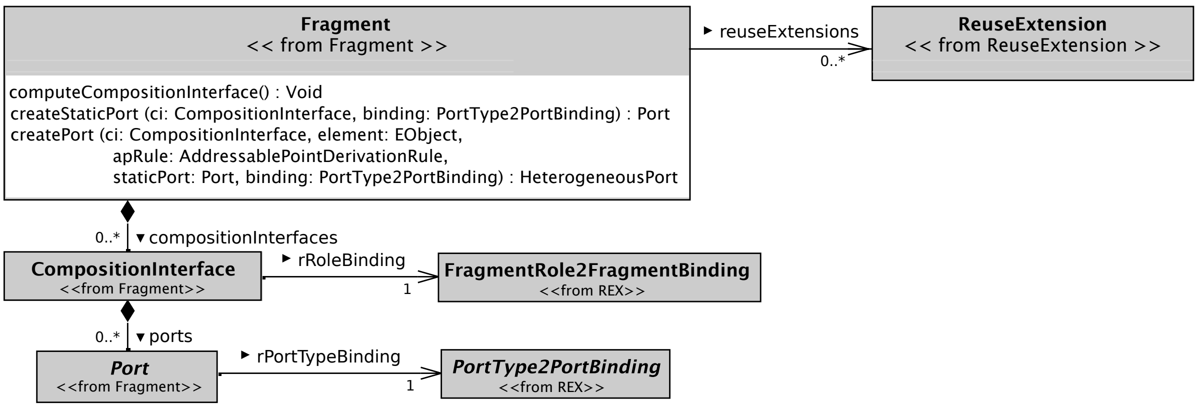

Fragment
<< from Fragment >>
computeCompositionInterface() : Void
createStaticPort (ci: CompositionInterface, binding: PortType2PortBinding) : Port
createPort (ci: CompositionInterface, element: EObject,
apRule: AddressablePointDerivationRule,
staticPort: Port, binding: PortType2PortBinding) : HeterogeneousPort
reuseExtensions
0..*
ReuseExtension
<< from ReuseExtension >>
0..*
compositionInterfaces
CompositionInterface
<<from Fragment>>
rRoleBinding
1
FragmentRole2FragmentBinding
<<from REX>>
0..*
ports
Port
<<from Fragment>>
rPortTypeBinding
1
PortType2PortBinding
<<from REX>>

Figure A.2: Relationships between fragment (cf. Figure 5.21) and reuse extension (cf. Figure 5.27) metamodels.

A.2 Composition Interface Computation

This section defines the semantics for composition interface computation based on REX_{CM} specifications described in Section 5.3.

Figure A.2 depicts the relationships between the composition interface (cf. Figure 5.21) and the REX_{CM} (cf. Figure 5.27) metamodels. The reference `reuseExtensions` is used to link a fragment to the reuse extensions that define how its interface is computed. The created composition interface will link to the fragment role and port type bindings that are responsible for its creation. For this, the references `rRoleBinding` and `rPortTypeBinding` are needed.

`computeCompositionInterface()` is depicted in Figure A.3. In principle it iterates over the `ComponentModelSpecifications` that are reached via `reuseExtensions`. Then it attempts to apply each rule specified in the current specification to each node of the fragment. It makes use of the additional operation `createStaticPort()` (Figure A.4) and `createPort()` (Figure A.5). In detail, it works as follows.

1. All composition interfaces created in previous `computeCompositionInterface()` calls are removed.
2. The following patterns are performed for each REX_{CM} specification that is reached via `reuseExtensions`. To distinguish different composition interfaces for different fragment roles, one composition interface is created here for each fragment role bound in the REX_{CM} specifications. To decide whether to continue to Pattern 3, we utilise an external Evaluator (cf. Section 7.4) that can evaluate OCL expressions by calls into the MDT OCL interpreter [Ecl10c]. The `eval()` operation evaluates towards a boolean value. It accepts a UFI (i.e., a list of strings), an `EObject` (or a list of `EObjects` as in this case) and an OCL expression (as plain string) as arguments. It returns *true* if evaluating the expression on one of the `EObjects` returns *true* or if the expression is empty. Thus, in this case, it evaluates to *true* if the `isExpression` is satisfied for one of the root elements of the model (`contents` reference) or if no expression is specified. In that case, we continue.

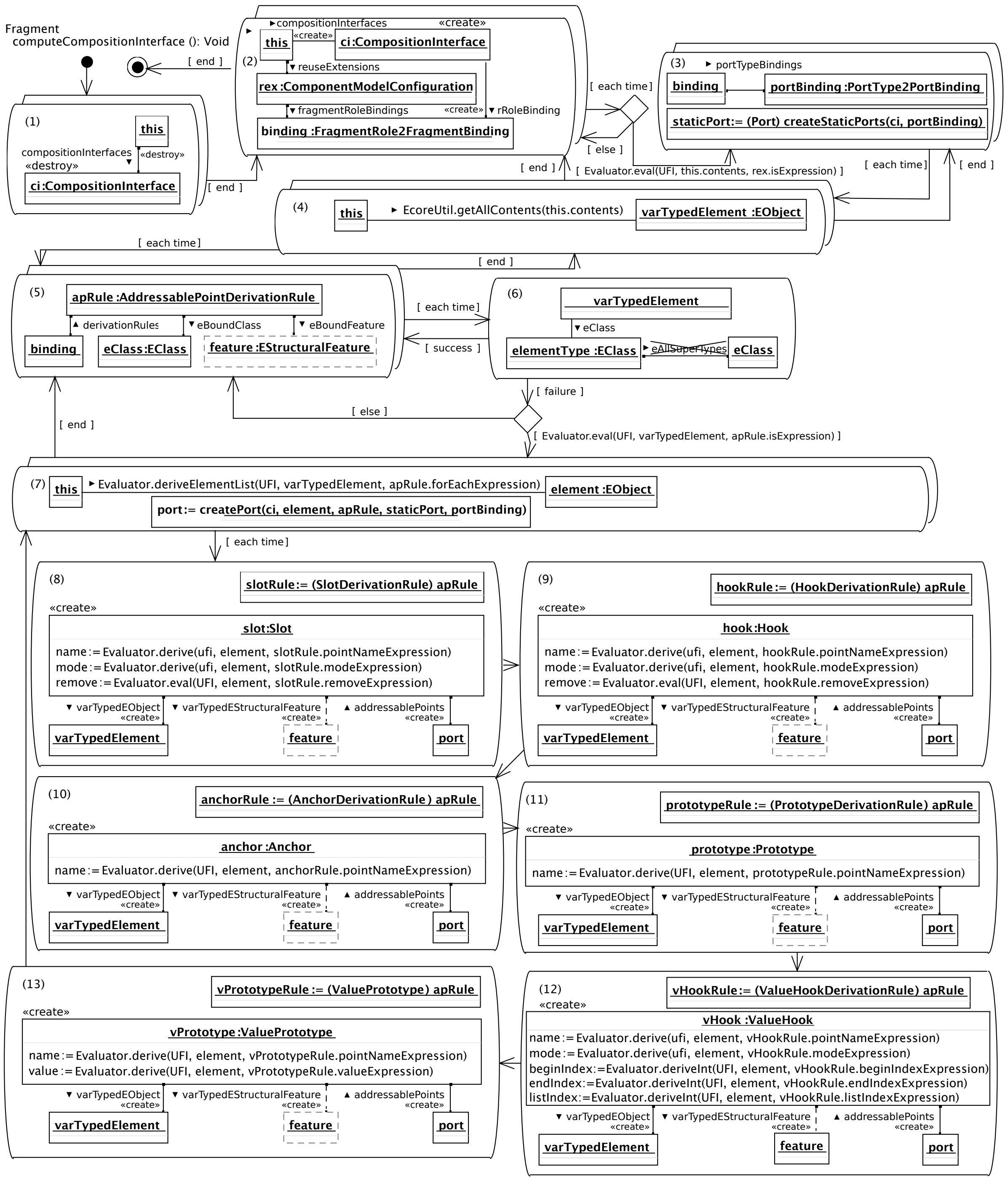

Fragment
computeCompositionInterface (): Void
(1)
this
compositionInterfaces
«destroy»
«destroy»
ci:CompositionInterface
[end]
(2)
compositionInterfaces
«create»
this
«create»
ci:CompositionInterface
reuseExtensions
rex :ComponentModelConfiguration
fragmentRoleBindings
«create»
rRoleBinding
binding :FragmentRole2FragmentBinding
[end]
[each time]
[else]
[end]
[Evaluator.eval(UFI, this.contents, rex.isExpression)]
(3)
portTypeBindings
binding
portBinding :PortType2PortBinding
staticPort:= (Port) createStaticPorts(ci, portBinding)
[each time]
[end]
(4)
this
EcoreUtil.getAllContents(this.contents)
varTypedElement :EObject
[each time]
[end]
(5)
apRule :AddressablePointDerivationRule
derivationRules
eBoundClass
eBoundFeature
binding
eClass:EClass
feature :EStructuralFeature
[each time]
[success]
(6)
varTypedElement
eClass
elementType :EClass
~~eAllSuperTypes~~
eClass
[failure]
[else]
[Evaluator.eval(UFI, varTypedElement, apRule.isExpression)]
[end]
(7)
this
Evaluator.deriveElementList(UFI, varTypedElement, apRule.forEachExpression)
element :EObject
port:= createPort(ci, element, apRule, staticPort, portBinding)
[each time]
(8)
slotRule:= (SlotDerivationRule) apRule
«create»
slot:Slot
name := Evaluator.derive(ufi, element, slotRule.pointNameExpression)
mode := Evaluator.derive(ufi, element, slotRule.modeExpression)
remove := Evaluator.eval(UFI, element, slotRule.removeExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port
(9)
hookRule := (HookDerivationRule) apRule
«create»
hook:Hook
name := Evaluator.derive(ufi, element, hookRule.pointNameExpression)
mode := Evaluator.derive(ufi, element, hookRule.modeExpression)
remove := Evaluator.eval(UFI, element, hookRule.removeExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port
(10)
anchorRule := (AnchorDerivationRule) apRule
«create»
anchor :Anchor
name := Evaluator.derive(UFI, element, anchorRule.pointNameExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port
(11)
prototypeRule := (PrototypeDerivationRule) apRule
«create»
prototype :Prototype
name := Evaluator.derive(UFI, element, prototypeRule.pointNameExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port
(13)
vPrototypeRule := (ValuePrototype) apRule
«create»
vPrototype :ValuePrototype
name := Evaluator.derive(UFI, element, vPrototypeRule.pointNameExpression)
value := Evaluator.derive(UFI, element, vPrototypeRule.valueExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port
(12)
vHookRule := (ValueHookDerivationRule) apRule
«create»
vHook :ValueHook
name := Evaluator.derive(UFI, element, vHookRule.pointNameExpression)
mode := Evaluator.derive(ufi, element, vHookRule.modeExpression)
beginIndex:=Evaluator.deriveInt(UFI, element, vHookRule.beginIndexExpression)
endIndex:=Evaluator.deriveInt(UFI, element, vHookRule.endIndexExpression)
listIndex:=Evaluator.deriveInt(UFI, element, vHookRule.listIndexExpression)
varTypedEObject «create»
varTypedEStructuralFeature «create»
addressablePoints «create»
varTypedElement
feature
port

Figure A.3: `Fragment.computeCompositionInterface()` operation.

3. In this pattern, static port types are processed. For each static port type, a port is created using the `createStaticPort()` operations defined below.
4. Now that we have verified that the current REX_{CM} specification applies to this fragment and created an empty interface for it, we have to determine the variability type for each of the model elements. To obtain an iterator over all elements, we make use of the `EcoreUtil.getAllContents()` operation provided by the EMF that iterates over the containment hierarchy (i.e., the spanning forest) of this fragment's graph (`contents` reference).
5. Each derivation rule has to be checked for each model element. Thus, we also iterate of all rules defined in the REX_{CM} specification. For each model element and rule combination, we execute the following patterns.
6. Initially, we check if the current element is an instance of the `EClass`. For that, we apply negation to find wrong matches. If the type is correct, we evaluate the `isExpression` using the Evaluator before continuing to Pattern 7.
7. We now iterate over all elements that are reached by interpreting the `forEachExpression`. For this, the Evaluator offers the `deriveElementList()` operation. This operation interprets the given query (here `forEachExpression`) in the context of the given element (here `varTypedElement`) and returns the result as a list. It returns a list containing the original element if the passed query is empty, which is the case if no `forEachExpression` expression is specified. Based on the rule applied to the current element, a port is selected or newly created using the `createPort()` (Figure A.5) operation that is described below.
8. to 13. The actual variability type assignments are performed by creating appropriate addressable points. Which type of addressable point is created is determined by the type of the addressable point derivation rule (e.g., a slot is created for a slot derivation rule). In each case the Evaluator's `derive()` operation is called to derive the name of the addressable point using the point name expression. In the case of a value hook (Pattern 12), the `beginIndex` and the `endIndex` are derived in a similar fashion and in the case of a value prototype (Pattern 13) the `value` is derived. The variability type is attached to the element by setting the `varTypedEObject` reference of the created addressable point to the element. If a feature was specified, this is set in the `varTypedEStructuralFeature` reference. Finally, the addressable point is added to the port.

The operation `createStaticPort()` is shown in Figure A.4 and explained in the following.

1. We find the `StaticPortType` bound by the port type binding. Afterwards, we either create a heterogeneous port or a homogeneous port depending on the concrete port type binding
2. In the former case we do:
   2.1 If a heterogeneous port that is linked with the binding exists, we return that port.
   2.2 If no heterogeneous port for the given binding exists, we create one, assign the name of the bound port type as port name to it, and link it to the binding.
3. In the latter case (homogeneous port) we do:
   3.1 If a homogeneous port that is linked with the binding exists, we return that port.

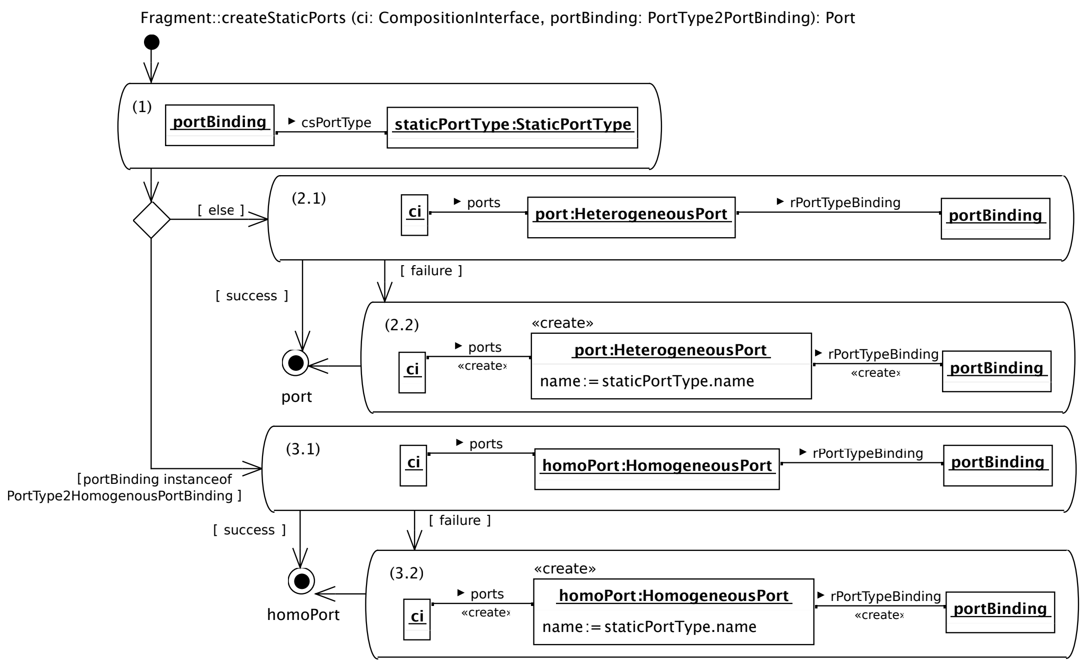


Figure A.4: `Fragment.createStaticPort()` operation.

3.2 If no homogeneous port for the given binding exists, we create one, assign the name of the bound port type as port name to it, and link it to the binding.

The operation `createPort()` that was utilised above is shown in Figure A.5 and described in the following. According to the port and homogeneous port name expressions of a given derivation rule, the operation selects an existing port from a given composition interface or creates a new one and adds it to the interface.

1. If the `staticPort` parameter is bound to a `HomogeneousPort`, we set `homoPort` to the parameter value and directly proceed with Pattern 3.3 to handle heterogeneous ports. If the `staticPort` parameter is bound to a `HeterogeneousPort`, no further ports need to be created dynamically and that port is returned.
2. If no static port exists, we have to decide if a homogeneous port is needed in addition to a heterogeneous port. If a homogeneous port name expression is defined in the derivation rule, we can create a homogeneous port that wraps a set of heterogeneous ports (cf. Section 5.2.4). If not, we create only a heterogeneous port (cf. Section 5.2.3). In the latter case we do:
   2.1 We derive the port name from the derivation rule's port name expression using the Evaluator. If a heterogeneous port with that name already exists on the given composition interface, this port is returned.
   2.2 If no heterogeneous port with this name exists, it is created, the name is assigned, it is added to the composition interface, linked to the binding, and returned.
3. In the former case (homogeneous port is required) we do:

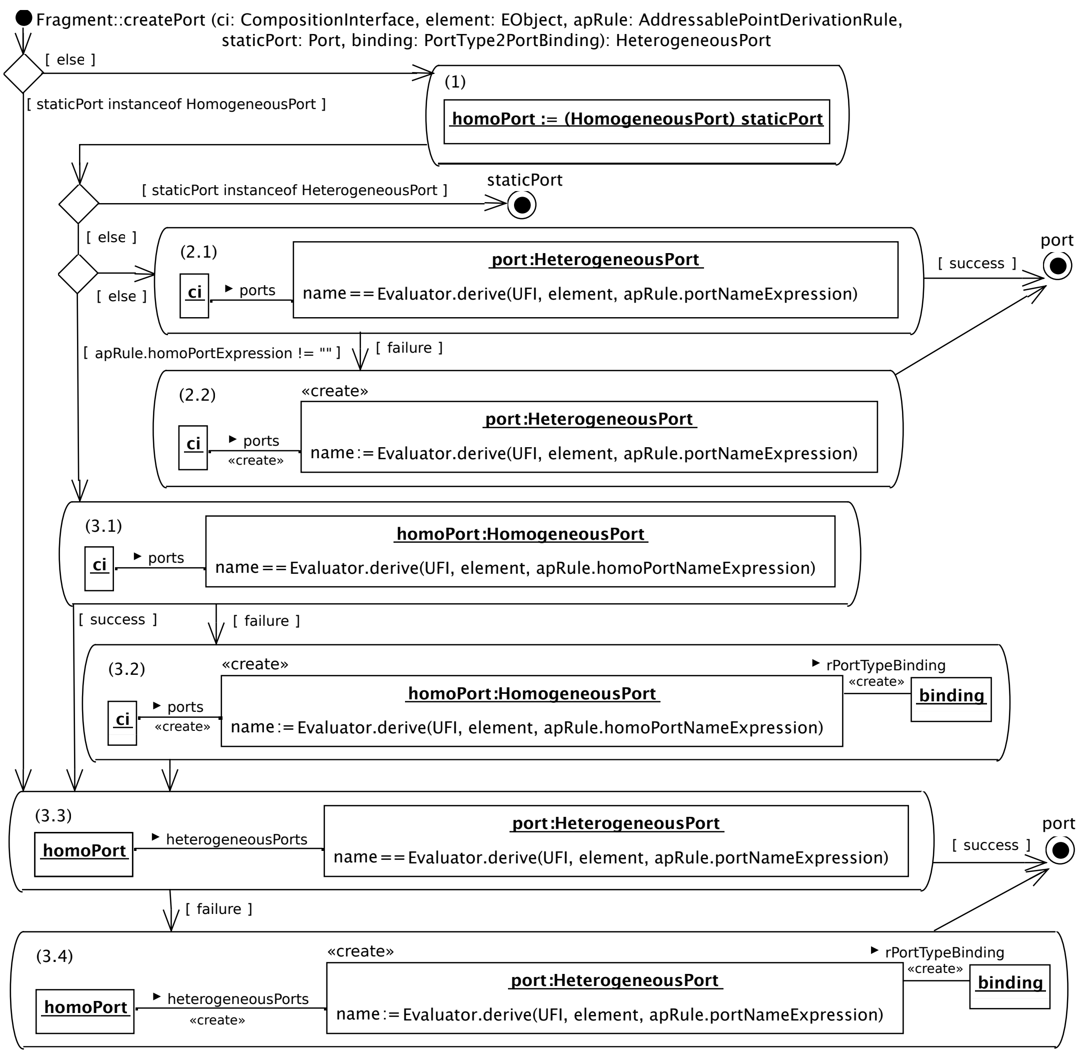

Fragment::createPort (ci: CompositionInterface, element: EObject, apRule: AddressablePointDerivationRule, staticPort: Port, binding: PortType2PortBinding): HeterogeneousPort
[else]
[staticPort instanceof HomogeneousPort]
(1)
homoPort := (HomogeneousPort) staticPort
[staticPort instanceof HeterogeneousPort]
staticPort
[else]
[else]
(2.1)
ci
ports
port:HeterogeneousPort
name==Evaluator.derive(UFI, element, apRule.portNameExpression)
[success]
port
[apRule.homoPortExpression != ""]
[failure]
(2.2)
«create»
port:HeterogeneousPort
ci
ports
«create»
name:=Evaluator.derive(UFI, element, apRule.portNameExpression)
(3.1)
homoPort:HomogeneousPort
ci
ports
name==Evaluator.derive(UFI, element, apRule.homoPortNameExpression)
[success]
[failure]
(3.2)
«create»
rPortTypeBinding
«create»
homoPort:HomogeneousPort
binding
ci
ports
«create»
name:=Evaluator.derive(UFI, element, apRule.homoPortNameExpression)
(3.3)
port:HeterogeneousPort
homoPort
heterogeneousPorts
name==Evaluator.derive(UFI, element, apRule.portNameExpression)
[success]
port
[failure]
(3.4)
«create»
rPortTypeBinding
«create»
port:HeterogeneousPort
binding
homoPort
heterogeneousPorts
«create»
name:=Evaluator.derive(UFI, element, apRule.portNameExpression)

Figure A.5: `Fragment.createPort()` operation.

3.1 We derive the name of the homogeneous port from the derivation rule's homogeneous port name expression using the Evaluator. If a homogeneous port with that name already exists we continue with 3.3.
3.2 If no homogeneous port with this name exists, it is created, the name is assigned and it is added to the composition interface.
3.3 We derive the name of the heterogeneous port from the derivation rule's port name expression. If a heterogeneous port with that name exists in the homogeneous port, it is returned.
3.4 If no heterogeneous port with this name exists in the homogeneous port, it is created, the name is assigned, it is added to the homogeneous port, linked to the binding, and returned.

A.3 Universal Composition Language Semantics

This section defines the semantics for UCL described in Section 6.3.

A.3.1 Composition Link Matching

Two operations for matching addressable points (cf. Figure 5.21a) are defined. `VariationPoint.typeMatch()` checks if the domain type of a given reference point, which is defined in the corresponding metamodel, matches the type of the variation point. `AddressablePoint.match()` checks if two addressable points are compatible from the variability typing viewpoint. In the following, we first explain the details of `typeMatch()` shown in Figure A.6. Afterwards, we examine `match()` in more detail.

1. First we navigate to the `Fragment` containing this `VariationPoint` because it is needed later in Patterns 5.1 and 6.1.
2. If a `HomogeneousPort` is involved, the previous pattern fails to find the `Fragment`, so this pattern performs the task.
3. Patterns 3–6 perform a case analysis on whether a node (`varTypedEObject`) or a feature (`varTypedEStructuralFeature`) is variability typed. This pattern covers the case, where two `EReferences` are variability typed.
   3.1 We apply negation to find negative matches. If the type of both `EReferences` is not equal, then the type of this `VariationPoint` has to be a super type of the type of the `ReferencePoint` (`eAllSuperClasses` edges). If this is not the case, the error pattern matches and `false` is returned.
4. Next, the case where the reference point types a node is covered.
   4.1 We apply negation to find wrong matches. If the type of the `EReferences` that is typed by the variation point is not equal the type of the `EObject` that is typed by the reference point, then the type of this `VariationPoint` has to be a super type of the type of the `ReferencePoint` (`eAllSuperClasses` edges). If this is not the case, the error pattern matches and `false` is returned.
5. Next, the case where the variation point types a node is covered.
   5.1 We apply negation to find wrong matches. The following condition has to be true for each edge that points at the `EObject` typed by the variation point: If the type of one of these edges (`Settings`) is neither equal nor a super type of the type of the `EReferences` typed by the reference point, the error pattern matches and `false` is returned. This is necessary, because all edges that point at the variation node are altered during composition. Thus, all types have to match.
6. Finally, the case where both the variation point and the reference point type a node (and not an edge list) is covered.
   6.1 We apply negation to find wrong matches. The following has to be true for each edge that points at the `EObject` typed by the variation point: If the type of one of these edges (`Settings`) is neither equal nor a super type of the type of the `EObject` typed by the reference point, the error pattern matches and `false` is returned.

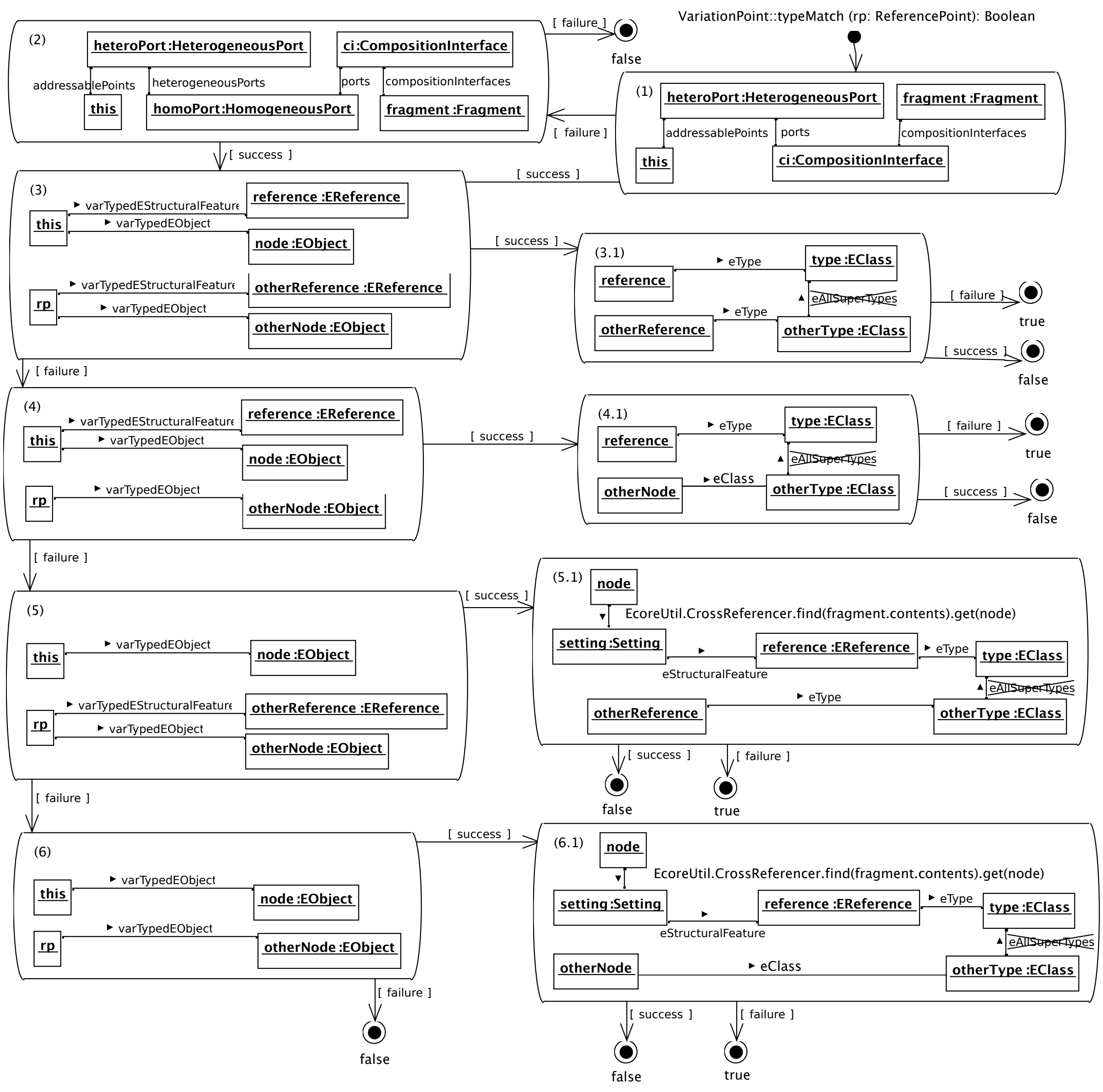


Figure A.6: `VariationPoint.typeMatch()` operation.

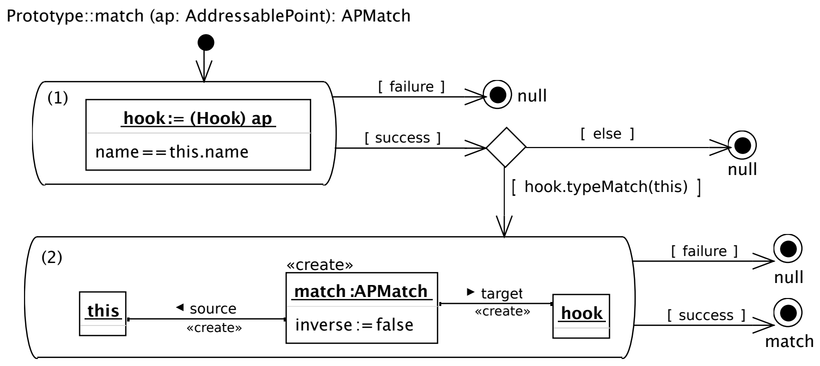


Figure A.7: `Prototype.match()` operation (as example for `AddressablePoint.match()`).

In addition to the type matching above, also the variability types between addressable points need to be matched, which is performed in `AddressablePoint.match()`. It is specified similarly for each concrete subclass of `AddressablePoint` and is shown and explained for `Prototype` in Figure A.7.

If `AddressablePoint.match()` succeeds, it returns an `APMatch` used to cache and transport the match. The `inverse` flag of `APMatch` indicates the direction in which `source` and `target` are related. The default direction implies that `source` can replace `target` (i.e., `source` is a reference and `target` is a variation point). If `inverse` is set, the opposite is the case.

1. First, it is checked if the `AddressablePoint` given as parameter is a `Hook`, since this is the variability type compatible with `Prototype`. It is also checked if names of `Hook` and `Prototype` are equal. This check will also succeed if both names are not set.
2. Second, the domain type compatibility check is performed by calling the above described `typeMatch()` operation (cf. Figure A.6) on the `Hook`. If this succeeds, an `APMatch` is created with this `Prototype` as `source` and the `Hook` as target. Since this is the default matching direction, `inverse` is set to *false*. Note that the `typeMatch` call is missing in the definitions of `match()` for `ValueHook` and `ValuePrototype`, since attributes do not have a domain type and therefore always match.

To match a complete composition link, the `CompositionLink.match()` operation shown in Figure A.9 is performed. It relies on the type and variability type matching of variation and reference points.

`match()` determines if the link is valid. To record matches in the case of a valid link, we introduce the `APMatchGroup` concept between `CompositionLink` and `APMatch` (cf. Figure 5.21) shown in Figure A.8. `match()` sets the `matches` reference of `CompositionLink` to an `APMatchGroup` in which all `APMatches`, which are produced by performing `AddressablePoint.match()` operations, are collected. Thus, since an `APMatch` describes the match between two addressable points, `match()` determines if and how the addressable points behind the ports of the two connected port instances can be composed.

The `match()` operation of Figure A.9 is detailed in the following:

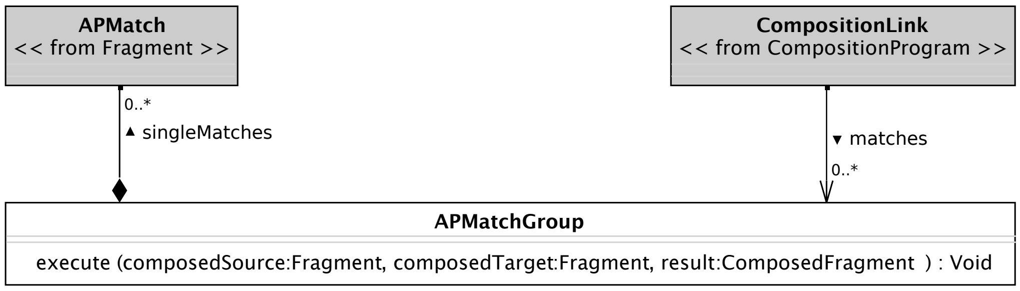

APMatch
<< from Fragment >>
CompositionLink
<< from CompositionProgram >>
0..*
▲ singleMatches
▼ matches
0..*
APMatchGroup
execute (composedSource:Fragment, composedTarget:Fragment, result:ComposedFragment) : Void

Figure A.8: `APMatchGroup` and its relations to `APMatch` (cf. Figure 5.21) and `CompositionLink` (cf. Figure 6.13).

1. The attributes `valid`, `contributes`, and `empty` are reset. They are computed in the following patterns.
2. All old `APMatchGroups` are destroyed.
3. The following Patterns 5–13 are performed for each combination of heterogeneous port behind the source...
4. ...and target port instances.
5. An `APMatchGroup` (cf. Figure A.8) for the pair of source and target heterogeneous ports is created.
6. Each combination of addressable points is individually matched via the `match()` operation on the source addressable point and all successful matches are recorded in the current match group.
7. If no match was found, the link is set invalid because there is nothing to compose.
8. If a variation point is target of several matches that do not have reference points with similar names as source, the matching is ambiguous and the link is set invalid.
9. If a reference point is source of several matches, the matching is ambiguous and the link is set invalid.
10. It is checked if at least one reference point was matched either on the source...
11. ...or on the target port.
12. If no reference point was matched, the link is set to invalid because there is nothing to compose.
13. If the link is valid and connects a receiving with a contributing port, the `contributes` flag is set to `true`.
14. If no match group was created (i.e., there were no ports behind the port instances to connect), the link is set to empty.
15. If the link is not valid, match groups that were created in the attempt to find a valid match are destroyed.

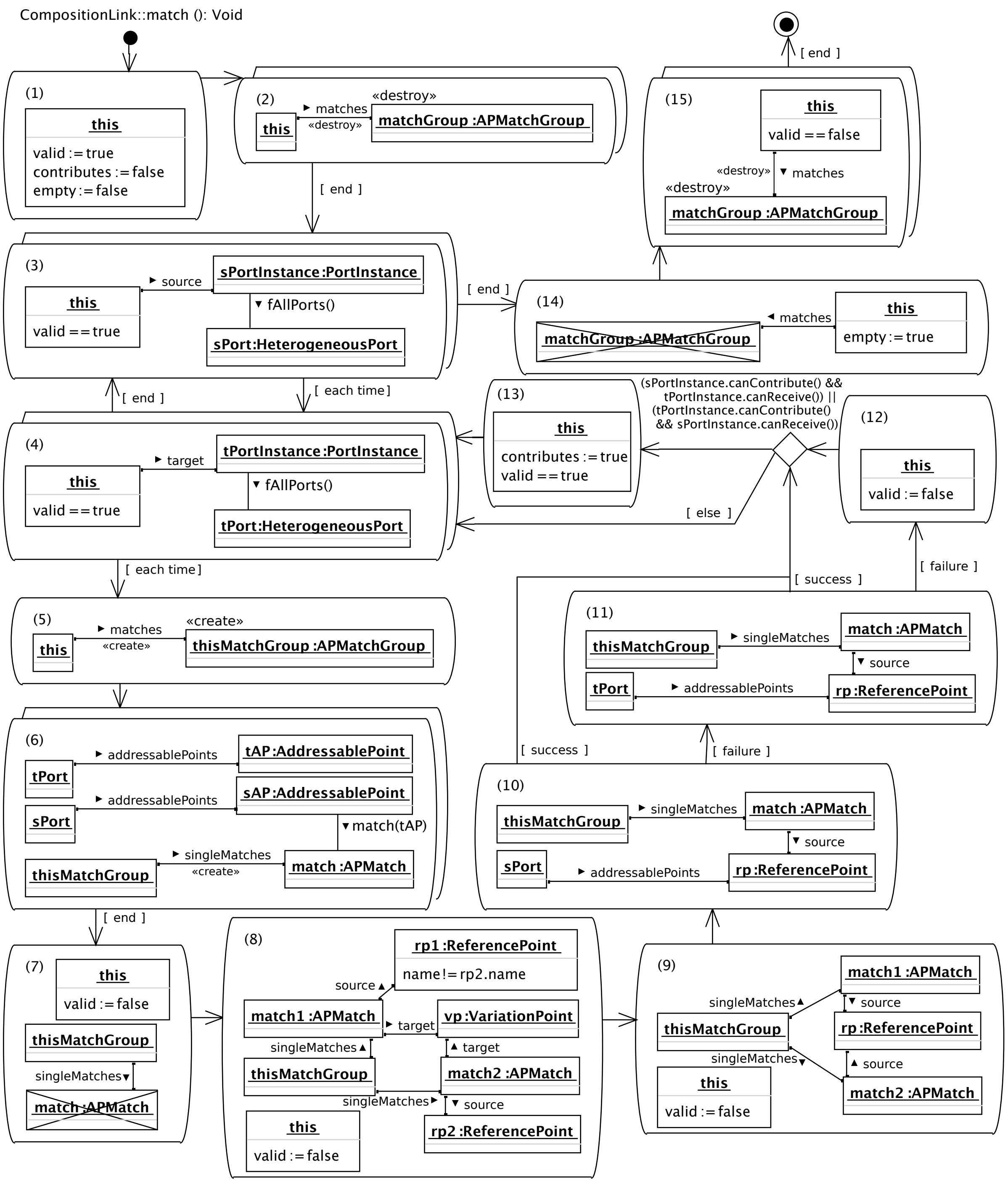


Figure A.9: `CompositionLink.match()` operation.

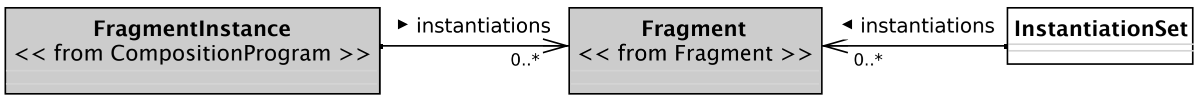


Figure A.10: `InstantiationSet` and `instantiations` relations between `InstantiationSet`, `Fragment` (cf. Figure 5.21), and `FragmentInstance` (cf. Figure 6.13).

A.3.2 Composition Step Identification

The composition execution is controlled by `FragmentInstance.fCompose()` that determines the composition steps and triggers copying of fragments. `fCompose()` is shown in Figure A.11.

1. A copy of the current fragment (`result`) is created using the `Fragment.copy()` operation (cf. Section A.3.3). This copy is—for the time of this operation—marked as concrete instantiation of the fragment by adding it to an `InstantiationSet` (temporarily created for the execution time of this operation) and to the `instantiations` of the current fragment instance. `InstantiationSets`, depicted in Figure A.10, are used to hold a temporal state for the duration of one composition execution. With an instantiation we here refer to a composed fragment that was originally copied from the fragment it is an instantiation of.
2. For all port instances, all settings are executed in the context of the result (see description of `Setting.execute()` below).
3. For each fragment instance of the composition program (except the current one)—which is a potential *contributor* of graph parts—we perform the following activities.
   3.1 In the case that the contributor indeed contributes to the current fragment instance through at least one contributing composition link, one link is selected as primary link and the next activity is triggered.
   3.2 For each match group of the primary link, we create an instantiation of the contributor by a recursive `fCompose()` call. The result is added to the instantiation set and marked as instantiation of the contributor.
   3.3 For each contributing composition link between the current fragment instance and the contributor, a match group is executed (cf. description of `MatchGroup.execute()` below). Note that in cases of multiple match groups, which might exist if homogeneous ports are involved, the primary link controls which match group is currently evaluated. It is expected that all contributing links between two fragments have the same number of match groups, due to the nature of homogeneous ports.
4. The helper operation `completeInstantiation()` is called that can (through recursive calls to itself) perform additional instantiations of fragment instances that potentially contribute additional elements (cf. description below).
5. The `execute()` operation of all match groups (details below) that belong to non-contributing links is performed. The `execute()` for the match groups of contributing links was executed in Pattern 3.3.
6. All the temporarily created `instantiations` edges are destroyed.

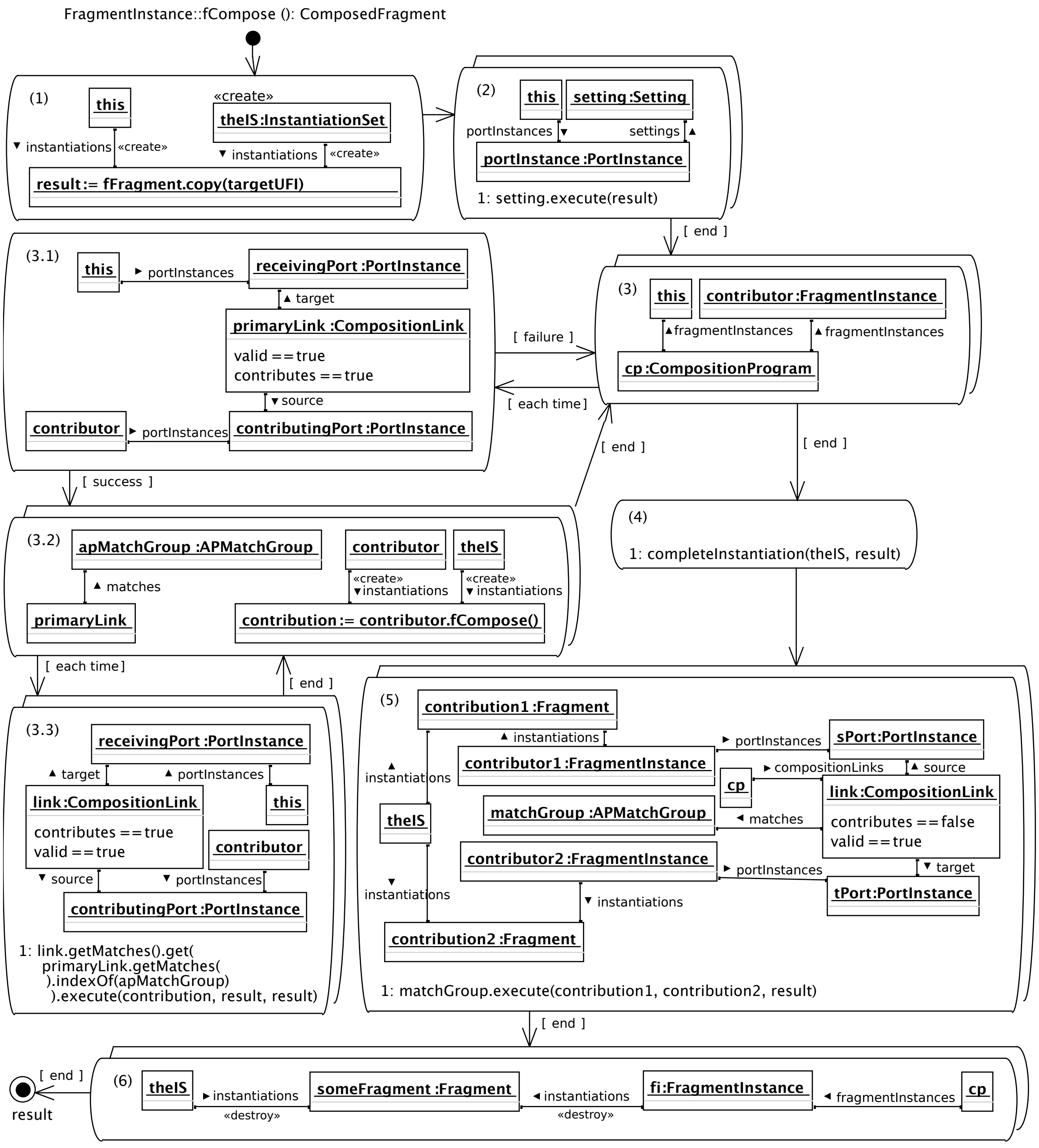
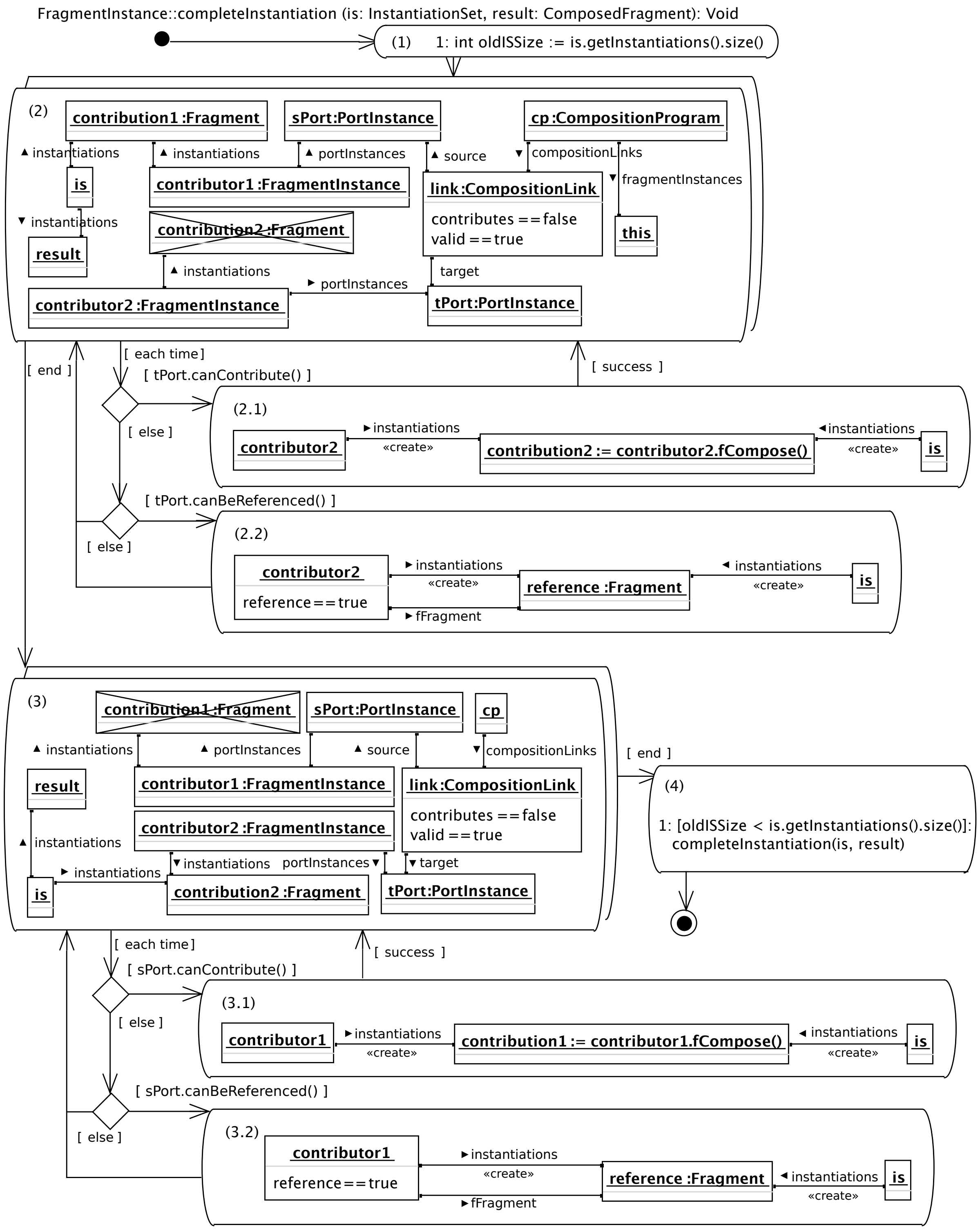


Figure A.11: `FragmentInstance.fCompose()` operation.



FragmentInstance::completeInstantiation (is: InstantiationSet, result: ComposedFragment): Void
(1) 1: int oldISSize := is.getInstantiations().size()
(2)
contribution1 :Fragment
sPort:PortInstance
cp :CompositionProgram
instantiations
portInstances
source
compositionLinks
fragmentInstances
is
contributor1 :FragmentInstance
link :CompositionLink
contributes == false
valid == true
this
~~contribution2 :Fragment~~
result
target
contributor2 :FragmentInstance
tPort:PortInstance
[each time]
[end]
[tPort.canContribute()]
[success]
(2.1)
[else]
contributor2
«create»
contribution2 := contributor2.fCompose()
[tPort.canBeReferenced()]
(2.2)
reference == true
reference :Fragment
fFragment
(3)
~~contribution1 :Fragment~~
cp
contributor2 :FragmentInstance
contribution2 :Fragment
(4)
1: [oldISSize < is.getInstantiations().size()]: completeInstantiation(is, result)
[sPort.canContribute()]
(3.1)
contributor1
contribution1 := contributor1.fCompose()
[sPort.canBeReferenced()]
(3.2)

Figure A.12: `FragmentInstance.completeInstantiation()` operation.

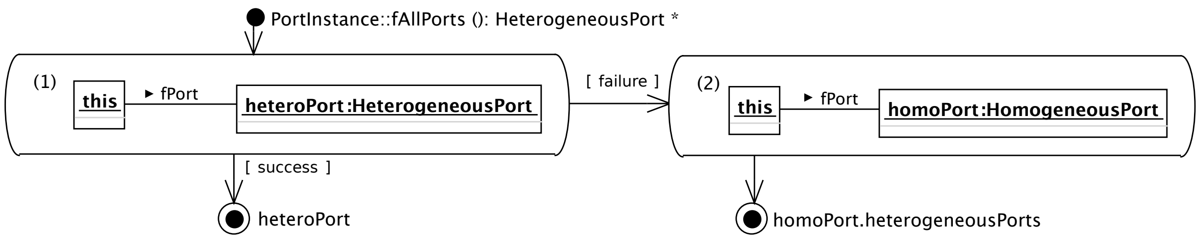

PortInstance::fAllPorts (): HeterogeneousPort *
(1)
this
fPort
heteroPort:HeterogeneousPort
[failure]
(2)
this
fPort
homoPort:HomogeneousPort
[success]
heteroPort
homoPort.heterogeneousPorts

Figure A.13: `FragmentInstance.fAllPorts()` operation.

The following describes the helper operation `completeInstantiation()` shown in Figure A.12 in detail.

1. First, we remember the number of instantiations that currently exists.
2. We examine all valid links of the current composition program that have the `contributes` flag set to `false`. For each fragment we encounter that is target of a link but does not have an instantiation yet, we do the following.
   2.1 If the port can contribute elements despite the fact that it is not marked as contributing, an instantiation is created.
   2.2 If the port can be referenced and the fragment instance is set to `reference`, the fragment behind the instance is taken as the instantiation without copying it. This way, the composition accepts the original fragment as participant in the composition and allows the creation of new cross-reference to this fragment. Note that, since `Port.canBeReferenced()` returns *true* only when the port contains anchors, only cross-references towards the original fragment are established, but the fragment itself is never modified, which makes its direct participation in the invasive composition acceptable.
3. This pattern (with 3.1 and 3.2) is similar to Pattern 2 (with 2.1 and 2.2) with the only difference that the computations are performed for the source of each composition link (and not for the target).
4. If the execution of this operation did produce new instances, the operation calls itself again, since the new instances potentially contains more composition links of interest to other yet uninstantiated fragment instances.

There are some small operations on `PortInstance` that were used above. The operation `fAllPorts()` (Figure A.13) is provided to conveniently access all `HeterogeneousPorts` directly in cases these are grouped by a `HomogeneousePort`. The operations 1) `canContribute()`, 2) `canReceive()`, and 3) `canBeReferenced()` (Figures A.14–A.16) compute if 1) the port instance can contribute nodes because its port contains prototypes 2) the port instance may receive nodes because its port contains hooks 3) the port instance can be referenced because its port contains anchors.

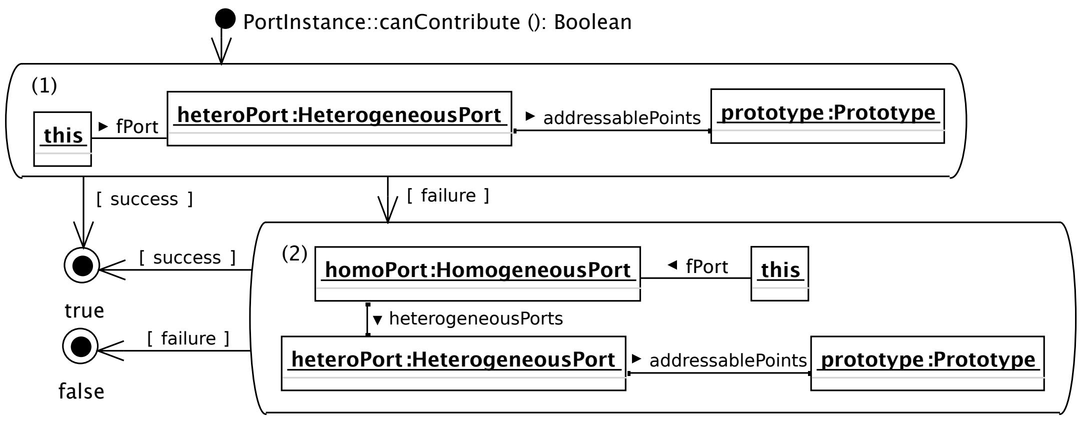


Figure A.14: `PortInstance.canContribute()` operation.

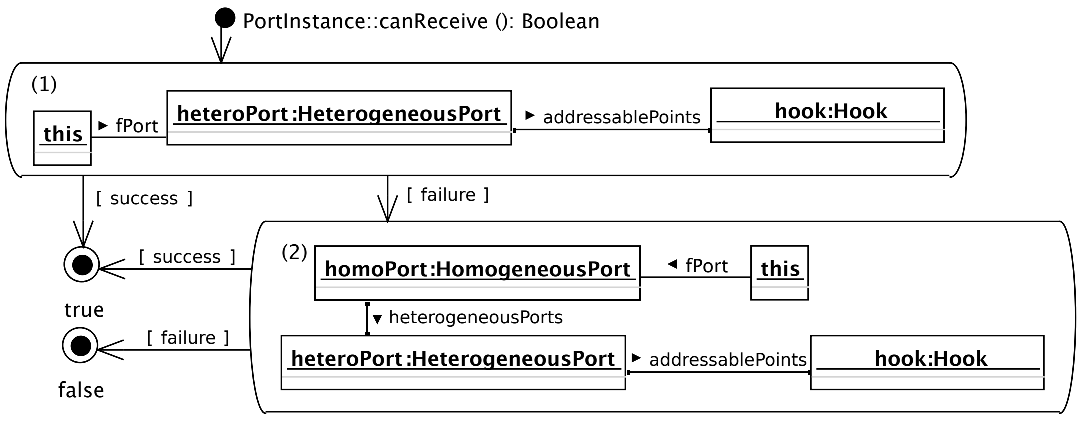


Figure A.15: `PortInstance.canReceive()` operation.

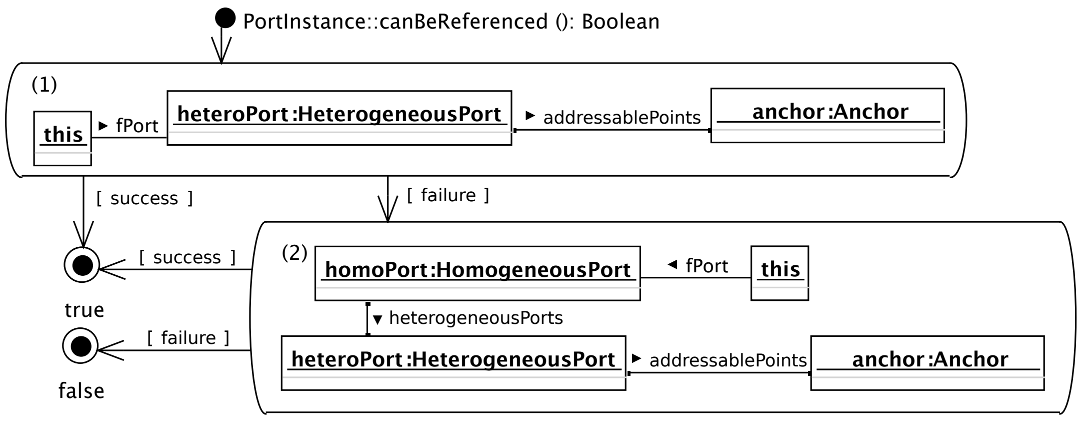


Figure A.16: `PortInstance.canBeReferenced()` operation.

A.3.3 Fragment Copying

Copying graph fragments boils down to producing an exact copy of the underlying graph. A tool for this task is already delivered with EMF in the `EcoreUtil.Copier` utility [SBPM09, Section 16.4] that we reuse here. The `EcoreUtil.Copier` provides the operation `copyAll()` which takes a list of root `EObjects` (i.e., root nodes) of an arbitrary graph fragment. It then copies these nodes and all their contained nodes along the containment spanning trees (i.e., all nodes that can be reached via an edge $e_G \in T_G$) of the graph. For each node, it stores a mapping between the original and the copy.

In the node copying step via `copyAll()` the non-containment edges ($e_G \notin T_G$) are not considered. These are copied through a call to `copyReferences()` also available on `EcoreUtil.Copier`. This mechanism allows the successive copying of cross-connected graphs via `copyAll()` before calling `copyReferences()`.

Furthermore, we introduce a cache for graph fragment copies in a tool called `SyncEcoreUtil`. This cache remembers the relation between a fragment copy and a fragment original as well as an original and its copies. Thus it can be asked to provide an original for a copy or all copies for an original. Retrieving this information is important for calling the basic composition operators on the correct copies but can also be exploited by other tools (e.g., for diagram composition or round-trip support discussed in Section 12.2).

Figure A.17 shows the story diagram that defines the `Fragment.copy()` operation. In the story diagram we use collaboration statements to integrate the `EcoreUtil.Copier` and the `SyncEcoreUtil` explained above.

1. An `EcoreUtil.Copier` is instantiated and used to copy all nodes and edges of the graph fragment that is reached via `Fragment.contents`.
2. A new `ComposedFragment` is created with the `copyUFI` as UFI that was passed as parameter to this operation. Thus, always when a fragment is created via the `copy()` operation it will be a `ComposedFragment` (as opposed to a `PhysicalFragment`). This allows us to distinguish fragments that were created by developers (`PhysicalFragment`) from fragments that were created by copying and composition (`ComposedFragment`). The latter can always be recreated from other fragments and have to be recreated if fragments they depend on change. The created copy is registered with the `SyncEcoreUtil`.
3. For each root node, we obtain the copy from the `EcoreUtil.Copier` (via `get()` operation) and add it to the `contents` of the copy.

A.3.4 Basic Composition Operator Calls

The actual composition of graph fragments is performed in the two execute operations `MatchGroup.execute()` (Figure A.18) and `Setting.execute()` (Figure A.19), which are called at appropriated places in `fCompose()` as described above.

`MatchGroup.execute()` iterates over all its matches, finds the appropriate copies of nodes that are to be composed and then calls the variation point `compose()` operation discussed below which in turn calls the basic composition operators (cf. Section 6.1). In detail, this is shown in Figure A.18 and explained in the following.

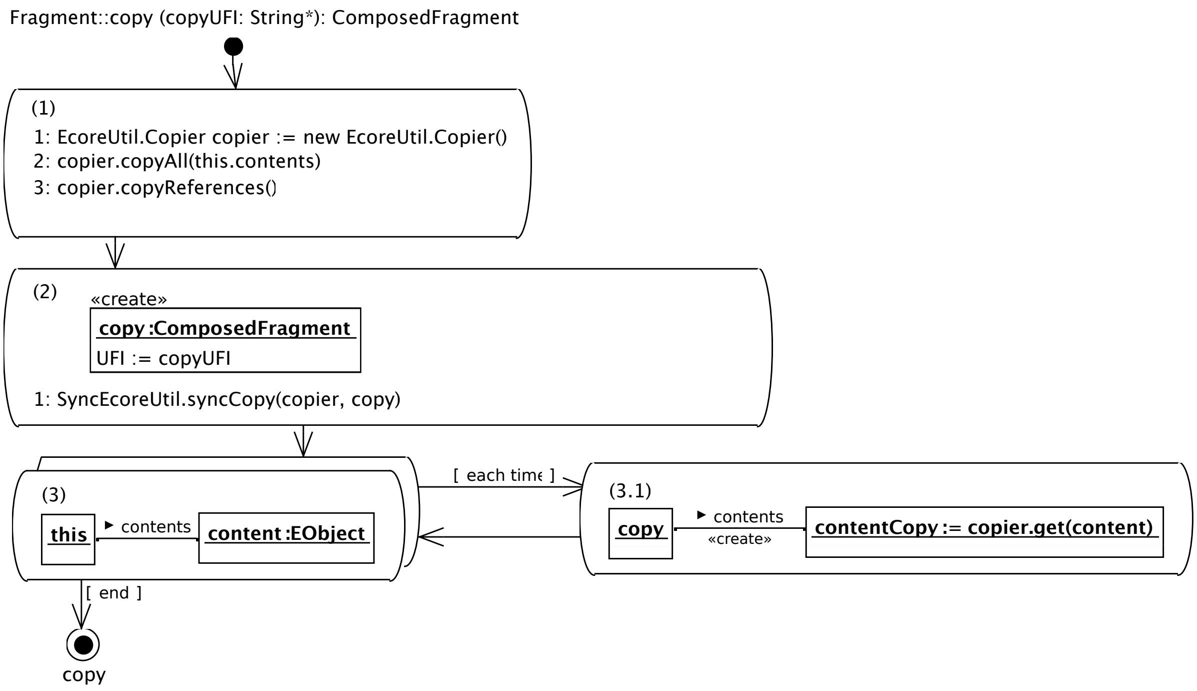

Fragment::copy (copyUFI: String*): ComposedFragment
(1)
1: EcoreUtil.Copier copier := new EcoreUtil.Copier()
2: copier.copyAll(this.contents)
3: copier.copyReferences()
(2)
«create»
copy :ComposedFragment
UFI := copyUFI
1: SyncEcoreUtil.syncCopy(copier, copy)
(3)
this
contents
content :EObject
[each time]
(3.1)
copy
contents
«create»
contentCopy := copier.get(content)
[end]
copy

Figure A.17: `Fragment.copy()` operation.

1. The parameters of the operation are `composedSource` (the composed fragment that is source of this match), `composedTarget` (the composed fragment that is target of this match) and `result` (the composed fragment that is the target of the whole composition step). These fragments contain the copies that are of interest for composition execution. Since nodes move between the copies during one composition step, the `result` is needed here as well because a node might already have moved there (from `composedSource` or `composedTarget`) by a previous execute call for matches between prototypes and hooks. We combine the `contents` of all three fragment copies to form the context into which we have to look for node copies.
2. We iterate over all matches with default match direction (`inverse` = *false*). For each match, we obtain a copy of the node typed by the variation point that is target of the match. This is done using the `SyncEcoreUtil` in which the relationship between original fragments and composed fragments (i.e., fragment copies) is recorded (cf. Section A.3.3). In addition to the original node (here `variationNode`) and the context, the `SyncEcoreUtil` requires a composed fragment as parameter, which is the one for which the copy was originally created (here `composedTarget`). This way, the correct copy can be selected, if multiple copies for one node exist.
   2.1 We also obtain the copy for the current reference node. Then we call `compose()` (described below) on the current variation point passing the obtained copies as parameters. This pattern fails in cases where the source fragment is set to `reference`. In this case, the original reference node is treated as copy…

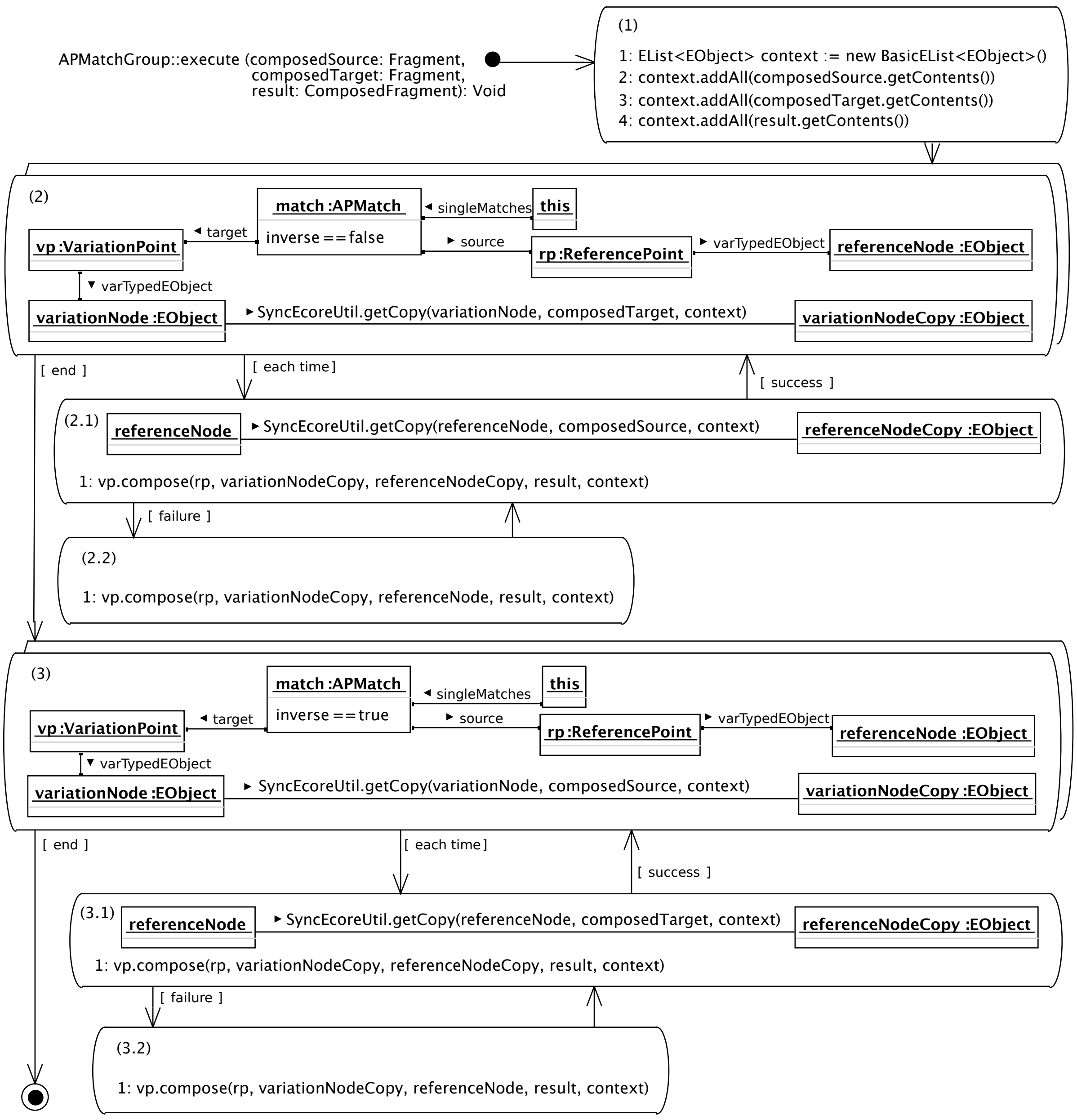

APMatchGroup::execute (composedSource: Fragment,
composedTarget: Fragment,
result: ComposedFragment): Void
(1)
1: EList<EObject> context := new BasicEList<EObject>()
2: context.addAll(composedSource.getContents())
3: context.addAll(composedTarget.getContents())
4: context.addAll(result.getContents())
(2)
match :APMatch
inverse == false
singleMatches
this
vp :VariationPoint
target
source
rp :ReferencePoint
varTypedEObject
referenceNode :EObject
varTypedEObject
variationNode :EObject
SyncEcoreUtil.getCopy(variationNode, composedTarget, context)
variationNodeCopy :EObject
[end]
[each time]
[success]
(2.1)
referenceNode
SyncEcoreUtil.getCopy(referenceNode, composedSource, context)
referenceNodeCopy :EObject
1: vp.compose(rp, variationNodeCopy, referenceNodeCopy, result, context)
[failure]
(2.2)
1: vp.compose(rp, variationNodeCopy, referenceNode, result, context)
(3)
match :APMatch
inverse == true
singleMatches
this
vp :VariationPoint
target
source
rp :ReferencePoint
varTypedEObject
referenceNode :EObject
varTypedEObject
variationNode :EObject
SyncEcoreUtil.getCopy(variationNode, composedSource, context)
variationNodeCopy :EObject
[end]
[each time]
[success]
(3.1)
referenceNode
SyncEcoreUtil.getCopy(referenceNode, composedTarget, context)
referenceNodeCopy :EObject
1: vp.compose(rp, variationNodeCopy, referenceNodeCopy, result, context)
[failure]
(3.2)
1: vp.compose(rp, variationNodeCopy, referenceNode, result, context)

Figure A.18: `APMatchGroup.execute()` operation.

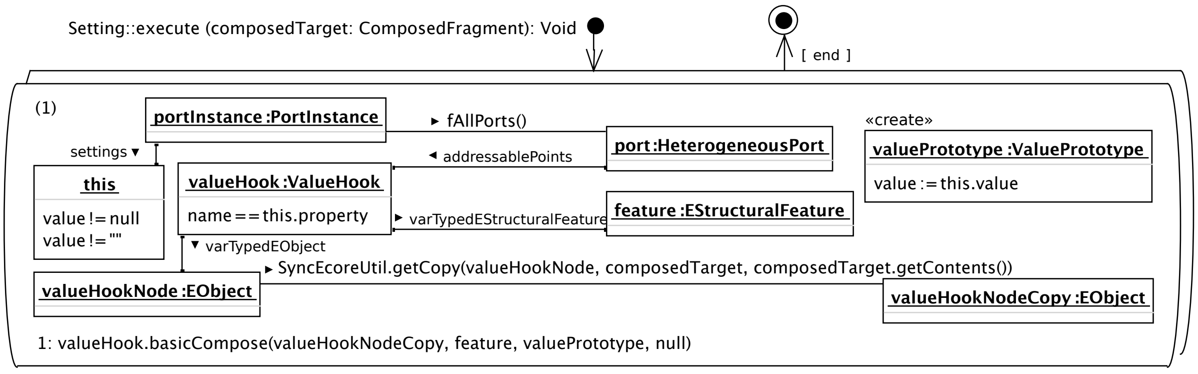

Setting::execute (composedTarget: ComposedFragment): Void
[end]
(1)
portInstance :PortInstance
fAllPorts()
settings
addressablePoints
port :HeterogeneousPort
«create»
valuePrototype :ValuePrototype
value := this.value
this
value != null
value != ""
valueHook :ValueHook
name == this.property
varTypedEStructuralFeature
feature :EStructuralFeature
varTypedEObject
SyncEcoreUtil.getCopy(valueHookNode, composedTarget, composedTarget.getContents())
valueHookNode :EObject
valueHookNodeCopy :EObject
1: valueHook.basicCompose(valueHookNodeCopy, feature, valuePrototype, null)

Figure A.19: `Setting.execute()` operation.

2.2 ... and passed as parameter to `compose()`.

3. This pattern (with 3.1 and 3.2) performs similar to Pattern 2, with the difference that it treats inverse matches (`inverse` = *true*). This means that for the variation point the copy related to the `composedSource` (and not the `composedTarget` as in Pattern 2) is requested from `SyncEcoreUtil` and for the reference point the copy related to the `composedTarget` is requested.

`Setting.execute()`, depicted in Figure A.19, works in a similar, but simpler, manner. Here, never nodes and edges are modified but only attributes. Therefore, this operation can be executed independent of the rest of a composition step (because attributes are always copied and never cross-referenced).

1. We navigate from a setting to the corresponding port and find the value hook for the setting. In cases of a homogeneous port, `fAllPorts()` returns multiple ports over which we iterate. We obtain the copy of the attributed node that holds the value hook (`valueHookNodeCopy`) using the `SyncEcoreUtil`. Since a string value has to be encapsulated into a value prototype before it can be passed to a basic composition operator (cf. Section 6.1), we temporarily create a value prototype that encapsulates the value of this setting. We then call `basicCompose()` on the value hook with the required parameters.

The operation `VariationPoint,compose()` encapsulates the composition of one variation point by calling the basic composition operators defined in Section 6.1. Each basic composition operator has the following parameters: `parent`, `feature`, `referenceNode`, and `variationNode`. `parent` and `feature` identify the `Setting` that is to be manipulated by the basic composition operator.[2] `referenceNode` is the node that will be added to the graph by the basic composition operator. `variationNode` is an optional parameter that is only bound if there is a variation node (and not only an edge list) the operator can work on.

There are two reasons for having the nodes `parent`, `referenceNode`, and `variationNode` as parameters although `VariationPoint` links to the node it types (cf. `varTypedEObject` in Figure 5.21). The first is that the basic operators always create only one new edge between the `parent` and `referenceNode`. The composition of one variation point can involve multiple

[2]The reason for not having one parameter of type `Setting` here is that `Setting` is an internal type in Ecore that cannot be used in interfaces.

edges and hence multiple basic composer calls. Thus, there are often multiple `parents` or `referenceNodes` where a basic composition operator has to be called for each. The second reason is that we always operate on copies of these nodes and not on the original nodes. The copies are produced and selected during composition as discussed above.

Figure A.20 shows the `VariationPoint.compose()` operation which is described in the following. The operation takes as parameters the reference point to be bound to this variation point (`rp`), the concrete `variationNode` and `referenceNode` copies that it operates on, the composed fragment (`result`) that will hold the copied nodes after composition, and a `context` containing the root `EObjects` of all graph fragments involved in the current composition. The context is needed to compute the list of incoming edges of a variation node $i(vn_G)$.

1. We distinguish four cases depending on whether the current variation point and the given reference point concern nodes (`varTypedEStructuralFeature` not set) or edge lists/attributes (`varTypedEStructuralFeature` set). First, if `varTypedEStructuralFeature` is set for both variation point and reference point the following is performed:
   1.1 Since the reference point types an edge list in this case, the reference node is the source of the edge list. The nodes that are actually composed are reached via following these edges. Therefore, we iterate over the value edges of the `Setting` that represents the edge list, since there is potentially more than one edge.
   1.2 to 1.4 For the first edge, we call the `basicCompose()` operator which will then forward to another basic operator (cf. Section 6.1). For all other edges, we directly call the `append()` operator to preserve the order of the list.
2. Second, we look at cases where only this variation point is referring to an edge list, but not the reference point.
   2.1 We need to distinguish between reference nodes and attributes. If the reference point is a `ValuePrototype`, we perform an attribute composition and pass the `ValuePrototype` itself as reference node to the `basicCompose()` (`referenceNode` is unbound in this case).
   2.2 Otherwise, we pass the reference node.
3. Third, we look at cases where only this reference point is referring to an edge list, but not the variation point.
   3.1 As in Pattern 1.1, we deal with an edge list possibly containing several entries we iterate over.
   3.2 (with 3.2.1 and 3.2.2) Here, we have a concrete variation node vn_G and need to handle all incoming edges $i(vn_G)$. To compute $i(vn_G)$ we use the `EcoreUtil.CrossReferencer` as described in Section 5.1.3. In this pattern we only consider non-containment edges ($e_G \notin T$). Similar as in Patterns 1.2–1.4, we call `basicCompose()` for the first value in the iteration of 3.1 and `append()` for all other values.
   3.3 to 3.5 In the case that this is a hook, we also change the containment edge of the variation node in a similar fashion as done for the other edges in Pattern 3.2.
4. (with 4.1) In the fourth and last case, this variation and the reference point are nodes. The composition of all incoming edges (and the containing edge in case of a hook) is performed in a similar fashion as in Patterns 3.2 and 3.3.

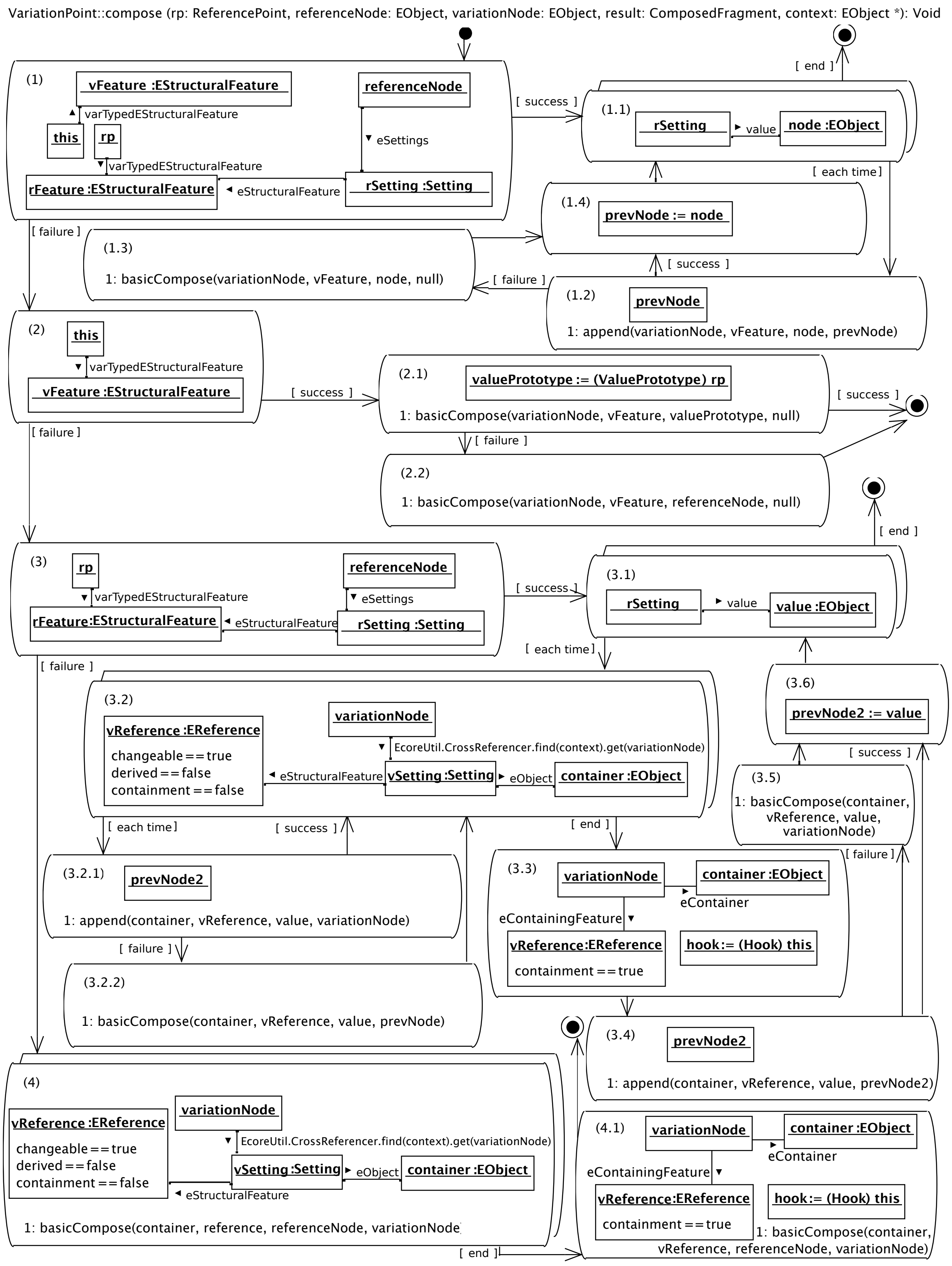

VariationPoint::compose (rp: ReferencePoint, referenceNode: EObject, variationNode: EObject, result: ComposedFragment, context: EObject *): Void
(1)
vFeature :EStructuralFeature
varTypedEStructuralFeature
this
rp
varTypedEStructuralFeature
rFeature :EStructuralFeature
eStructuralFeature
referenceNode
eSettings
rSetting :Setting
[success]
[failure]
[end]
(1.1)
rSetting
value
node :EObject
[each time]
(1.4)
prevNode := node
(1.3)
1: basicCompose(variationNode, vFeature, node, null)
[failure]
[success]
(1.2)
prevNode
1: append(variationNode, vFeature, node, prevNode)
(2)
this
varTypedEStructuralFeature
vFeature :EStructuralFeature
[success]
(2.1)
valuePrototype := (ValuePrototype) rp
1: basicCompose(variationNode, vFeature, valuePrototype, null)
[success]
[failure]
[failure]
(2.2)
1: basicCompose(variationNode, vFeature, referenceNode, null)
[end]
(3)
rp
varTypedEStructuralFeature
rFeature:EStructuralFeature
eStructuralFeature
referenceNode
eSettings
rSetting :Setting
[success]
(3.1)
rSetting
value
value :EObject
[each time]
[failure]
(3.2)
vReference :EReference
changeable==true
derived==false
containment==false
eStructuralFeature
variationNode
EcoreUtil.CrossReferencer.find(context).get(variationNode)
vSetting :Setting
eObject
container :EObject
(3.6)
prevNode2 := value
[success]
(3.5)
1: basicCompose(container, vReference, value, variationNode)
[each time]
[success]
[end]
[failure]
(3.2.1)
prevNode2
1: append(container, vReference, value, variationNode)
[failure]
(3.3)
variationNode
container :EObject
eContainer
eContainingFeature
vReference:EReference
containment==true
hook:= (Hook) this
(3.2.2)
1: basicCompose(container, vReference, value, prevNode)
(3.4)
prevNode2
1: append(container, vReference, value, prevNode2)
(4)
vReference :EReference
changeable==true
derived==false
containment==false
variationNode
EcoreUtil.CrossReferencer.find(context).get(variationNode)
vSetting :Setting
eObject
container :EObject
eStructuralFeature
1: basicCompose(container, reference, referenceNode, variationNode
[end]
(4.1)
variationNode
container :EObject
eContainer
eContainingFeature
vReference:EReference
containment==true
hook:= (Hook) this
1: basicCompose(container, vReference, referenceNode, variationNode)

Figure A.20: `VariationPoint.compose()` operation.

A.4 Fragment Instantiation in UCL

This section defines the semantics for interface creation in UCL composition programs that developers can interactively connect with composition links as described in Section 6.4.

The semantics are define in the operations `FragmentInstance.update()` (Figure A.21) and `PortInstance.update()` (Figure A.22). `FragmentInstance.update()`, is an updating operation that can be called repetitively. If the underlying fragment changes, `update()` can be re-evaluated. This is important because fragments already used in a composition program might still change over time. Another thing to notice here is that `update()` also removes invalid port instances. It does however only do so if the port instances are not linked (cf. next section).

1. We iterate over all `Ports` of the `Fragment` that is instantiated by this `FragmentInstance` (reached via `fFragment`) to check the instantiation of each of these `Ports`.
2. If a `PortInstance` of the current `Port` already exists (reached via `fPort`), no new instantiation needs to be performed for this port. Otherwise, we continue to perform a port instantiation.
3. We instantiate the current `Port` by creating a `PortInstance` and setting the `portName` to the name of the `Port` since the name uniquely identifies the `Port` (cf. Section 5.2.5). Furthermore, we create the `fPort` link between the `PortInstance` and the `Port`.
4. On each `PortInstances` we call `PortInstances.update()` (cf. Figure A.22) which checks if the port instance is still valid and updates the `Settings` of the port instance.

In the following we describe `PortInstances.update()` shown in Figure A.22.

1. We check a series of conditions, which lead to the removal of this `PortInstance` if they all fail. First we check, if the instantiated `Port` still exists by following `fPort`.
   1.1 If the port exists, we update the `Settings` (cf. Section 6.2.4) for this `PortInstance`. On a newly created instance this is the initialisation of the `Settings`. A `Setting` is created for each value hook of the current `Port`. First, we need to find the `HeterogeneousPort`...
   1.2 ...which can be encapsulated in a `HomogeneousPort`.
   1.3 For all value hooks in the port we do the following:
   1.4 If a `Setting` with the name of the current value hook already exists (`property` attribute of `Setting`) we do not need to create one. The name uniquely identifies the `ValueHook` (cf. Section 5.2.5).
   1.5 Else, we create a `Setting` for the `ValueHook` in the `HeterogeneousPort`. We set the `property` attribute of the `Setting` to the name of the `ValueHook`.
2. We check if there are `CompositionLinks` that have this `PortInstance` as `source`. In this case we do not remove this instance to allow the user to inspect the problem.
3. We check if there are `CompositionLinks` that have this `PortInstance` as `target`. In this case we do not remove this instance to allow the user to inspect the problem.
4. We check if there are `Settings` connected to this `PortInstance`. In this case we do not remove this instance to allow the user to inspect the problem.
5. If this `PortInstance` is invalid and not connected to anything else, it is destroyed.

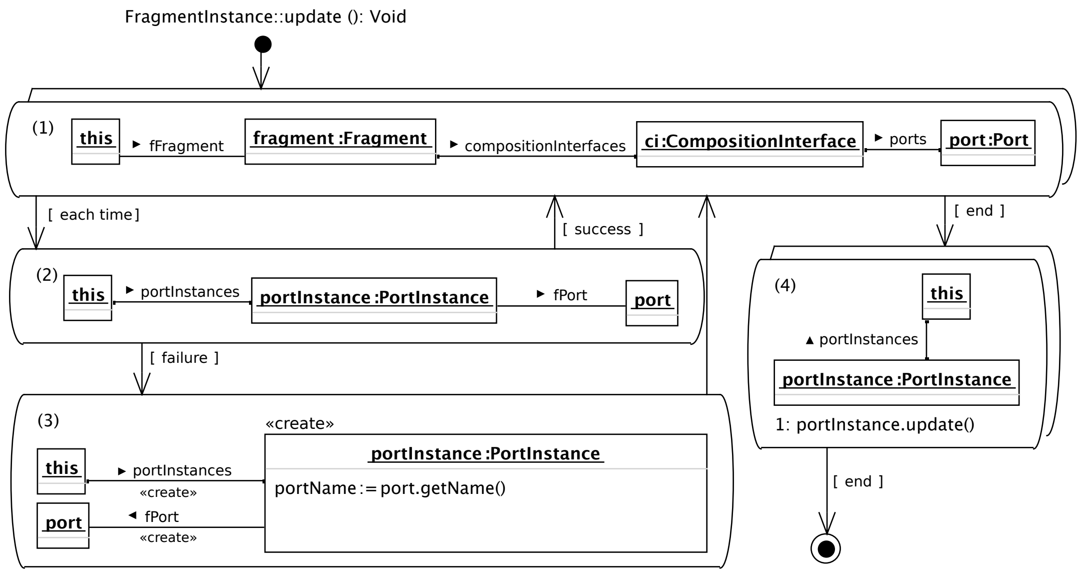


Figure A.21: `FragmentInstance.update()` operation.

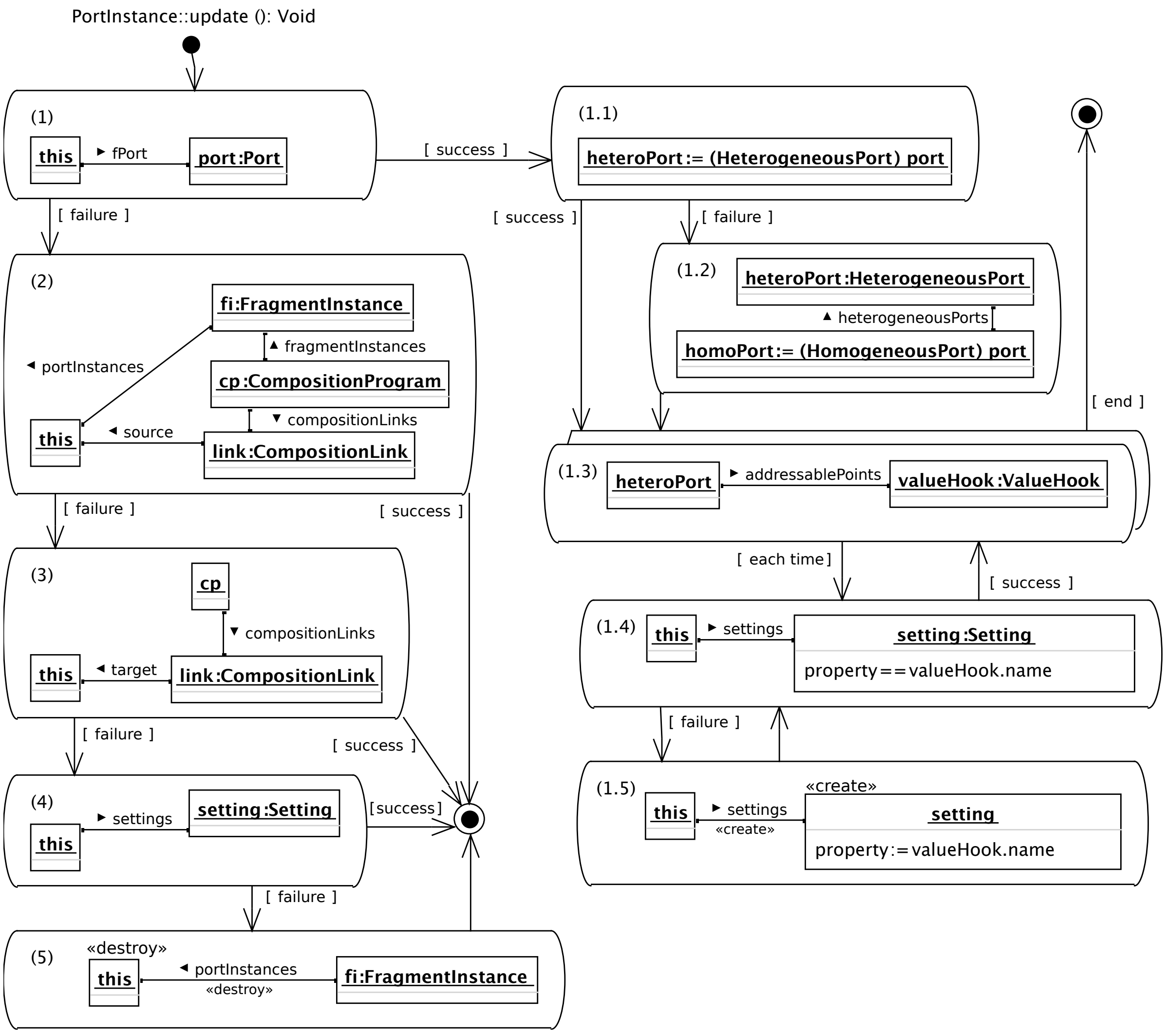


Figure A.22: `PortInstance.update()` operation.

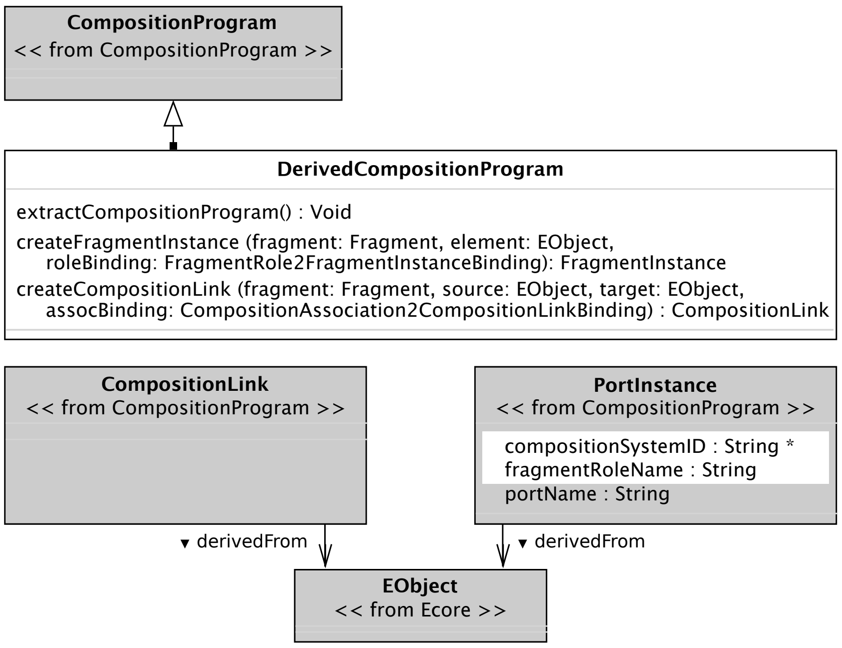

CompositionProgram
<< from CompositionProgram >>
DerivedCompositionProgram
extractCompositionProgram() : Void
createFragmentInstance (fragment: Fragment, element: EObject, roleBinding: FragmentRole2FragmentInstanceBinding): FragmentInstance
createCompositionLink (fragment: Fragment, source: EObject, target: EObject, assocBinding: CompositionAssociation2CompositionLinkBinding) : CompositionLink
CompositionLink
<< from CompositionProgram >>
PortInstance
<< from CompositionProgram >>
compositionSystemID : String *
fragmentRoleName : String
portName : String
▾ derivedFrom
▾ derivedFrom
EObject
<< from Ecore >>

Figure A.23: Extensions of the UCL metamodel (cf. Figure 6.13)

A.5 Composition Program Extraction

This section defines the semantics for composition program extraction based on REX$_{\mathrm{CL}}$ specifications described in Section 6.5.

Figure A.23 shows extensions of the composition program metamodel from Figure 6.13 required for REX$_{\mathrm{CL}}$ interpretation. `DerivedCompositionProgram` is added as subclass of `CompositionProgram` to distinguish manually defined and derived composition programs. Furthermore, we add a `derivedFrom` reference to `FragmentInstance` and `CompositionLink` that may point at any node in a graph (i.e., any `EObject`). These references are used to trace which element triggered a derivation. This is also useful for debugging to trace composition problems back to their source.

The `DerivedCompositionProgram` metaclass has a `extractCompositionProgram()` operation that is shown in Figure A.24. The `extractCompositionProgram()` operation follows the same strategy as a fragment's `computeCompositionInterface()` operation (cf. Figure A.3). Fragments, from which composition programs are derived, are linked to a `DerivedCompositionProgram` via `derivedFrom` links. The REX$_{\mathrm{CL}}$ specifications are linked with these fragments via `reuseExtension` links (cf. Figure A.2). In the `computeCompositionInterface()` operation, the `DerivedCompositionProgram` is filled with fragment instances and composition links based on these graph fragments and REX$_{\mathrm{CL}}$ specifications. Such a derived composition programs can then be executed (cf. Section 6.3) just as manually defined composition programs.

The details of `extractCompositionProgram()` are described in the following.

1. At first, all old existing composition links…

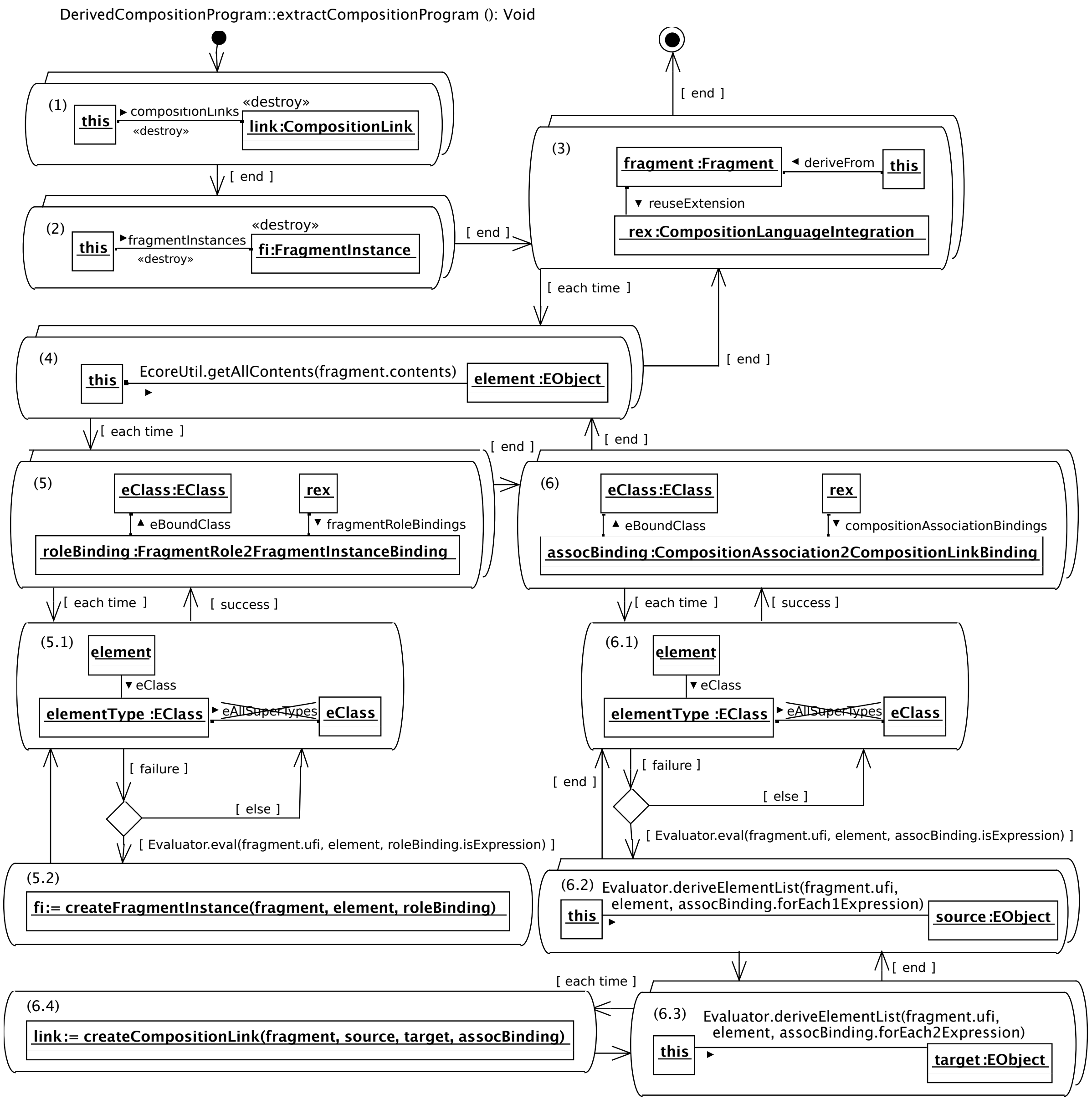

DerivedCompositionProgram::extractCompositionProgram (): Void
(1)
this
compositionLinks
«destroy»
«destroy»
link:CompositionLink
[end]
(2)
this
fragmentInstances
«destroy»
«destroy»
fi:FragmentInstance
[end]
(3)
fragment :Fragment
deriveFrom
this
reuseExtension
rex :CompositionLanguageIntegration
[end]
[each time]
(4)
this
EcoreUtil.getAllContents(fragment.contents)
element :EObject
[end]
[each time]
[end]
[end]
(5)
eClass:EClass
rex
eBoundClass
fragmentRoleBindings
roleBinding :FragmentRole2FragmentInstanceBinding
(6)
eClass:EClass
rex
eBoundClass
compositionAssociationBindings
assocBinding :CompositionAssociation2CompositionLinkBinding
[each time]
[success]
[each time]
[success]
(5.1)
element
eClass
elementType :EClass
~~eAllSuperTypes~~
eClass
(6.1)
element
eClass
elementType :EClass
~~eAllSuperTypes~~
eClass
[failure]
[else]
[Evaluator.eval(fragment.ufi, element, roleBinding.isExpression)]
[end]
[failure]
[else]
[Evaluator.eval(fragment.ufi, element, assocBinding.isExpression)]
(5.2)
fi:= createFragmentInstance(fragment, element, roleBinding)
(6.2)
Evaluator.deriveElementList(fragment.ufi, element, assocBinding.forEach1Expression)
this
source :EObject
[end]
[each time]
(6.4)
link:= createCompositionLink(fragment, source, target, assocBinding)
(6.3)
Evaluator.deriveElementList(fragment.ufi, element, assocBinding.forEach2Expression)
this
target :EObject

Figure A.24: `DerivedCompositionProgram.extractCompositionProgram()` operation.

2. ... and fragment instances are removed.
3. For all fragments and REX_{CL} specifications the composition program should be derived from, the following is performed.
4. We iterate over all nodes of the current graph fragment. We make use of the `EcoreUtil.getAllContents()` operation provided by EMF. The following is performed for each node.
5. We iterate over all fragment role bindings and find the affected `EClass` for each.
   5.1 We check if the current element is an instance of the `EClass`. For that we apply negation to find wrong matches. If the type is correct, we evaluate the `isExpression` using the Evaluator before continuing to Pattern 5.2.
   5.2 The `createFragmentInstance()` operation described below is used to create a new or update an existing fragment instance.
6. We iterate over all composition association bindings and find the affected `EClass` for each.
   6.1 We check if the current element is an instance of the `EClass`. For that we apply negation to find wrong matches. If the type is correct, we evaluate the `isExpression` using the Evaluator before continuing to Pattern 6.2.
   6.2 The `forEach1Expression` is evaluated to obtain a list of elements from which the source port instances of a composition links are derived.
   6.3 The `forEach2Expression` is evaluated to obtain a list of elements from which the target port instances of composition links are derived.
   6.4 The `createCompositionLinkInstance()` operation described below is used to create a new or update an existing composition link for each combination of elements that were obtained by evaluating the *for each* expressions in Patterns 6.2 and 6.3.

The `createFragmentInstance()` operation shown in Figure A.25 is described next:

1. First we check if the fragment instance, identified by its unique name, already exists. In any case, we add the element to the `derivedFrom` reference since different sources can cause the creation of the same fragment instance.
2. If the fragment instance did not yet exist it is created. `name` and `UFI` are derived using the Evaluator. The `reference` flag is set if the role binding is a `FragmentRole2FragmentReferenceBinding`.
3. If a target UFI expression was specified, the `targetUFI` is derived by evaluating it and the `target` flag is set to true.
4. After we ensured that the fragment instance exists in the previous patterns, we evaluate `PortType2SettingBindings`. For this we iterate over all these bindings and their setting derivation rules.
   4.1 We check if an instance of the current port type already exists. The port is identified by the name of the port type (that corresponds to the name of the port in the case of a static port type), the name of the fragment role the port belongs to, and the ID of the composition system the fragment role is defined in.
   4.2 If the required port instance does not yet exist, it is created.

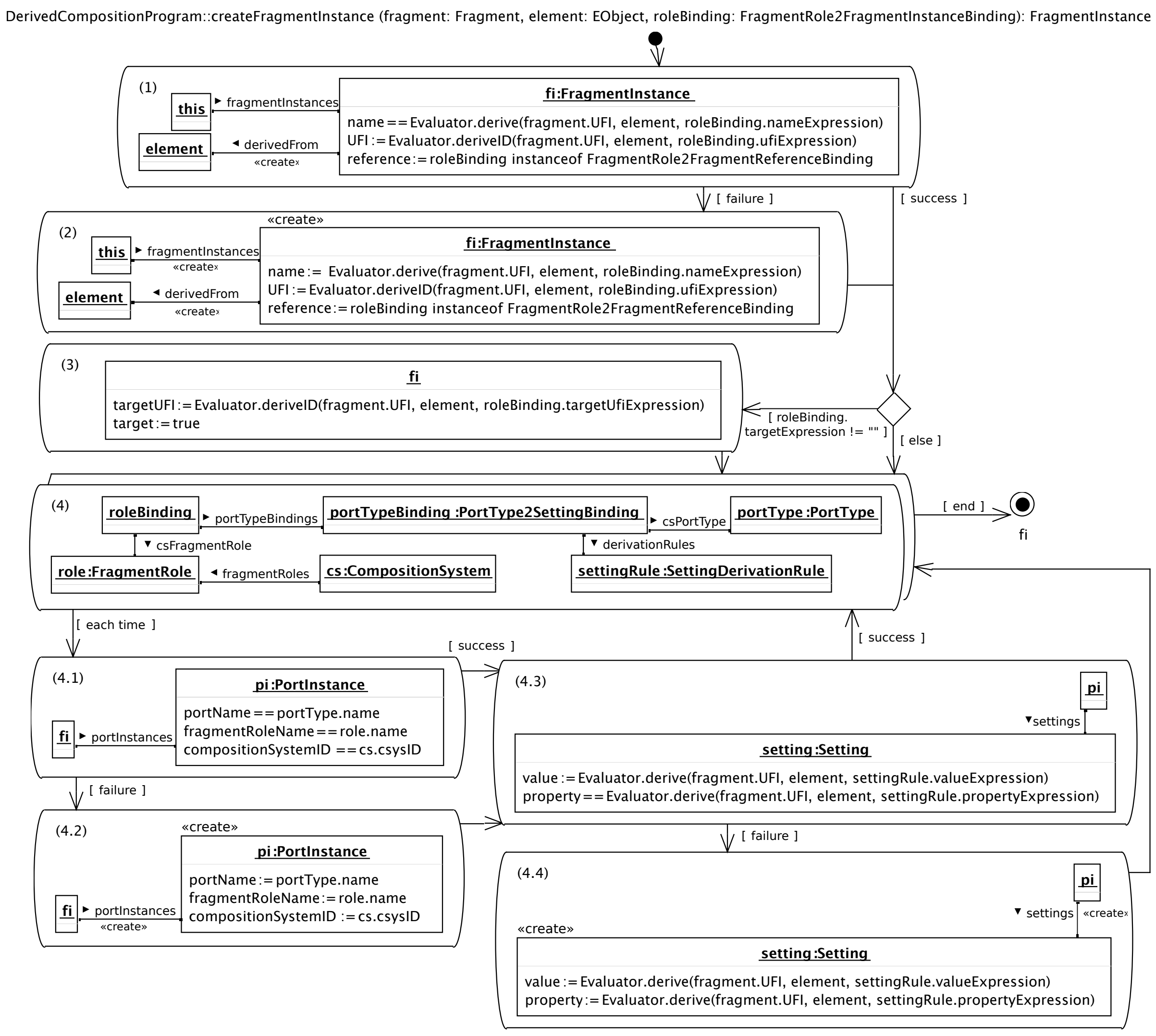

DerivedCompositionProgram::createFragmentInstance (fragment: Fragment, element: EObject, roleBinding: FragmentRole2FragmentInstanceBinding): FragmentInstance
(1)
this
fragmentInstances
element
derivedFrom
«create»
fi:FragmentInstance
name == Evaluator.derive(fragment.UFI, element, roleBinding.nameExpression)
UFI := Evaluator.deriveID(fragment.UFI, element, roleBinding.ufiExpression)
reference := roleBinding instanceof FragmentRole2FragmentReferenceBinding
[failure]
[success]
(2)
«create»
this
fragmentInstances
«create»
element
derivedFrom
«create»
fi:FragmentInstance
name := Evaluator.derive(fragment.UFI, element, roleBinding.nameExpression)
UFI := Evaluator.deriveID(fragment.UFI, element, roleBinding.ufiExpression)
reference := roleBinding instanceof FragmentRole2FragmentReferenceBinding
(3)
fi
targetUFI := Evaluator.deriveID(fragment.UFI, element, roleBinding.targetUfiExpression)
target := true
[roleBinding. targetExpression != ""]
[else]
(4)
roleBinding
portTypeBindings
portTypeBinding :PortType2SettingBinding
csPortType
portType :PortType
csFragmentRole
derivationRules
role :FragmentRole
fragmentRoles
cs :CompositionSystem
settingRule :SettingDerivationRule
[end]
fi
[each time]
[success]
(4.1)
fi
portInstances
pi :PortInstance
portName == portType.name
fragmentRoleName == role.name
compositionSystemID == cs.csysID
[failure]
(4.2)
«create»
fi
portInstances
«create»
pi :PortInstance
portName := portType.name
fragmentRoleName := role.name
compositionSystemID := cs.csysID
[success]
(4.3)
pi
settings
setting :Setting
value := Evaluator.derive(fragment.UFI, element, settingRule.valueExpression)
property == Evaluator.derive(fragment.UFI, element, settingRule.propertyExpression)
[failure]
(4.4)
pi
settings
«create»
«create»
setting :Setting
value := Evaluator.derive(fragment.UFI, element, settingRule.valueExpression)
property := Evaluator.derive(fragment.UFI, element, settingRule.propertyExpression)

Figure A.25: `DerivedCompositionProgram.createFragmentInstance()` operation.

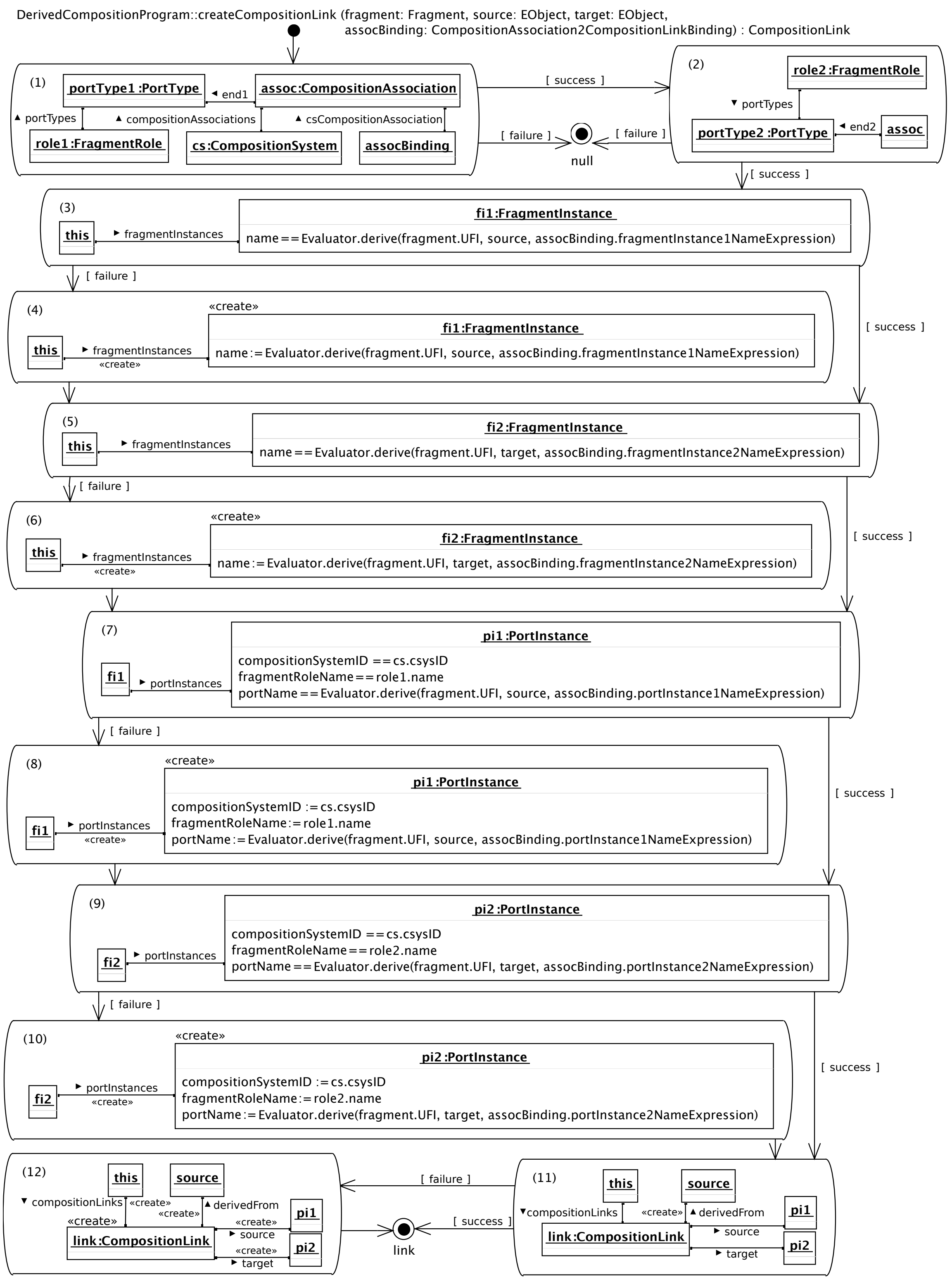

DerivedCompositionProgram::createCompositionLink (fragment: Fragment, source: EObject, target: EObject, assocBinding: CompositionAssociation2CompositionLinkBinding) : CompositionLink
(1)
portType1 :PortType
end1
assoc:CompositionAssociation
portTypes
compositionAssociations
csCompositionAssociation
role1 :FragmentRole
cs:CompositionSystem
assocBinding
[success]
[failure]
null
[failure]
(2)
role2 :FragmentRole
portTypes
portType2 :PortType
end2
assoc
[success]
(3)
this
fragmentInstances
fi1:FragmentInstance
name==Evaluator.derive(fragment.UFI, source, assocBinding.fragmentInstance1NameExpression)
[failure]
(4)
«create»
this
fragmentInstances
«create»
fi1:FragmentInstance
name:=Evaluator.derive(fragment.UFI, source, assocBinding.fragmentInstance1NameExpression)
[success]
(5)
this
fragmentInstances
fi2:FragmentInstance
name==Evaluator.derive(fragment.UFI, target, assocBinding.fragmentInstance2NameExpression)
[failure]
(6)
«create»
this
fragmentInstances
«create»
fi2:FragmentInstance
name:=Evaluator.derive(fragment.UFI, target, assocBinding.fragmentInstance2NameExpression)
[success]
(7)
fi1
portInstances
pi1 :PortInstance
compositionSystemID ==cs.csysID
fragmentRoleName==role1.name
portName==Evaluator.derive(fragment.UFI, source, assocBinding.portInstance1NameExpression)
[failure]
(8)
«create»
fi1
portInstances
«create»
pi1 :PortInstance
compositionSystemID :=cs.csysID
fragmentRoleName:=role1.name
portName:=Evaluator.derive(fragment.UFI, source, assocBinding.portInstance1NameExpression)
[success]
(9)
fi2
portInstances
pi2 :PortInstance
compositionSystemID ==cs.csysID
fragmentRoleName==role2.name
portName==Evaluator.derive(fragment.UFI, target, assocBinding.portInstance2NameExpression)
[failure]
(10)
«create»
fi2
portInstances
«create»
pi2 :PortInstance
compositionSystemID :=cs.csysID
fragmentRoleName:=role2.name
portName:=Evaluator.derive(fragment.UFI, target, assocBinding.portInstance2NameExpression)
[success]
(12)
this
source
compositionLinks
«create»
«create»
derivedFrom
«create»
link:CompositionLink
«create»
source
pi1
«create»
pi2
target
[failure]
link
[success]
(11)
this
source
compositionLinks
«create»
derivedFrom
pi1
link:CompositionLink
source
target
pi2

Figure A.26: `DerivedCompositionProgram.createCompositionLink()` operation.

4.3 We derive property and value utilising the current setting derivation rule. If the setting already exists for the derived property, the value in that setting is changed.

4.4 Otherwise the setting is created with the derived property/value pair.

The `createCompositionLink()` operation shown in Figure A.26 is described in the following:

1. For the first end, the port type and the fragment role is determined.
2. For the second end, the port type and the fragment role is determined as well.
3. to 10. We check for each fragment role and port type, if the corresponding fragment instance and port instance exists and create them if they do not.

11. After we ensured that the two port instances exist, we check if they are already linked. If this is the case, the current element is added to the `derivedFrom` reference of the link and we are done.
12. Otherwise, a composition link is created between the two port instances.

B Concern Dimensions of Chapter 8

This appendix contains the specifications that realise the different concern dimensions of the model-driven architecture defined with ModelSoC style in Chapter 8. For each concern dimension, a FraCol, a set of REX$_{\mathrm{CL}}$ specifications (to extract concern composition information), and a set of REX$_{\mathrm{CM}}$ specifications with fragments (defining unit prototypes) are provided.

B.1 Concern Dimension Usecase

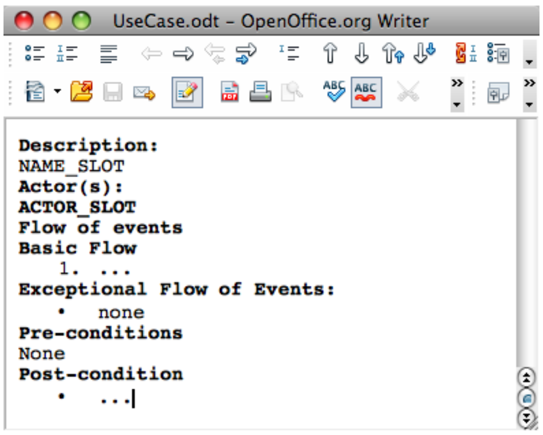
B.1.1 FraCol Definition

```
fracol org.reuseware.lib.systems.usecase.usecase {
  fragment role UseCase {
    static port type Name;
  }
}
```

B.1.2 OpenOffice REX$_{\mathrm{CL}}$ Specification

```
compositionlanguage org.reuseware.lib.systems.usecase.cl.odt
implements org.reuseware.lib.systems.usecase.usecase
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  fragment role UseCase {
    odfoffice::DocumentRoot {
      fragment = $'UseCase:'.concat(ufi.trimExtension().segment(-1))$
      ufi = $Sequence{'org','reuseware','lib','systems','usecase','lib','UseCase.'.concat(variant)}$
      target ufi = $ufi.replace('fragments','integrated').replace('odt',variant).trimExtension(
                        ).appendExtension(variant)$
      port type Name {
        $'name'$ = $ufi.trimExtension().segment(-1)$
      }
    }
}}
```

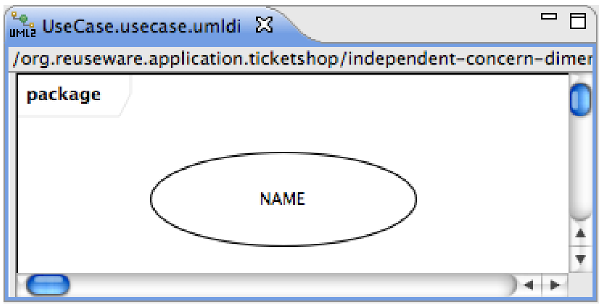
B.1.3 OpenOffice Unit Prototype



Description:
NAME_SLOT
Actor(s):
ACTOR_SLOT
Flow of events
Basic Flow
1. ...
Exceptional Flow of Events:
• none
Pre-conditions
None
Post-condition
• ...

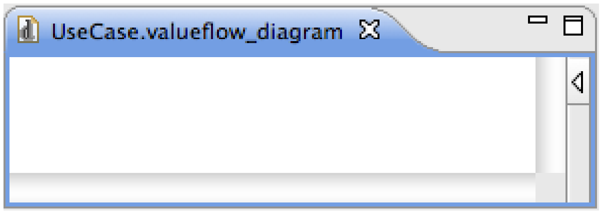
```
componentmodel org.reuseware.lib.systems.usecase.cm.odt
implements org.reuseware.lib.systems.usecase.usecase
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot {
  fragment role UseCase {
    port type Name {
      odftext::PType.mixed is value hook if $mixed->size() = 1 and mixed->at(1).getValue() = 'NAME_SLOT'$ {
        mode = $'bind'$
        point = $'name'$
      }
    }
  }
}
```

B.1.4 UML Use Case Unit Prototype


UseCase.usecase.umldi
/org.reuseware.application.ticketshop/independent-concern-dimer
package
NAME

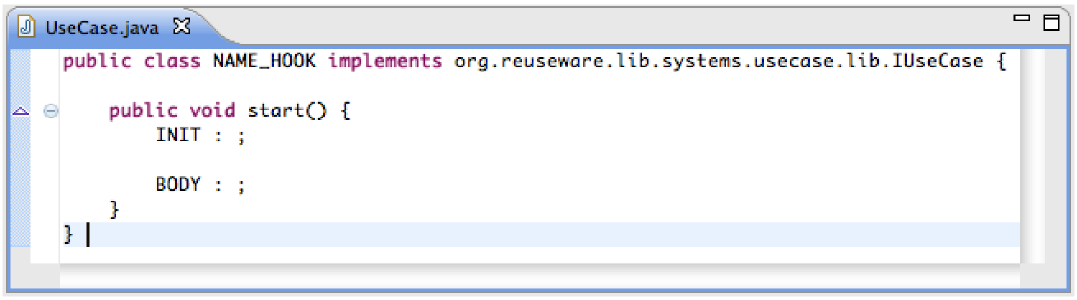
```
componentmodel org.reuseware.lib.systems.usecase.cm.usecase_uml
implements org.reuseware.lib.systems.usecase.usecase
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass Model {
  fragment role UseCase {
    port type Name {
      UseCase.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.1.5 Value Flow Unit Prototype

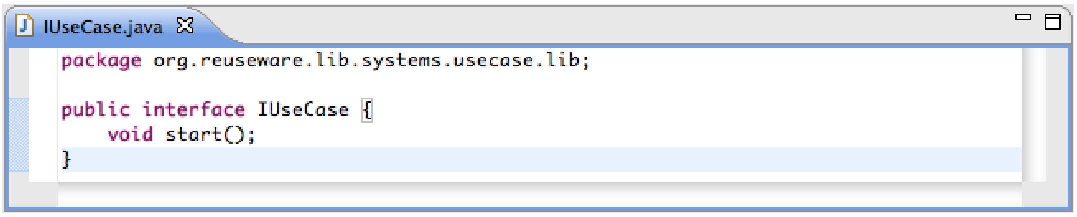


```
componentmodel org.reuseware.lib.systems.usecase.cm.valueflow
implements org.reuseware.lib.systems.usecase.usecase
epackages <http://www.emftext.org/language/valueflow>
rootclass Model {
  fragment role UseCase {
    port type Name {
      Model.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.1.6 Java Unit Prototype



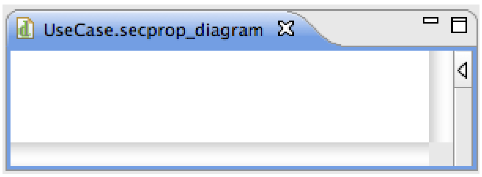
```
public class NAME_HOOK implements org.reuseware.lib.systems.usecase.lib.IUseCase {

    public void start() {
        INIT : ;

        BODY : ;
    }
}
```



```
package org.reuseware.lib.systems.usecase.lib;

public interface IUseCase {
    void start();
}
```

```
componentmodel org.reuseware.lib.systems.usecase.cm.java
implements org.reuseware.lib.systems.usecase.usecase
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role UseCase {
    homo port type Name {
      java::commons::NamedElement.name is value hook if $name.contains('NAME_HOOK')${
        point = $'name'$
        begin idx = $name.indexOf('NAME_HOOK')$
        end   idx = $name.indexOf('NAME_HOOK') + 'NAME_HOOK'.length() - 1$
      }
    }
  }
}
```

B.1.7 SecProp Unit Prototype



```
componentmodel org.reuseware.lib.systems.usecase.cm.secprop
implements org.reuseware.lib.systems.usecase.usecase
epackages <http://www.emftext.org/language/SecProp>
rootclass SecPropModel {
  fragment role UseCase {
    port type Name {}
  }
}
```

B.2 Concern Dimension Participation

B.2.1 FraCol Definition

```
fracol org.reuseware.lib.systems.participation.participation {
  fragment role Participant {
    static port type Contrib;
    static port type Name;
  }
  fragment role Collaboration {
    static port type Rec;
  }
  contributing association Participation {
    Participant.Contrib --> Collaboration.Rec
  }
}
```

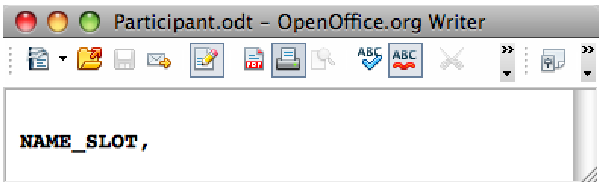
B.2.2 OpenOffice REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.participation.cl.odt
implements org.reuseware.lib.systems.participation.participation
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  fragment role Participant {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                         '_').concat(ufi.trimExtension().segment(-1))$
      ufi = $Sequence{'org','reuseware','lib','systems','participation','lib','Participant.'.concat(
                         variant)}$
      port type Name { $'name'$ = $mixed->at(1).getValue()$ }
    }
  }
  association Participation {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                         '_').concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'UseCase:'.concat(ufi.trimExtension().segment(-1))$
    }
} }
```

B.2.3 UML Use Case with Invariant REX_{CL} Specification

```
compositionlanguage org.reuseware.lib.systems.participation.cl.usecase_uml
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Participant {
    uml::Actor {
      fragment = $'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(
                     ufi.segment(-2))$
      ufi = $Sequence{'org','reuseware','lib','systems','participation','lib','Participant.'.concat(
                     variant)}$
      port type Name {  $'name'$ = $ufi.trimExtension().segment(-1)$ }
    }
  }
  association Participation {
    uml::Actor {
      fragment = $'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(
                     ufi.segment(-2))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
}
```

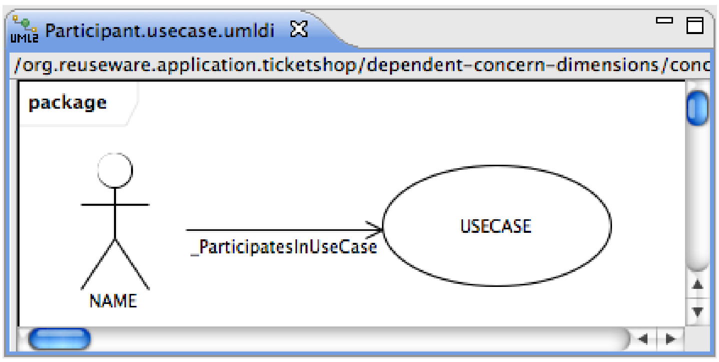
B.2.4 OpenOffice Unit Prototype



NAME_SLOT,

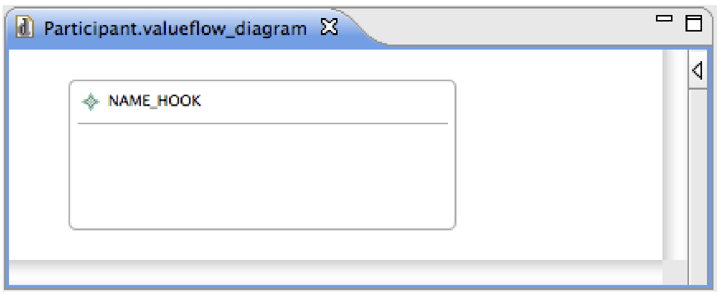
```
componentmodel org.reuseware.lib.systems.participation.cm.odt
implements org.reuseware.lib.systems.participation.participation
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot {
  fragment role Participant {
    port type Name {
      odftext::SpanType.mixed is value hook if $styleName = 'Actor'$ {
        mode = $'bind'$
        point = $'name'$
      }
    }
    port type Contrib {
      odftext::SpanType is prototype {}
    }
  }
  fragment role Collaboration {
    port type Rec {
      odftext::SpanType is hook if $styleName = 'Actor' and mixed->at(1).getValue() = 'ACTOR_SLOT'$ {
        remove = $true$
      }
    }
  }
}
```

B.2.5 UML Use Case Unit Prototype


Participant.usecase.umldi
/org.reuseware.application.ticketshop/dependent-concern-dimensions/conc
package
NAME
_ParticipatesInUseCase
USECASE

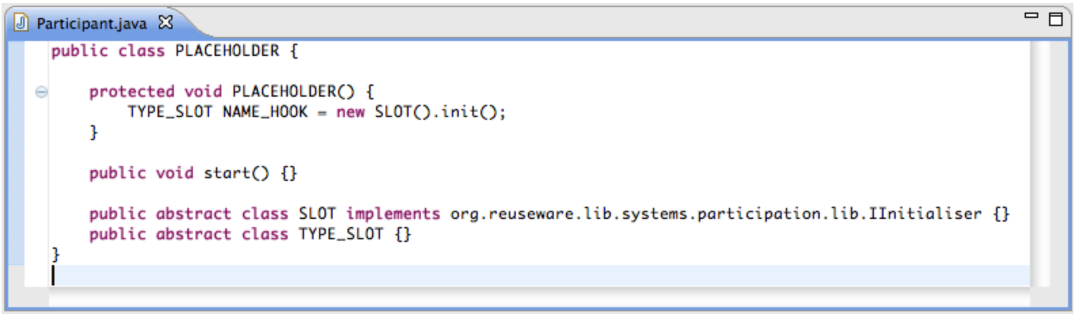
```
componentmodel org.reuseware.lib.systems.participation.cm.usecase_uml
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
  fragment role Participant {
    homo port type Name {
      uml::Actor.name is value hook {
        port type =$'ActoreName'$
        point = $'name'$
      }
      uml::Association.name is value hook {
        port type =$'AssociationName'$
        point = $'name'$
        begin idx = $'0'$
        end idx = $'0'$
      }
    }
    port type Contrib {
      uml::Actor is prototype {}
      uml::Association is prototype {}
      uml::UseCase is slot {}
    }
  }
  fragment role Collaboration {
    port type Rec {
      uml::Package.packagedElement is hook if $packagedElement->exists(e|e.oclIsKindOf(UseCase))$ {}
      uml::Package.ownedComment is hook if $packagedElement->exists(e|e.oclIsKindOf(UseCase))$ {}
      uml::UseCase is anchor {}
    }
  }
}
```

B.2.6 Value Flow Unit Prototype

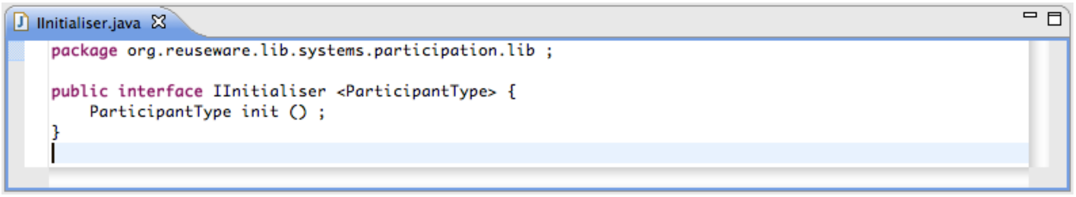


```
componentmodel org.reuseware.lib.systems.participation.cm.valueflow
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.emftext.org/language/valueflow>
rootclass valueflow::Model {
  fragment role Participant {
    port type Name {
      valueflow::Agent.name is value hook if $name = 'NAME_HOOK'$ {
        point = $'name'$
      }
    }
    port type Contrib {
      valueflow::Agent is prototype if $name = 'NAME_HOOK'$ {}
    }
  }
  fragment role Collaboration {
    port type Rec {
      valueflow::Model.agents is hook {}
    }
  }
}
```

B.2.7 Java Unit Prototype



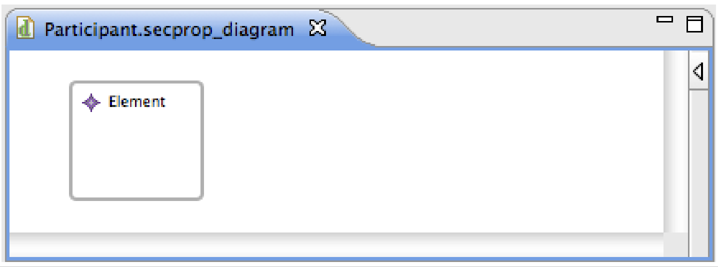
```
public class PLACEHOLDER {

    protected void PLACEHOLDER() {
        TYPE_SLOT NAME_HOOK = new SLOT().init();
    }

    public void start() {}

    public abstract class SLOT implements org.reuseware.lib.systems.participation.lib.IInitialiser {}
    public abstract class TYPE_SLOT {}
}
```



```
package org.reuseware.lib.systems.participation.lib ;

public interface IInitialiser <ParticipantType> {
    ParticipantType init () ;
}
```

```
componentmodel org.reuseware.lib.systems.participation.cm.java
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Participant {
    homo port type Name {
      java::commons::NamedElement.name is value hook if $name.contains('NAME_HOOK')${
        point = $'name'$
        begin idx = $name.indexOf('NAME_HOOK')$
        end   idx = $name.indexOf('NAME_HOOK') + 'NAME_HOOK'.length() - 1$
      }
    }
    port type Contrib {
      java::statements::StatementListContainer.statements is prototype if $self.oclIsKindOf(java::
            commons::NamedElement) and self.oclAsType(java::commons::NamedElement).name = 'PLACEHOLDER'$ {
        point = $'INIT'$
      }
    }
  }
  fragment role Collaboration {
    port type Rec {
      java::statements::JumpLabel is hook if $name.toUpperCase() = name$ {
        point = $name$
        remove = $true$
      }
    }
  }
}
```

B.2.8 SecProp Unit Prototype



```
componentmodel org.reuseware.lib.systems.participation.cm.secprop
implements org.reuseware.lib.systems.participation.participation
epackages <http://www.emftext.org/language/SecProp>
rootclass secprop::SecPropModel {
  fragment role Collaboration {
    port type Rec {
      secprop::SecPropModel.elements is hook {}
      secprop::SecPropModel.channels is hook {}
    }
  }

  fragment role Participant {
    port type Contrib {
      secprop::Element is prototype {}
    }
    port type Name {
      secprop::Element.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.3 Concern Dimension Exchange

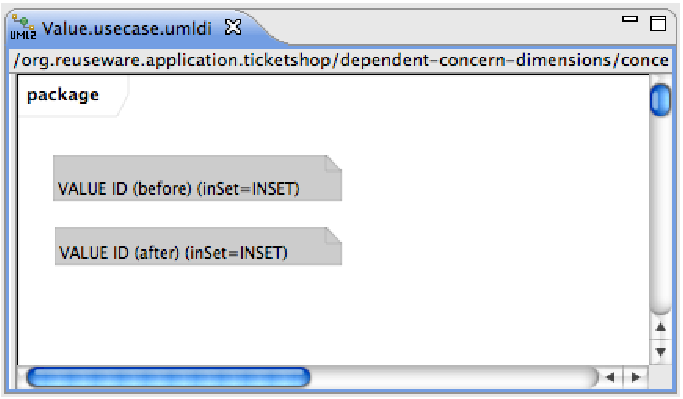
B.3.1 FraCol Definition

```
fracol org.reuseware.lib.systems.exchange.exchange {
  fragment role Container {
    static port type Rec;
  }
  fragment role Value {
    static port type Contrib;
    static port type Provider;
    static port type Consumer;
  }
  fragment role Provider {
    static port type Self;
  }
  fragment role Consumer {
    static port type Self;
  }
  contributing association Contribution {
     Value.Contrib --> Container.Rec
  }
  configuring association Provide {
     Value.Provider --> Provider.Self
  }
  configuring association Consume {
     Value.Consumer --> Consumer.Self
  }
}
```

B.3.2 UML Use Case with Invariant REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.exchange.cl.usecase_uml
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Value {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      ufi = $Sequence{'org','reuseware','lib','systems','exchange','lib','Value.'.concat(variant)}$
      port type Provider {
        $'value'$ = $body.split(' ')->at(1)$
        $'name'$  = $body.split(' ')->at(1).concat('_').concat(body.split(' ')->at(2))$
        $'ID'$    = $body.split(' ')->at(2)$
        $'inSet'$    = $body.contains('inSet')$
      }
    }
  }
  association Contribution {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
  association Provide {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      -->
      fragment=$'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(ufi.segment(-2))$
    }
  }
  fragment role Value {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      ufi = $Sequence{'org','reuseware','lib','systems','exchange','lib','Value.'.concat(variant)}$
      port type Consumer {
        $'value'$ = $body.split(' ')->at(1)$
        $'name'$  = $body.split(' ')->at(1).concat('_').concat(body.split(' ')->at(2))$
        $'ID'$    = $body.split(' ')->at(2)$
        $'inSet'$    = $body.contains('inSet')$
      }
    }
  }
  association Contribution {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
  association Consume {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
      -->
      fragment=$'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(ufi.segment(-2))$
    }
  }
}
```

B.3.3 UML Use Case Unit Prototype

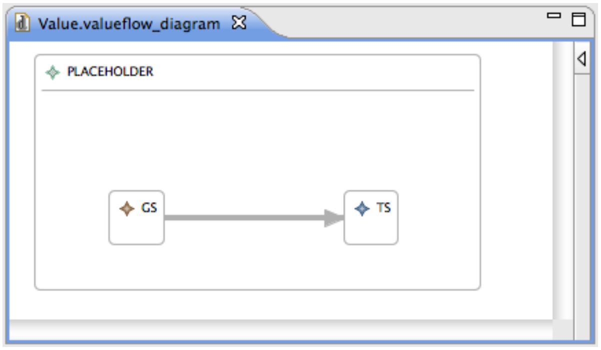

Value.usecase.umldi
/org.reuseware.application.ticketshop/dependent-concern-dimensions/conce
package

VALUE ID (before) (inSet=INSET)

VALUE ID (after) (inSet=INSET)

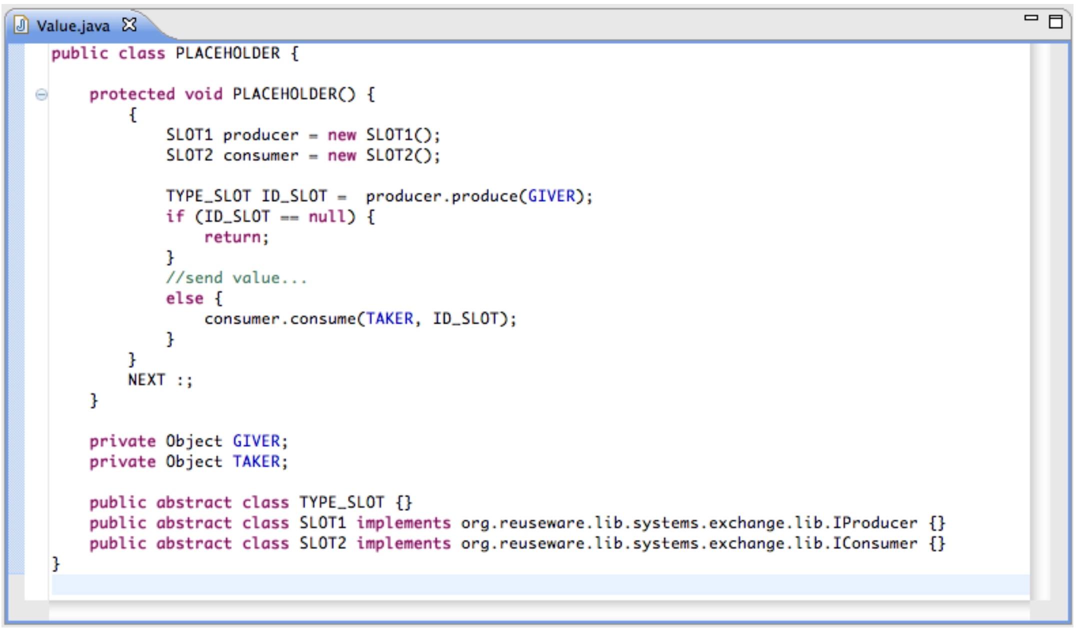
```
componentmodel org.reuseware.lib.systems.exchange.cm.usecase_uml
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
  fragment role Container {
    port type Rec {
      uml::Package.ownedComment is hook if $self.oclIsTypeOf(Package)$ {}
    }
  }
  fragment role Value {
    port type Contrib{
      uml::Comment is prototype {}
    }
    port type Provider {
      uml::Comment.annotatedElement is slot if $body.contains('(before)')${}
      uml::Comment.body is value hook if $body.contains('(before)')$ {
        point = $'value'$
        begin idx = $body.indexOf('VALUE')$
        end idx = $body.indexOf('VALUE') + 'VALUE'.length() + 1$
      }
      uml::Comment.body is value hook if $body.contains('(before)')$ {
        point = $'ID'$
        begin idx = $body.indexOf('ID')$
        end idx = $body.indexOf('ID') + 'ID'.length() + 1$
      }
      uml::Comment.body is value hook if $body.contains('(before)')$ {
        point = $'inSet'$
        begin idx = $body.indexOf('INSET')$
        end idx = $body.indexOf('INSET') + 'INSET'.length() + 1$
      }
    }
    port type Consumer {
      uml::Comment.annotatedElement is slot if $body.contains('(after)')${}
      uml::Comment.body is value hook if $body.contains('(after)')$ {
        point = $'value'$
        begin idx = $body.indexOf('VALUE')$
        end idx = $body.indexOf('VALUE') + 'VALUE'.length() + 1$
      }
      uml::Comment.body is value hook if $body.contains('(after)')$ {
        point = $'ID'$
        begin idx = $body.indexOf('ID')$
        end idx = $body.indexOf('ID') + 'ID'.length() + 1$
      }
      uml::Comment.body is value hook if $body.contains('(after)')$ {
        point = $'inSet'$
        begin idx = $body.indexOf('INSET')$
        end idx = $body.indexOf('INSET') + 'INSET'.length() + 1$
      }
    }
  }
  fragment role Provider {
    port type Self {
      uml::Actor is anchor {}
    }
  }
  fragment role Consumer {
    port type Self {
      uml::Actor is anchor {}
    }
  }
}
```

B.3.4 Value Flow Unit Prototype



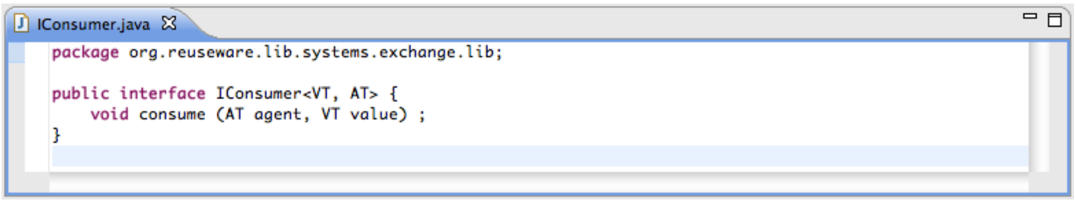
```
componentmodel org.reuseware.lib.systems.exchange.cm.valueflow
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.emftext.org/language/valueflow>
rootclass valueflow::Model {
  fragment role Container {
    port type Rec {}
  }
  fragment role Value {
    port type Contrib{}
    port type Provider {
      valueflow::GiveState is prototype {}
      valueflow::GiveState._value is value hook {
        point = $'value'$
      }
      valueflow::GiveState.name is value hook {
        point = $'name'$
      }
      valueflow::GiveState.ID is value hook {
        point = $'ID'$
      }
    }
    port type Consumer {
      valueflow::TakeState is prototype  if $name <> 'NEXT'$ {}
      valueflow::TakeState.name is value hook  if $name <> 'NEXT'$ {
        point = $'name'$
      }
    }
  }
  fragment role Provider {
    port type Self {
      valueflow::Agent.states is hook if $name = 'NAME_HOOK'$  {}
    }
  }
  fragment role Consumer {
    port type Self {
      valueflow::Agent.states is hook if $name = 'NAME_HOOK'$  {}
    }
  }
}
```

B.3.5 Java Unit Prototype



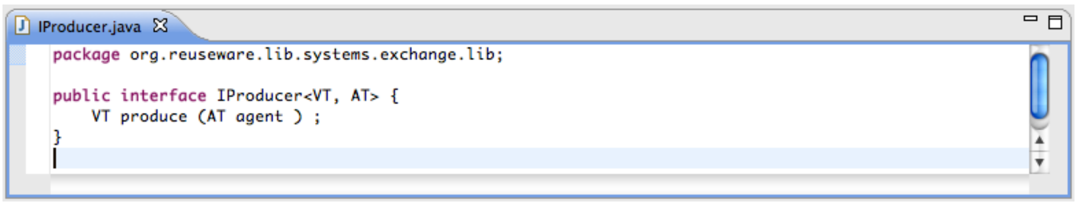
Value.java

```
public class PLACEHOLDER {

    protected void PLACEHOLDER() {
        {
            SLOT1 producer = new SLOT1();
            SLOT2 consumer = new SLOT2();

            TYPE_SLOT ID_SLOT =  producer.produce(GIVER);
            if (ID_SLOT == null) {
                return;
            }
            //send value...
            else {
                consumer.consume(TAKER, ID_SLOT);
            }
        }
        NEXT :;
    }

    private Object GIVER;
    private Object TAKER;

    public abstract class TYPE_SLOT {}
    public abstract class SLOT1 implements org.reuseware.lib.systems.exchange.lib.IProducer {}
    public abstract class SLOT2 implements org.reuseware.lib.systems.exchange.lib.IConsumer {}
}
```



IConsumer.java

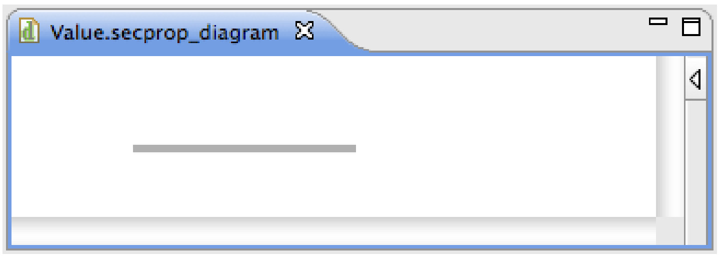
```
package org.reuseware.lib.systems.exchange.lib;

public interface IConsumer<VT, AT> {
    void consume (AT agent, VT value) ;
}
```



```
package org.reuseware.lib.systems.exchange.lib;

public interface IProducer<VT, AT> {
    VT produce (AT agent ) ;
}
```

```
componentmodel org.reuseware.lib.systems.exchange.cm.java
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Container {
    port type Rec {}
  }
  fragment role Value {
    port type Contrib {}
    port type Provider {
      java::commons::NamedElement is slot if $name = 'GIVER'${}
      java::commons::NamedElement.name is value hook if $name = 'ID_SLOT'$ {
        point = $'ID'$
      }
    }
    port type Consumer {
      java::commons::NamedElement is slot if $name = 'TAKER'${}
    }
  }

  fragment role Provider {
    port type Self {
      java::commons::NamedElement is anchor if $name.contains('NAME_HOOK')${}
    }
  }

  fragment role Consumer {
    port type Self {
      java::commons::NamedElement is anchor if $name.contains('NAME_HOOK')${}
    }
  }

}
```

B.3.6 SecProp Unit Prototype



```
componentmodel org.reuseware.lib.systems.exchange.cm.secprop
implements org.reuseware.lib.systems.exchange.exchange
epackages <http://www.emftext.org/language/SecProp>
rootclass secprop::SecPropModel {
  fragment role Container {
    port type Rec {
      secprop::SecPropModel.elements is hook {}
      secprop::SecPropModel.channels is hook {}
      secprop::SecPropModel.data is hook {}
    }
  }
  fragment role Provider {
    port type Self {
      secprop::Element is anchor {}
    }
  }
  fragment role Consumer {
    port type Self {
      secprop::Element is anchor {}
    }
  }
  fragment role Value {
    port type Contrib{
      secprop::Channel is prototype {}
      secprop::Data is prototype {}
    }
    port type Provider {
      secprop::Channel.source is slot {}
      secprop::Data.name is value hook {
        point = $'value'$
      }
      secprop::Channel.name is value hook {
        point = $'name'$
      }
    }
    port type Consumer {
      secprop::Channel._target is slot {}
    }
  }
}
```

B.4 Concern Dimension Flow

B.4.1 FraCol Definition

```
fracol org.reuseware.lib.systems.flow.flow {
  fragment role Participant1 {
    static port type Self;
    static port type Next;
  }
  fragment role Participant2 {
    static port type Self;
    static port type Next;
  }
  configuring association Flow1To1 {
    Participant1.Next --> Participant1.Self
  }
  configuring association Flow1To2 {
    Participant1.Next --> Participant2.Self
  }
  configuring association Flow2To1 {
    Participant2.Next --> Participant1.Self
  }
  configuring association Flow2To2 {
    Participant2.Next --> Participant2.Self
  }
}
```

B.4.2 Value Flow REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.flow.cl.valueflow
implements org.reuseware.lib.systems.flow.flow
epackages <http://www.emftext.org/language/valueflow>
rootclass valueflow::Model
ucpi = $ufi.replace('fragments','integrated').replace('valueflow',variant).trim(2).append(
'Main').appendExtension('fc')$ {
  association Flow1To1 {
    valueflow::GiveState if $nextState.oclIsTypeOf(GiveState)$ {
      fragment = $'Value:'.concat(name)$
      --> fragment = $'Value:'.concat(nextState.name)$
    }
  }
  association Flow1To2 {
    valueflow::GiveState if $nextState.oclIsTypeOf(TakeState)$ {
      fragment = $'Value:'.concat(name)$
      --> fragment = $'Value:'.concat(nextState.name)$
    }
  }
  association Flow2To1 {
    valueflow::TakeState if $nextState.oclIsTypeOf(GiveState)$ {
      fragment = $'Value:'.concat(name)$
      --> fragment = $'Value:'.concat(nextState.name)$
    }
  }
  association Flow2To2 {
    valueflow::TakeState if $nextState.oclIsTypeOf(TakeState)$ {
      fragment = $'Value:'.concat(name)$
      --> fragment = $'Value:'.concat(nextState.name)$
    }
  }
}
```

B.4.3 Value Flow Unit Prototype

```
componentmodel org.reuseware.lib.systems.flow.cm.valueflow
implements org.reuseware.lib.systems.flow.flow
epackages <http://www.emftext.org/language/valueflow>
rootclass valueflow::Model {
  fragment role Participant1 {
    port type Self {
      valueflow::GiveState is anchor {}
    }
    port type Next {
      valueflow::GiveState.nextState is slot {}
    }
  }
  fragment role Participant2 {
    port type Self {
      valueflow::TakeState is anchor {}
    }
    port type Next {
      valueflow::TakeState.nextState is slot {}
    }
  }
}
```

B.4.4 Java Unit Prototype

```
componentmodel org.reuseware.lib.systems.flow.cm.java
implements org.reuseware.lib.systems.flow.flow
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Participant1 {
    port type Self {
      java::statements::StatementListContainer.statements is prototype if $self.oclIsKindOf(java::
          commons::NamedElement) and self.oclAsType(java::commons::NamedElement).name = 'PLACEHOLDER'$ {
      }
    }
    port type Next {
      java::statements::JumpLabel is hook if $name = 'NEXT'$ {
        remove = $true$
      }
    }
  }
  fragment role Participant2 {
    port type Self {}
    port type Next {
      java::statements::JumpLabel is hook if $name = 'NEXT'$ {
        remove = $true$
      }
    }
  }
}
```

B.5 Concern Dimension Trigger

B.5.1 FraCol Definition

```
fracol org.reuseware.lib.systems.trigger.trigger {
  fragment role Container {
    static port type Rec;
  }
  fragment role Trigger {
    static port type Contrib;
  }
  contributing association Contribution {
    Trigger.Contrib --> Container.Rec
  }
}
```

B.5.2 Value Flow REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.trigger.cl.valueflow
implements org.reuseware.lib.systems.trigger.trigger
epackages <http://www.emftext.org/language/valueflow>
rootclass Model
ucpi = $ufi.replace('fragments','integrated').replace('valueflow',variant).trim(2).append('Main.fc')$ {
  association Contribution {
    GiveState if $previousState.oclIsUndefined()$  {
      fragment = $'Value:'.concat(name)$
      --> fragment = $'UseCase:'.concat(ufi.trim(1).segment(-1))$
    }
  }
}
```

B.5.3 Java Unit Prototype

```
componentmodel org.reuseware.lib.systems.trigger.cm.java
implements org.reuseware.lib.systems.trigger.trigger
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Container {
    port type Rec {
      java::statements::JumpLabel is hook if $name = 'BODY'$ {
        remove = $'true'$
      }
    }
  }
  fragment role Trigger {
    port type Contrib {
      java::statements::StatementListContainer.statements is prototype if $self.oclIsKindOf(java::
          commons::NamedElement) and self.oclAsType(java::commons::NamedElement).name = 'PLACEHOLDER'$ {
      }
    }
  }
}
```

B.6 Concern Dimension Factory

B.6.1 FraCol Definition

```
fracol org.reuseware.lib.systems.factory.factory {
  fragment role Factory {
    static port type Product;
  }
  fragment role Receiver {
    static port type Slot;
  }
  fragment role TwinReceiver {
    static port type Slot1;
    static port type Slot2;
  }
  configuring association Produce {
    Factory.Product --> Receiver.Slot
  }
  configuring association Produce1 {
    Factory.Product --> TwinReceiver.Slot1
  }
  configuring association Produce2 {
    Factory.Product --> TwinReceiver.Slot2
  }
}
```

B.6.2 OpenOffice REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.factory.cl.odt
implements org.reuseware.lib.systems.factory.factory
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  reference fragment role Factory {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Factory:'.concat(mixed->at(1).getValue().oclAsType(String))$
      ufi =  $ufi.replace('odt',variant).trim(1).append(mixed->at(1).getValue().oclAsType(
              String).toLowerCase()).append(mixed->at(1).getValue().oclAsType(String).concat(
                    'Initialiser')).appendExtension(variant)$
    }
  }
  association Produce {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Factory:'.concat(mixed->at(1).getValue().oclAsType(String))$
      --> fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                          '_').concat(ufi.trimExtension().segment(-1))$
    }
  }
}
```

B.6.3 UML Use Case with Invariant REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.factory.cl.usecase_uml
implements org.reuseware.lib.systems.factory.factory
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  reference fragment role Factory {
    uml::Actor {
      fragment = $'Factory:'.concat(ufi.trimExtension().segment(-1))$
      ufi =  $ufi.replace('usecase.uml',variant).trim(2).append(name.toLowerCase()).append(
                  name.concat('Initialiser')).appendExtension(variant)$
    }
  }
  association Produce {
    uml::Actor {
      fragment = $'Factory:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'Participant:'.concat(ufi.trimExtension().segment(-1)).concat('_').concat(
                    ufi.segment(-2))$
    }
  }
  reference fragment role Factory {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Factory:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(
                    ' ')->at(2)).concat('Producer')$
      ufi = $ufi.replace('usecase.uml',variant).trim(2).append(ufi.trimExtension().segment(
              -1).toLowerCase()).append('Produce'.concat(body.split(' ')->at(1))).appendExtension(variant)$
    }
  }
  reference fragment role Factory {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Factory:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(
                          ' ')->at(2)).concat('Consumer')$
      ufi = $ufi.replace('usecase.uml',variant).trim(2).append(ufi.trimExtension().segment(
        -1).toLowerCase()).append('Consume'.concat(body.split(' ')->at(1))).appendExtension(variant)$
    }
  }
  association Produce1 {
    uml::Comment if $body.contains('before')$ {
      fragment = $'Factory:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(
                    ' ')->at(2)).concat('Producer')$
      -->
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
    }
  }
  association Produce2 {
    uml::Comment if $body.contains('after')$ {
      fragment = $'Factory:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(
                        ' ')->at(2)).concat('Consumer')$
      -->
      fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
    }
  }
}
```

B.6.4 Java Unit Prototype

```
componentmodel org.reuseware.lib.systems.factory.cm.java
implements org.reuseware.lib.systems.factory.factory
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Receiver {
    port type Slot {
      java::commons::NamedElement is slot if $name = 'SLOT'${}
    }
  }
  fragment role TwinReceiver {
    port type Slot1 {
      java::commons::NamedElement is slot if $name = 'SLOT1'${}
    }
    port type Slot2 {
      java::commons::NamedElement is slot if $name = 'SLOT2'${}
    }
  }
  fragment role Factory {
    port type Product {
      java::classifiers::Class is anchor if $eContainer().oclIsTypeOf(java::containers::CompilationUnit)$ {}
    }
  }
}
```

B.7 Concern Dimension Class

B.7.1 FraCol Definition

```
fracol org.reuseware.lib.systems.class.class {
  fragment role Core {
    static port type Rec;
  }
  fragment role Class {
    static port type Contrib;
    static port type Self;
    static port type Name;
  }
  contributing association Contribution {
    Class.Contrib --> Core.Rec
  }
}
```

B.7.2 OpenOffice REX$_{CL}$ Specification

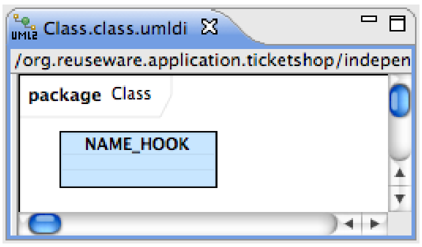
```
compositionlanguage org.reuseware.lib.systems.class.cl.odt
implements org.reuseware.lib.systems.class.class
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
```

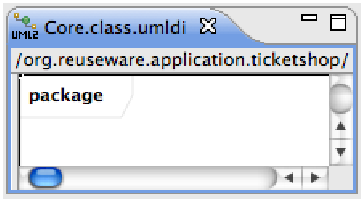
```
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  fragment role Core {
    odfoffice::DocumentRoot {
      fragment = $'CLASS_CORE'$
      ufi = $Sequence{'org','reuseware','lib','systems','class','lib','Core.'.concat(variant)}$
      target ufi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append(
                    'Main').appendExtension(variant)$
    }
  }
  fragment role Class {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Class:'.concat(mixed->at(1).getValue().oclAsType(String))$
      ufi = $Sequence{'org','reuseware','lib','systems','class','lib','Class.'.concat(variant)}$
      port type Name {
        $'name'$ = $mixed->at(1).getValue()$
      }
    }
  }
  association Contribution {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Class:'.concat(mixed->at(1).getValue().oclAsType(String))$
      --> fragment = $'CLASS_CORE'$
    }
  }
}
```

B.7.3 UML Use Case with Invariant REX_{CL} Specification

```
compositionlanguage org.reuseware.lib.systems.class.cl.usecase_uml
implements org.reuseware.lib.systems.class.class
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Class {
    uml::Actor {
      fragment = $'Class:'.concat(ufi.trimExtension().segment(-1))$
      ufi = $Sequence{'org','reuseware','lib','systems','class','lib','Class.'.concat(variant)}$
      port type Name {
        $'name'$ = $ufi.trimExtension().segment(-1)$
      }
    }
  }
  association Contribution {
    uml::Actor {
      fragment = $'Class:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'CLASS_CORE'$
    }
  }
}
```

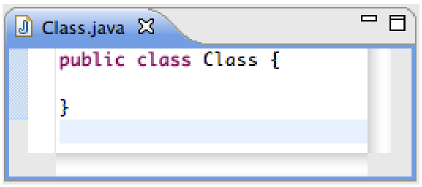
B.7.4 UML Class Unit Prototype



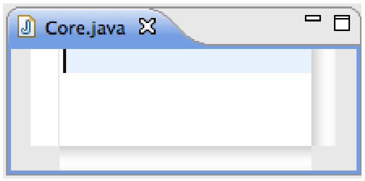


```
componentmodel org.reuseware.lib.systems.class.cm.uml
implements org.reuseware.lib.systems.class.class
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
  fragment role Core {
    port type Rec {
      uml::Package.packagedElement is hook if $self.eContainer().oclIsTypeOf(Model)$ {}
    }
  }
  fragment role Class {
    port type Contrib {
      uml::Class is prototype {}
    }
    port type Self {
      uml::Class is anchor {}
    }
    port type Name {
      uml::Class.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.7.5 Java Unit Prototype



```
public class Class {

}
```



```
componentmodel org.reuseware.lib.systems.class.cm.java
implements org.reuseware.lib.systems.class.class
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit, java::containers::EmptyModel {
  fragment role Core {
    port type Rec {
      java::containers::EmptyModel is hook {}
    }
  }
  fragment role Class {
    port type Contrib {
      java::containers::CompilationUnit is prototype {}
    }
    port type Self {
      java::classifiers::Class is anchor {}
      java::classifiers::Class.members is hook {}
    }
    port type Name {
      java::classifiers::Class.name is value hook if $eContainer().oclIsTypeOf(java::
                                                       containers::CompilationUnit)$ {
        point = $'name'$
      }
    }
  }
}
```

B.8 Concern Dimension Dataclass

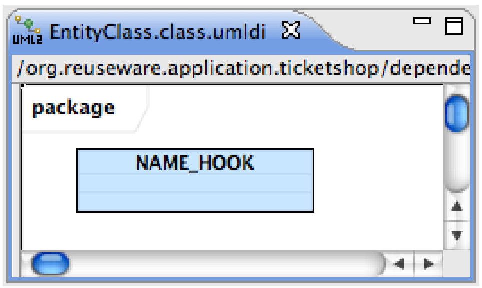
B.8.1 FraCol Definition

```
fracol org.reuseware.lib.systems.dataclass.dataclass {
  fragment role Core {
    static port type Rec;
  }
  fragment role Class {
    static port type Contrib;
    static port type Self;
    static port type Name;
  }
  contributing association Contribution {
    Class.Contrib --> Core.Rec
  }
}
```

B.8.2 UML Use Case with Invariant REX$_{CL}$ Specification

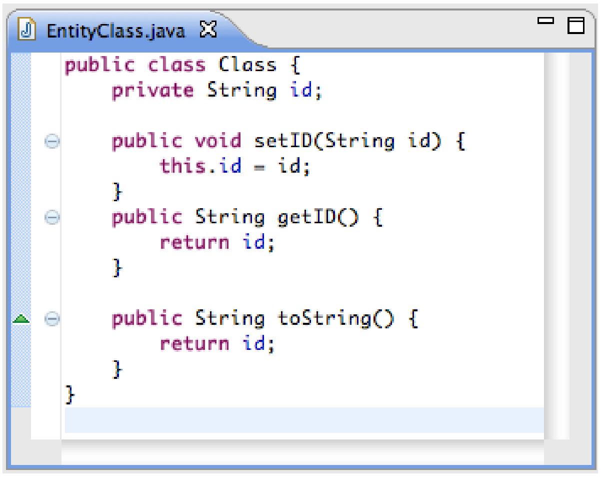
```
compositionlanguage org.reuseware.lib.systems.dataclass.cl.usecase_uml
implements org.reuseware.lib.systems.dataclass.dataclass
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Class {
    uml::Comment {
      fragment = $'Class:'.concat(body.split(' ')->at(1))$
      ufi = $Sequence{'org','reuseware','lib','systems','dataclass','lib','EntityClass.'.concat(variant)}$
      port type Name {
        $'name'$ = $body.split(' ')->at(1)$
      }
    }
  }
  association Contribution {
    uml::Comment {
      fragment = $'Class:'.concat(body.split(' ')->at(1))$
      --> fragment = $'CLASS_CORE'$
    }
  }
}
```

B.8.3 UML Class Unit Prototype



```
componentmodel org.reuseware.lib.systems.dataclass.cm.uml
implements org.reuseware.lib.systems.dataclass.dataclass
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
  fragment role Core {
    port type Rec {
      uml::Package.packagedElement is hook if $self.eContainer().oclIsTypeOf(Model)$ {}
    }
  }
  fragment role Class {
    port type Contrib {
      uml::Class is prototype {}
    }
    port type Self {
      uml::Class is anchor {}
    }
    port type Name {
      uml::Class.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.8.4 Java Unit Prototype



```
public class Class {
    private String id;

    public void setID(String id) {
        this.id = id;
    }
    public String getID() {
        return id;
    }

    public String toString() {
        return id;
    }
}
```

```
componentmodel org.reuseware.lib.systems.dataclass.cm.java
implements org.reuseware.lib.systems.dataclass.dataclass
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit, java::containers::EmptyModel {
  fragment role Core {
    port type Rec {
      java::containers::EmptyModel is hook {}
    }
  }
  fragment role Class {
    port type Contrib {
      java::containers::CompilationUnit is prototype {}
    }
    port type Self {
      java::classifiers::Class is anchor {}
      java::classifiers::Class.members is hook {}
    }
    port type Name {
      java::classifiers::Class.name is value hook if $eContainer().oclIsTypeOf(java::
                                                       containers::CompilationUnit)$ {
        point = $'name'$
      }
    }
  }
}
```

B.9 Concern Dimension Associate

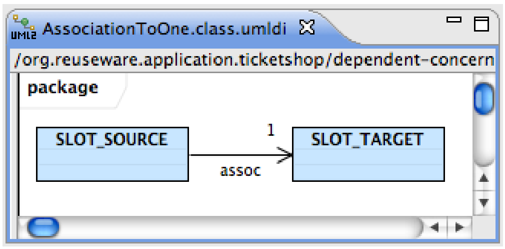
B.9.1 FraCol Definition

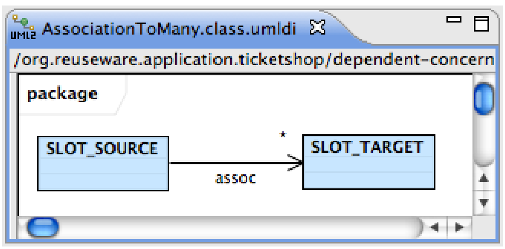
```
fracol org.reuseware.lib.systems.associate.associate {
  fragment role Core {
    static port type Rec;
  }
  fragment role Associated {
    static port type Self;
  }
  fragment role Association {
    static port type Contrib;
    static port type Source;
    static port type Target;
    static port type Name;
    static port type NameLowerCase;
  }
  contributing association Contribution {
    Association.Contrib --> Core.Rec
  }
  configuring association AssociationSource {
    Association.Source --> Associated.Self
  }
  configuring association AssociationTarget {
    Association.Target --> Associated.Self
  }
}
```

B.9.2 UML Use Case with Invariant REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.associate.cl.usecase_uml
implements org.reuseware.lib.systems.associate.associate
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  fragment role Association {
    uml::Comment if $not body.contains('inSet')$ {
      fragment = $'Association:'.concat(ufi.trimExtension().segment(-1)).concat('2').concat(body.split(
                  ' ')->at(1))$
      ufi = $Sequence{'org','reuseware','lib','systems','associate','lib','AssociationToOne.'.concat(
                  variant)}$
      port type Name {  $'name'$ = $body.split(' ')->at(1)$ }
      port type NameLowerCase { $'name'$ = $body.split(' ')->at(1).toLowerCase()$ }
    }
  }
  fragment role Association {
    uml::Comment if $body.contains('inSet')$ {
      fragment = $'Association:'.concat(ufi.trimExtension().segment(-1)).concat('2').concat(body.split(
                  ' ')->at(1))$
      ufi = $Sequence{'org','reuseware','lib','systems','associate','lib','AssociationToMany.'.concat(
                  variant)}$
      port type Name {  $'name'$ = $body.split(' ')->at(1)$ }
      port type NameLowerCase { $'name'$ = $body.split(' ')->at(1).toLowerCase()$ }
    }
  }
  association Contribution {
    uml::Comment {
      fragment = $'Association:'.concat(ufi.trimExtension().segment(-1)).concat('2').concat(body.split(
                  ' ')->at(1))$
      --> fragment = $'CLASS_CORE'$
    }
  }
  association AssociationSource {
    uml::Comment {
      fragment = $'Association:'.concat(ufi.trimExtension().segment(-1)).concat('2').concat(body.split(
                  ' ')->at(1))$
      --> fragment = $'Class:'.concat(ufi.trimExtension().segment(-1))$
    }
  }
  association AssociationTarget {
    uml::Comment {
      fragment = $'Association:'.concat(ufi.trimExtension().segment(-1)).concat('2').concat(body.split(
                  ' ')->at(1))$
      --> fragment = $'Class:'.concat(body.split(' ')->at(1))$
    }
  }
}
```

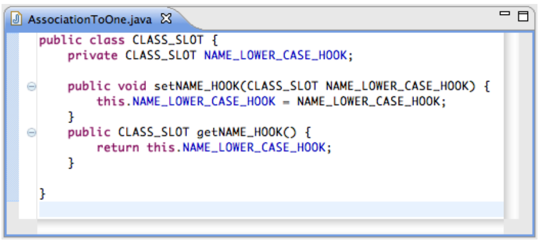
B.9.3 UML Class Unit Prototype


AssociationToOne.class.umldi
/org.reuseware.application.ticketshop/dependent-concern
package
SLOT_SOURCE
1
assoc
SLOT_TARGET

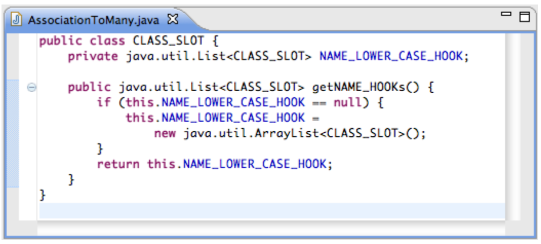

AssociationToMany.class.umldi
/org.reuseware.application.ticketshop/dependent-concern
package
SLOT_SOURCE
*
assoc
SLOT_TARGET

```
componentmodel org.reuseware.lib.systems.associate.cm.uml
implements org.reuseware.lib.systems.associate.associate
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model {
  fragment role Core {
    port type Rec {
      uml::Package.packagedElement is hook if $self.eContainer().oclIsTypeOf(uml::Model)$ {}
    }
  }
  fragment role Associated {
    port type Self {
      uml::Class is anchor {}
    }
  }
  fragment role Association {
    port type Contrib {
      uml::Association is prototype {}
    }
    port type Source {
      uml::Class is slot if $name = 'SLOT_SOURCE'$ {}
    }
    port type Target {
      uml::Class is slot if $name = 'SLOT_TARGET'$ {}
    }
    port type Name {
      uml::Association.name is value hook {
        point = $'name'$
      }
    }
  }
}
```

B.9.4 Java Unit Prototype



```
public class CLASS_SLOT {
    private CLASS_SLOT NAME_LOWER_CASE_HOOK;

    public void setNAME_HOOK(CLASS_SLOT NAME_LOWER_CASE_HOOK) {
        this.NAME_LOWER_CASE_HOOK = NAME_LOWER_CASE_HOOK;
    }
    public CLASS_SLOT getNAME_HOOK() {
        return this.NAME_LOWER_CASE_HOOK;
    }

}
```



```
public class CLASS_SLOT {
    private java.util.List<CLASS_SLOT> NAME_LOWER_CASE_HOOK;

    public java.util.List<CLASS_SLOT> getNAME_HOOKs() {
        if (this.NAME_LOWER_CASE_HOOK == null) {
            this.NAME_LOWER_CASE_HOOK =
                new java.util.ArrayList<CLASS_SLOT>();
        }
        return this.NAME_LOWER_CASE_HOOK;
    }
}
```

```
componentmodel org.reuseware.lib.systems.associate.cm.java
implements org.reuseware.lib.systems.associate.associate
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit, java::containers::EmptyModel {
  fragment role Core {
    port type Rec {
      java::containers::EmptyModel is hook {}
    }
  }
  fragment role Associated {
    port type Self {
      java::classifiers::Class is anchor {}
      java::classifiers::Class.members is hook {}
    }
  }
  fragment role Association {
    port type Contrib {}
    homo port type Name {
      java::commons::NamedElement.name is value hook if $name.contains('NAME_HOOK')${
        point = $'name'$
        begin idx = $name.indexOf('NAME_HOOK')$
        end   idx = $name.indexOf('NAME_HOOK') + 'NAME_HOOK'.length() - 1$
      }
    }
    homo port type NameLowerCase {
      java::commons::NamedElement.name is value hook if $name.contains('NAME_LOWER_CASE_HOOK')${
        point = $'name'$
        begin idx = $name.indexOf('NAME_LOWER_CASE_HOOK')$
        end   idx = $name.indexOf('NAME_LOWER_CASE_HOOK') + 'NAME_LOWER_CASE_HOOK'.length() - 1$
      }
    }
    port type Source {
      java::classifiers::Class.members is prototype {}
    }
    port type Target {
      java::classifiers::Class is slot if $name.contains('CLASS_SLOT')$ {}
    }
  }
}
```

B.10 Concern Dimension Typebinding

B.10.1 FraCol Definition

```
fracol org.reuseware.lib.systems.typebinding.typebinding {
  fragment role GenericEntity {
    static port type TypeParameter;
  }
  fragment role Type {
    static port type Self;
  }
  configuring association Binding {
    Type.Self --> GenericEntity.TypeParameter
  }
}
```

B.10.2 OpenOffice REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.typebinding.cl.odt
implements org.reuseware.lib.systems.typebinding.typebinding
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  association Binding {
    odftext::SpanType if $styleName = 'Actor'$ {
      fragment = $'Class:'.concat(mixed->at(1).getValue().oclAsType(String))$
      --> fragment = $'Participant:'.concat(mixed->at(1).getValue().oclAsType(String)).concat(
                     '_').concat(ufi.trimExtension().segment(-1))$
    }
  }
}
```

B.10.3 UML Use Case with Invariant REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.typebinding.cl.usecase_uml
implements org.reuseware.lib.systems.typebinding.typebinding
epackages <http://www.eclipse.org/uml2/3.0.0/UML>
rootclass uml::Model
ucpi = $ufi.replace('fragments','integrated').replace('usecase.uml',variant).trim(2).append('Main.fc')$ {
  association Binding {
    uml::Actor {
      fragment = $'Class:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'Participant:'.concat(ufi.trimExtension().segment(-1)).concat(
                    '_').concat(ufi.segment(-2))$
    }
  }
  association Binding {
    uml::Comment {
      fragment = $'Class:'.concat(body.split(' ')->at(1))$
      --> fragment = $'Value:'.concat(body.split(' ')->at(1)).concat('_').concat(body.split(' ')->at(2))$
    }
  }
}
```

B.10.4 Java Unit Prototype

```
componentmodel org.reuseware.lib.systems.typebinding.cm.java
implements org.reuseware.lib.systems.typebinding.typebinding
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role GenericEntity {
    port type TypeParameter {
      java::classifiers::Class is slot if $name = 'TYPE_SLOT'$ {}
    }
  }
  fragment role Type {
    port type Self {
      java::classifiers::Class is anchor {}
    }
  }
}
```

B.11 Concern Dimension App

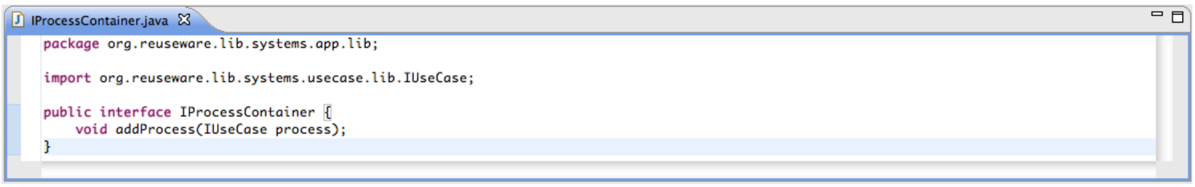
B.11.1 FraCol Definition

```
fracol org.reuseware.lib.systems.app.app {
  fragment role Main {
    static port type Rec;
    static port type Name;
  }
  fragment role Execution {
    static port type Contrib;
    static port type Executable;
  }
  fragment role Executable {
    static port type Self;
  }
  contributing association Execute {
    Execution.Contrib --> Main.Rec
  }
  configuring association AddExecutable {
    Execution.Executable --> Executable.Self }}
```

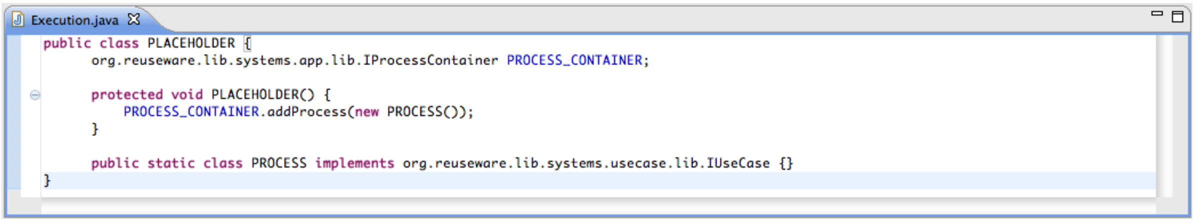
B.11.2 OpenOffice REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.app.cl.odt
implements org.reuseware.lib.systems.app.app
epackages <urn:oasis:names:tc:opendocument:xmlns:office:1.0>
          <urn:oasis:names:tc:opendocument:xmlns:text:1.0>
rootclass odfoffice::DocumentRoot
ucpi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append('Main.fc')$ {
  fragment role Main {
    odfoffice::DocumentRoot {
      fragment = $'APP_CORE'$
      ufi = $Sequence{'org','reuseware','lib','systems','app','lib','Main.'.concat(variant)}$
      target ufi = $ufi.replace('fragments','integrated').replace('odt',variant).trim(1).append(
                    ufi.segment(-4).substring(1,1).toUpperCase().concat(ufi.segment(-4).substring(
                    2,ufi.segment(-4).length()))).appendExtension(variant)$
      port type Name {
        $'name'$ = $ufi.segment(-4).substring(1,1).toUpperCase().concat(ufi.segment(-4).substring(
                    2,ufi.segment(-4).length()))$
      }
    }
  }
  fragment role Execution {
    odfoffice::DocumentRoot {
      fragment = $'Execution:'.concat(ufi.trimExtension().segment(-1))$
      ufi = $Sequence{'org','reuseware','lib','systems','app','lib','Execution.'.concat(variant)}$
    }
  }
  association Execute {
    odfoffice::DocumentRoot {
      fragment = $'Execution:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'APP_CORE'$
    }
  }
  association AddExecutable {
    odfoffice::DocumentRoot {
      fragment = $'Execution:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'UseCase:'.concat(ufi.trimExtension().segment(-1))$
    }
  }
}
```

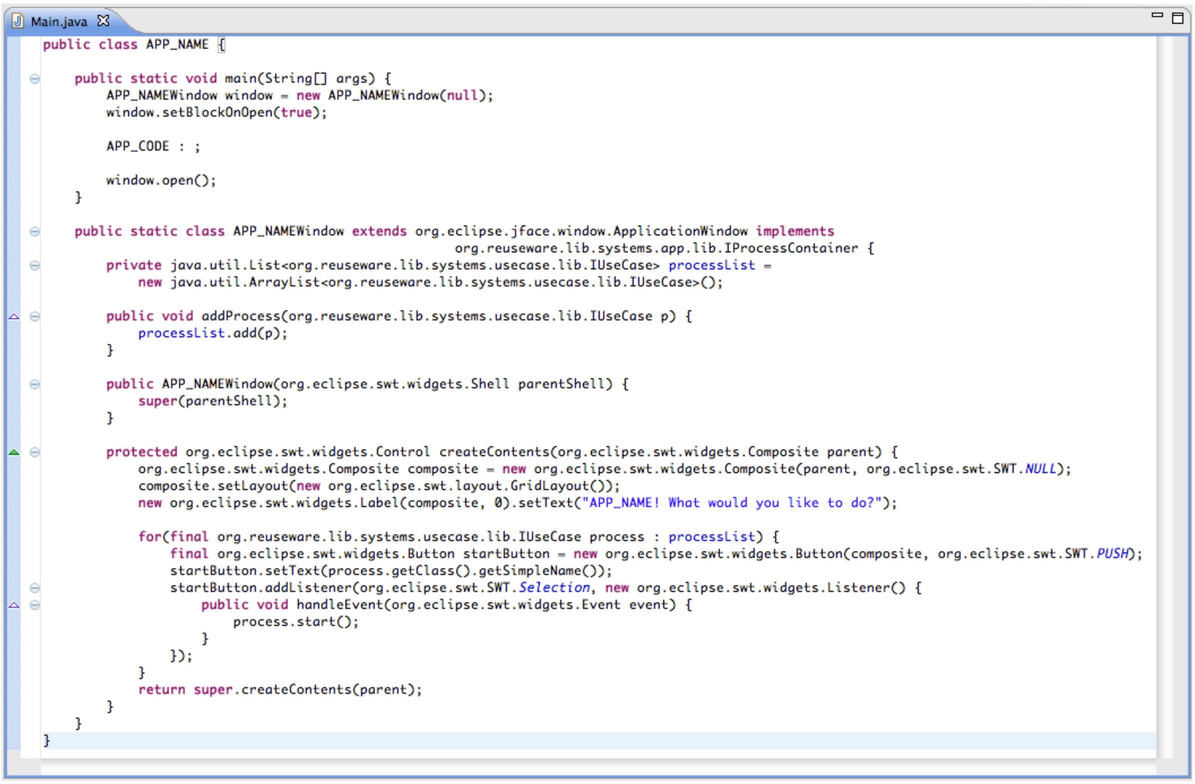
B.11.3 Java Unit Prototype


IProcessContainer.java

```
package org.reuseware.lib.systems.app.lib;

import org.reuseware.lib.systems.usecase.lib.IUseCase;

public interface IProcessContainer {
    void addProcess(IUseCase process);
}
```


Execution.java

```
public class PLACEHOLDER {
	org.reuseware.lib.systems.app.lib.IProcessContainer PROCESS_CONTAINER;

	protected void PLACEHOLDER() {
		PROCESS_CONTAINER.addProcess(new PROCESS());
	}

	public static class PROCESS implements org.reuseware.lib.systems.usecase.lib.IUseCase {}
}
```



Main.java

```
public class APP_NAME {

    public static void main(String[] args) {
        APP_NAMEWindow window = new APP_NAMEWindow(null);
        window.setBlockOnOpen(true);

        APP_CODE : ;

        window.open();
    }

    public static class APP_NAMEWindow extends org.eclipse.jface.window.ApplicationWindow implements
                                                org.reuseware.lib.systems.app.lib.IProcessContainer {
        private java.util.List<org.reuseware.lib.systems.usecase.lib.IUseCase> processList =
            new java.util.ArrayList<org.reuseware.lib.systems.usecase.lib.IUseCase>();

        public void addProcess(org.reuseware.lib.systems.usecase.lib.IUseCase p) {
            processList.add(p);
        }

        public APP_NAMEWindow(org.eclipse.swt.widgets.Shell parentShell) {
            super(parentShell);
        }

        protected org.eclipse.swt.widgets.Control createContents(org.eclipse.swt.widgets.Composite parent) {
            org.eclipse.swt.widgets.Composite composite = new org.eclipse.swt.widgets.Composite(parent, org.eclipse.swt.SWT.NULL);
            composite.setLayout(new org.eclipse.swt.layout.GridLayout());
            new org.eclipse.swt.widgets.Label(composite, 0).setText("APP_NAME! What would you like to do?");

            for(final org.reuseware.lib.systems.usecase.lib.IUseCase process : processList) {
                final org.eclipse.swt.widgets.Button startButton = new org.eclipse.swt.widgets.Button(composite, org.eclipse.swt.SWT.PUSH);
                startButton.setText(process.getClass().getSimpleName());
                startButton.addListener(org.eclipse.swt.SWT.Selection, new org.eclipse.swt.widgets.Listener() {
                    public void handleEvent(org.eclipse.swt.widgets.Event event) {
                        process.start();
                    }
                });
            }
            return super.createContents(parent);
        }
    }
}
```

```
componentmodel org.reuseware.lib.systems.app.cm.java
implements org.reuseware.lib.systems.app.app
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role Main {
    port type Rec {
      java::statements::JumpLabel is hook if $name = 'APP_CODE'$ {
        remove = $true$
      }
      java::variables::LocalVariable is anchor if $typeReference.getTarget().oclAsType(
                                                java::classifiers::Class).name = 'APP_NAMEWindow'$ {}
    }
    homo port type Name {
      java::commons::NamedElement.name is value hook if $name.contains('APP_NAME')${
        point = $'name'$
        begin idx = $name.indexOf('APP_NAME')$
        end   idx = $name.indexOf('APP_NAME') + 'APP_NAME'.length() - 1$
      }
      java::references::StringReference._value is value hook if $value.contains('APP_NAME')${
        point = $'name'$
        begin idx = $value.indexOf('APP_NAME')$
        end   idx = $value.indexOf('APP_NAME') + 'APP_NAME'.length() - 1$
      }
    }
  }
  fragment role Execution {
    port type Contrib {
      java::members::ClassMethod.statements is prototype {}
      java::members::Field is slot if $name.startsWith('PROCESS_CONTAINER')$ {}
    }
    port type Executable {
      java::classifiers::Class is slot if $name = 'PROCESS'$ {}
    }
  }
  fragment role Executable {
    port type Self {
      java::classifiers::Class is anchor if $eContainer().oclIsTypeOf(java::containers::CompilationUnit)$ {}
    }
  }
}
```

B.12 Concern Dimension Security

B.12.1 FraCol Definition

```
fracol org.reuseware.lib.systems.security.security {
  fragment role SecurityInformation {
    static port type Contrib;
    static port type ElementSlot;
  }
  fragment role Container {
    static port type Rec;
  }
  fragment role SecureElement {
    static port type Self;
  }
  contributing association Contributution {
    SecurityInformation.Contrib --> Container.Rec
  }
  configuring association Secure {
    SecurityInformation.ElementSlot --> SecureElement.Self
  }
}
```

B.12.2 SecProp REX$_{CL}$ Specification

```
compositionlanguage org.reuseware.lib.systems.security.cl.secprop
implements org.reuseware.lib.systems.security.security
epackages <http://www.emftext.org/language/SecProp>
rootclass secprop::SecPropModel
ucpi = $ufi.replace('fragments','integrated').replace('secprop',variant).trim(2).append('Main.fc')$ {
  fragment role SecurityInformation {
    secprop::Data {
      fragment = $'SecurityInformation:'.concat(ufi.trimExtension().segment(-1))$
      ufi = $ufi.replace('secprop',variant).trimExtension().appendExtension(variant)$
    }
  }
  association Secure {
    secprop::Data {
      fragment = $'SecurityInformation:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'Value:'.concat(ufi.trimExtension().segment(-1))$
    }
  }
  association Contributution {
    secprop::Data {
      fragment = $'SecurityInformation:'.concat(ufi.trimExtension().segment(-1))$
      --> fragment = $'UseCase:'.concat(ufi.segment(-2))$
    }
  }
}
```

B.12.3 SecProp Unit Prototype

```
componentmodel org.reuseware.lib.systems.security.cm.secprop
implements org.reuseware.lib.systems.security.security
epackages <http://www.emftext.org/language/SecProp>
rootclass secprop::SecPropModel {
  fragment role SecurityInformation {
    port type Contrib {
      secprop::Data is prototype {}
    }
    port type ElementSlot {
      secprop::Data.channel is slot {}
    }
  }
  fragment role SecureElement {
    port type Self {
      secprop::Channel is anchor {}
    }
  }
  fragment role Container {
    port type Rec {
      secprop::SecPropModel.data is hook {}
    }
  }
}
```

B.12.4 Java Unit Prototype

```
componentmodel org.reuseware.lib.systems.security.cm.java
implements org.reuseware.lib.systems.security.security
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {
  fragment role SecurityInformation {
    port type ElementSlot {
      java::members::ClassMethod.statements is prototype if $name = 'SECURITY_BEFORE_SEND'$ {
        point = $'before'$
      }
      java::members::ClassMethod.statements is prototype if $name = 'SECURITY_AFTER_RECEIVE'$ {
        point = $'after'$
      }
      java::members::Field is slot if $name = 'VALUE_SLOT'$ {}
    }
    port type Contrib {}
  }
  fragment role SecureElement {
    port type Self {
      java::statements::LocalVariableStatement is hook if $variable.name = 'ID_SLOT'$ {
        mode = $'append'$
        point = $'before'$
      }
      java::statements::ExpressionStatement is hook if
          $expression.oclAsType(java::references::IdentifierReference).target.name = 'consumer'$ {
        mode = $'prepend'$
        point = $'after'$
      }
      java::variables::LocalVariable is anchor if $name = 'ID_SLOT'$ {}
    }
  }
  fragment role Container {
    port type Rec {}
  }
}
```

C Glossary

architectural style for MDSD a way of organising automatic refinement in a *model-driven architecture.*

CB-MDSD short for *component-based model-driven software development*

component-based model-driven architecture a *model-driven architecture* that relies on *CB-MDSD* concepts.

component-based model-driven software development a way of developing software that combines properties of component-based and model-driven software development.

component language a language in which components are written.

component model the part of a *composition system* that defines what a component is that can be processed by the systems.

composition interface an interfaces to access a fragment for composition.

composition language a language for to write *composition programs.*

composition link a link between two *port instances* in a *composition program* defined in *UCL.*

composition program defines a system as a composition of components.

composition system a system that provides an infrastructure for defining and composing components.

composition technique The part of a *composition system* that defines how components are composed.

DSML short for *domain-specific modelling language.*

domain-specific modelling language a language defined in *EMOF.*

EMOF OMG standard for defining *domain-specific modelling language.*

FraCol short for *fragment collaboration.*

fragment short for *graph fragment*

fragment collaboration a contract about properties fragments have to provide between a component language and a composition language.

fragment role defines properties for one fragment in a *fragment collaboration*.

fragment instance a *fragment* that is reused in a *composition program* defined in *UCL*.

generic composition system a *composition system* with configurable parts.

graph fragment a *model* that is treated as component in *U-ISC/Graph*.

invasive software composition system a composition system that uses physical merging of components as composition technique.

metamodel a language definition in *EMOF*.

model an artefact defined in a *DSML*.

model-driven architecture a software architecture that supports domain abstraction with DSMLs and automated refinement.

ModelHiC an *architectural style for MDSD* based on hierarchical composition.

ModelSoC an *architectural style for MDSD* based on mutli-dimensional separation of concerns.

port one distinguishable point on the *composition interface* of a fragment.

port instance a *port* in the context of a *fragment instance*.

port type part of a *fragment role*.

$\textbf{REX}_{\textbf{CL}}$ a specification language used to integrated a *DSML* into a *composition system* as *composition language*.

$\textbf{REX}_{\textbf{CM}}$ a specification language used to integrated a *DSML* into a *composition system* as *component language*.

U-ISC/Graph short for *universal invasive software composition for graph fragments*

UCL short for *universal composition languages*.

universal composition language the composition language of *U-ISC/Graph*.

universal invasive software composition for graph fragments the *model* composition technique defined in this thesis.

Bibliography

[ABB+01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wust, and Jorg Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, November 2001.

[ABR06] Sven Apel, Don Batory, and Marko Rosenmüller. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations. In *Proceedings of the 1st Workshop on Aspect-Oriented Product Line Engineering (AOPLE'06)*, volume COMP-004-2007, pages 20–24. Lancaster University, October 2006.

[AEB03] Omar Aldawud, Tzilla Elrad, and Atef Bader. UML Profile for Aspect-Oriented Software Development. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling (AOM@AOSD'03)*. `http://www.aspect-modeling.org`, March 2003.

[AOM10] AOM Workshop Organisers. Workshop on Aspect-Oriented Modeling (AOM) Series. `http://www.aspect-modeling.org`, 2010.

[AS08] Colin Atkinson and Dietmar Stoll. Orthographic Modeling Environment. In *Proceedings of Fundamental Approaches to Software Engineering 2010 (FASE'10)*, volume 4961 of *LNCS*, pages 93–96. Springer, March 2008.

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer, April 2003.

[AWB+94] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions using Composition Filters. In *Proceedings of Object-Based Distributed Programming Workshop at ECOOP'93*, volume 791 of *LNCS*, pages 152–184. Springer, April 1994.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing Crosscutting Concerns using Composition Filters. In *Communications of the ACM*, volume 44 (10), pages 51–57. ACM, October 2001.

[Bax92] Ira D. Baxter. Design Maintenance Systems. In *Communications of the ACM*, volume 35(4), pages 73–89. ACM, April 1992.

[BEE+07] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-Based Query Answering. In *Reasoning Web – Third International Summer School 2007*, volume 4636 of *LNCS*, pages 1–153. Springer, August 2007.

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In *Proceedings of the European MDA Workshops: Foundations and Applications (MDAFA'03 and MDAFA'04)*, volume 3599 of *LNCS*, pages 33–46. Springer, August 2005.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*. `http://www.softmetaware.com/oopsla2004/mdsd-workshop.html`, October 2004.

[Bor10] Borland. Borland Together. `http://www.borland.com/us/products/together`, July 2010.

[CB05] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, April 2005.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*. `http://www.softmetaware.com/oopsla2003/mda-workshop.html`, October 2003.

[Cho56] Noam Chomsky. Three Models for the Description of Language. In *IRE Transactions on Information Theory*, volume 2 (3), pages 113–124. IEEE, September 1956.

[CvdBE07] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *Proceedings of 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, pages 85–96. ACM, March 2007.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121. ACM, April 1975.

[DMT10] DMTF – Distributed Management Task Force Inc. Common Information Model Standards. `http://www.dmtf.org/standards/cim`, January 2010.

[Ecl06] Eclipse Foundation. Eclipse Platform Technical Overview. `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`, April 2006.

[Ecl10a] Eclipse Foundation. ATLAS Transformation Language. `http://www.eclipse.org/m2m/atl`, July 2010.

[Ecl10b] Eclipse Foundation. Eclipse Model To Model (M2M) project. `http://www.eclipse.org/m2m`, July 2010.

[Ecl10c] Eclipse Foundation. EMF-based implementation of OCL2. `http://www.eclipse.org/modeling/mdt/?project=ocl`, July 2010.

[Ecl10d] Eclipse Foundation. EMF-based implementation of UML2 metamodel. `http://www.eclipse.org/modeling/mdt/?project=uml2`, July 2010.

[Ecl10e] Eclipse Foundation. Model to Text (M2T) project. `http://www.eclipse.org/modeling/m2t`, July 2010.

[Ecl10f] Eclipse Foundation. MOFScript. `http://www.eclipse.org/gmt/mofscript`, July 2010.

[EFM09] Andy Evans, Miguel A. Fernández, and Parastoo Mohagheghi. Experiences of Developing a Network Modeling Tool Using the Eclipse Environment. In *Proceedings of 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 301–312. Springer, June 2009.

[Fab07] Marcos Didonet Del Fabro. *Metadata Management using Model Weaving and Model Transformation.* PhD thesis, Université de Nantes, September 2007.

[FBFG08] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A Generic Approach For Automatic Model Composition. In *Workshops and Symposia at 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 5002 of *LNCS*, pages 7–15. Springer, June 2008.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, October 2005.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, RIACS, May 2000.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 296–309. Springer, February 2000.

[Gau10] Karsten Gaul. Management of Layout Information in Model-Driven Software Development Processes. Diploma thesis, Technische Universität Dresden, April 2010.

[GBD07] Leif Geige, Thomas Buchmann, and Alexander Dotor. EMF Code Generation with Fujaba. In *Proceedings of the 5th International Fujaba Days.* University of Kassel, October 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software.* Addison-Wesley Professional, November 1994.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Pearson Education, April 2009.

[Hen09] Jakob Henriksson. *A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web.* PhD thesis, Technische Universität Dresden, January 2009.

[Her02] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of International Conference NetObjectDays (NODe'02)*, volume 2591 of *LNCS*, pages 248–264. Springer, October 2002.

[HJK$^+$09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 114–129. Springer, June 2009.

[HJSW10] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the Gap between Modelling and Java. In *Proceedings of 2nd International Conference on Software Language Engineering (SLE'09)*, volume 5969 of *LNCS*, pages 374–383. Springer, March 2010.

[HKA10] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe Composition of Transformations. In *Proceedings of Theory and Practice of Model Transformations: Third International Conference (ICMT'10)*, volume 6142 of *LNCS*, pages 108–122. Springer, June 2010.

[HSVB+10] Aram Hovsepyan, Riccardo Scandariato, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. From Aspect-Oriented Models to Aspect-Oriented Code?: The Maintenance Perspective. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*, pages 85–96. ACM, March 2010.

[HT06] Brent Hailpern and Peri Tarr. Model-Driven Development: The Good, the Bad, and the Ugly. In *IBM Systems Journal*, volume 45(3), pages 451–461. IBM, July 2006.

[IBM10a] IBM. Rational Software Architect. `http://www.ibm.com/software/awdtools/architect/swarchitect`, July 2010.

[IBM10b] IBM. Rational Software Modeler. `http://www.ibm.com/software/awdtools/modeler/swmodeler`, July 2010.

[ikv10] ikv++. medini QVT. `http://projects.ikv.de/qvt`, July 2010.

[JG09] Jendrik Johannes and Karsten Gaul. Towards a Generic Layout Composition Framework for Domain Specific Models. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 113–118. HSE Print, October 2009.

[JN04] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley, December 2004.

[JSS09] Jendrik Johannes, Roland Samlaus, and Mirko Seifert. Round-trip Support for Invasive Software Composition Systems. In *Proceedings of International Conference on Software Composition 2009 (SC'09)*, volume 5634 of *LNCS*, pages 90–106. Springer, June 2009.

[JZF+09] Jendrik Johannes, Steffen Zschaler, Miguel A. Fernández, Antonio Castillo, Dimitrios S. Kolovos, and Richard F. Paige. Abstracting Complex Languages through Transformation and Composition. In *Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 546–550. Springer, October 2009.

[KAAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *Proceedings of 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 87–98. ACM, March 2009.

[KBJV06] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based DSL Frameworks. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages and Applications 2006 (OOPSLA'06)*, pages 602–616. ACM, October 2006.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2071 of *LNCS*, pages 128–142. Springer, June 2001.

[Kle06] Anneke Kleppe. MCC: A Model Transformation Environment. In *Proceedings of 2nd European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'06)*, volume 4066 of *LNCS*, pages 173–187. Springer, June 2006.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, December 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.

[Kol08] Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, June 2008.

[KRPP09] Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Fiona A.C. Polack. Raising the Level of Abstraction in the Development of GMF-based Graphical Model Editors. In *Proceedings of 3rd Workshop on Modeling in Software Engineering (MiSE'09)*, pages 13–19. IEEE, May 2009.

[Mey90] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, July 1990.

[MKBJ08] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A Generic Weaver for Supporting Product Lines. In *Proceedings of the 13th international workshop on Early Aspects (EA'08)*, pages 11–18. ACM, May 2008.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, July 1993.

[MOD08] MODELPLEX Project. Deliverable D1.1.a (v3): Case Study Scenario Definitions. `http://www.modelplex.org`, March 2008.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 742–745. ACM, June 2000.

[NoM10] NoMagic, Inc. MagicDraw. `http://www.magicdraw.com`, July 2010.

[OAS07] OASIS Consortium. Open Document Format for Office Applications (OpenDocument) v1.1. `http://www.oasis-open.org/committees/office`, February 2007.

[OMG01] OMG – Object Management Group. MDA Guide V1.0.1. `http://www.omg.org/mda`, June 2001.

[OMG06a] OMG – Object Management Group. MOF 2.0 Core Specification. `http://www.omg.org/spec/MOF/2.0`, January 2006.

[OMG06b] OMG – Object Management Group. Object Constraint Language 2.0. `http://www.omg.org/spec/OCL/2.0`, May 2006.

[OMG06c] OMG – Object Management Group. UML Diagram Interchange, Version 1.0. `http://www.omg.org/spec/UMLDI/1.0`, April 2006.

[OMG07] OMG – Object Management Group. XML Metadata Interchange (XMI), V2.1.1. `http://www.omg.org/spec/XMI/2.1.1`, December 2007.

[OMG08] OMG – Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.0. `http://www.omg.org/spec/QVT/1.0`, April 2008.

[OMG10a] OMG – Object Management Group. OMG – Object Management Group Website. `http://www.omg.org`, July 2010.

[OMG10b] OMG – Object Management Group. Unified Modeling Language (UML) Version 2.3. `http://www.omg.org/spec/UML/2.3`, May 2010.

[OT00] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[PD91] Rubén Prieto-Díaz. Implementing Faceted Classification for Software Reuse. In *Communications of the ACM*, volume 34(5), pages 88–97. ACM, May 1991.

[PDF87] Rubén Prieto-Díaz and Peter Freeman. Classifying Software for Reusability. In *IEEE Software*, volume 4(1), pages 6–16. IEEE, January 1987.

[PDF$^+$02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A UML Notation for Aspect-Oriented Software Design. In *Proceedings of the 1st Workshop on Aspect-Oriented Modeling with UML*. `http://www.aspect-modeling.org`, March 2002.

[RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *ACM SIGPLAN Notices*, volume 33 (10), pages 117–133. ACM, October 1998.

[RGR$^+$06] Raghu Reddy, Sudipto Ghosh, Robert B. Rance, Greg Straw, James M. Bieman, Eunjee Song, and Geri Georg. Directives for Composing Aspect-Oriented Design Class Models. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 75–105. Springer, February 2006.

[Roy08] Suman Roychoudhury. *GenAWeave: A Generic Aspect Weaver Framework based on Model-Driven Program Transformation*. PhD thesis, University of Alabama at Birmingham, September 2008.

[RW07] Boris Roussev and Jie Wu. Transforming Use Case Models to Class Models and OCL-Specifications. In *International Journal of Computers and Applications*, volume 29 (1), pages 59–69. ACTA Press, January 2007.

[Sav03] Ilie Savga. XML Recoder: A Framework for XML Code Analysis and Transformation. Master thesis, Linkoping University, December 2003.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework, 2nd Edition*. Pearson Education, January 2009.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the 40th International Conference on Tools Pacific (TOOLS Pacific'02)*, pages 53–60. Australian Computer Society, Inc., 2002.

[SHU06] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In *Proceedings of 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 12–26. ACM, March 2006.

[Sof10] Software Engineering group at the University of Twente. Compose* – Composition Filters Applied. `http://composestar.sourceforge.net`, July 2010.

[SPJF10] Matthias Schmidt, Jan Polowinski, Jendrik Johannes, and Miguel A. Fernández. An Integrated Facet-based Library for Arbitrary Software Components. In *In Proceedings of 6th European Conference on Modelling Foundations and Applications (ECMFA'10)*, volume 6138 of *LNCS*, pages 261–276. Springer, June 2010.

[SRK$^+$07] Andrea Schauerhuber, Werner Retschitzegger, Gerti Kappel, Elisabeth Kapsammer, Manuel Wimmer, and Wieland Schwinger. A Survey on Aspect-Oriented Modeling Approaches. Technical report, Vienna University of Technology, October 2007.

[Ste00] Friedrich Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modelling. In *Data & Knowledge Engineering*, volume 35 (1), pages 83–106. Elsevier, October 2000.

[TOP10] TOPCASED Development Team. TOPCASED Environment. `http://www.topcased.org`, July 2010.

[Tri10] Triskell Project Team. Kermeta: Triskell Metamodeling Kernel. `http://www.kermeta.org`, July 2010.

[VAB$^+$07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. UniTI: A Unified Transformation Infrastructure. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, pages 31–45. Springer, September 2007.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. In *ACM SIGPLAN Notices*, volume 35(6), pages 26–36. ACM, June 2000.

[vP07] Jens von Pilgrim. Mental Map and Model Driven Development. In *Proceedings of Workshop on the Layout of (Software) Engineering Diagrams 2007 (LED'07)*, volume 7. EASST, September 2007.

[vPVSGB08] Jens von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and Visualizing Transformation Chains. In *Proceedings of 4th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'08)*, volume 5095 of *LNCS*, pages 17–32. Springer, June 2008.

[W3C04] W3C – World Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. `www.w3.org/TR/rdf-schema`, February 2004.

[WJE$^+$09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and Joao Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *LNCS*, pages 191–237. Springer, October 2009.