Karlsruhe Institute of Technology

Generic and Extensible Model Weaving and its Application to Building Models

Diploma Thesis of

Max E. Kramer

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Yves Le Traon
Second reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	Dr. Jacques Klein
Second advisor:	Dr. Lucia Happe

Duration:: December 16, 2011 – April 25, 2012

Max E. Kramer: *Generic and Extensible Model Weaving and its Application to Building Models*, Diploma Thesis, Version 1.0, April 25, 2012

Zusammenfassung

Viele Verfahren im Rahmen der modellgetriebenen Softwareentwicklung modifizieren existierende Modelle auf eine Weise, welche an die Erfüllung von Bedingungen geknüpft ist und mehrere Modellbereiche betrifft. Diese Verfahren können in unterschiedlicher Gestalt auftreten. Beispielsweise als Restrukturierungen, Modellvervollständigungen, oder als aspektorientierte Modellweberei. Obwohl die zentralen Vorgänge dieser Verfahren unabhängig von der Anwendungsdomäne sind, werden selten generische Lösungen, die einfach benutz- und anpassbar sind, verwendet. Sowohl universelle Modelltransformationssprachen als auch bestehenende Modellweber führen zu domänenspezifischen Einschränkungen und ungewollter Komplexität. In dieser Diplomarbeit widmen wir uns diesen Problemen auf der Metamodellierungsebene mit einem Modellweber, der aufgrund seiner Allgemeinheit erweiterbar und zweckmäßig ist. Dazu führen wir vorangegangene Arbeiten zur Modellweberei zusammen, um einen verbesserten und dennoch vereinfachten Ansatz für beliebige Modelle anzubieten. Der vorgestellte Ansatz und seine prototypische Implementierung können nicht unmittelbar mit maßgeschneiderten Modellwebern verglichen werden, sondern stellen eine erweiterbare, generische Grundlage für domänenspezifische Anpassungen dar.

Wir führen eine erste Validierung der Erweiterungsmöglichkeiten unseres Ansatzes anhand einer Anpassung für Gebäudemodelle durch. Außerdem untersuchen wir den Zusammenhang, in dem diese Modelle auftreten, um Perspektiven für einen vollständigen Anwendungsfall zu bieten. Dieser Anwendungsfall verwendet semantisch reichhaltige, dreidimensionale Gebäudemodelle, die spezifische Detailinformationen enthalten können. Viele dieser möglichen Details werden jedoch nicht in die Gebäudemodelle eingearbeitet, da bestehende Werkzeuge keine Möglichkeiten zum automatischen Einfügen spezifischer Informationen an allen betroffenen Stellen bieten. Daher werden diese Details in einen separaten, natürlichsprachigen Text, die Gebäudespezifikation, ausgelagert. Infolgedessen weisen die Gebäudemodelle nicht alle für eine Analyse notwendigen Informationen auf sodass die Gebäudespezifikationen manuell interpretiert werden müssen, was zeitaufwändig und fehleranfällig sein kann. In dieser Diplomarbeit schlagen wir eine Methode vor, die Informationen aus Gebäudespezifikationen zu Gebäudemodellen mittels einer flexiblen, kontrollierten Sprache hinzufügt. Diese Sprache erzeugt Aspektmodelle, die mit einer Anpassung unseres Modellwebers in Gebäudemodelle eingearbeitet werden können. Mit dieser Anwendung zeigen wir, dass unser generischer Ansatz mit wenig Aufwand selbst an Anwendungsdomänen mit herausragenden Besonderheiten angepasst werden kann.

Abstract

Many tasks in Model-Driven Engineering (MDE) modify existing models in a way that depends on conditions for affected regions and cross-cuts the primary decomposition. These tasks may appear in different forms such as refactorings, model completions or aspect-oriented model weaving. Although the operations at the heart of these tasks are independent of the domain, generic solutions that can easily be used and customised are rare. General-purpose model transformation languages as well as existing model weavers introduce domain-specific restrictions and accidental complexity. In this thesis we address these problems on the metamodelling level with a model weaver that is extensible and practical *because* it is generic. For this generic model weaver we consolidate existing work on model weaving in order to provide an enhanced, yet simplified, approach that can be used for arbitrary models. The resulting weaving approach and its prototype implementation cannot directly be compared with tailor-made model weavers but provide an extensible, generic base for domain-specific customisations.

We conduct a preliminary validation of the extension capabilities of our approach using a customisation for building models. Furthermore, we investigate the application context for these models in order to analyse perspectives for a complete use case. This use case employs semantically rich, three-dimensional building models, which can capture specific information at a high level of detail. Many of these possible details are, however, not included in the models because existing tools do not provide possibilities to automatically add specific information at all affected places. Thus, these details are sourced out into a separate, natural language text called building specification. As a result, the building models lack some of the information needed for analyses and thus the building specifications have to be manually reinterpreted, which may be time-consuming and error-prone. In this thesis we propose a way to add building specification information to building models using a flexible, controlled natural language that produces aspect models. These aspect models can be woven into the building model using a customisation of our presented generic model weaver. With this application to building models we show that little additional work is needed to customise our generic approach for domains that exhibit outstanding specifics.

Acknowledgements

I am very grateful to all researchers that had their share in this thesis. Jacques Klein offered me to write this thesis at the University of Luxembourg and lead me through the fascinating world of model weaving. Yves Le Traon and his serval team provided the academic and personal atmosphere in which this project could grow and prosper. Ralf Reussner gave me the opportunity to freely write an external thesis while safeguarding it at my home university in Karlsruhe. Lucia Happe showed me methods and ways to communicate my results. Many thanks to all of you.

Special thanks go to Jim Steel and Jörg Kienzle, who encouraged and supported me with joint publications of the results of this thesis. Furthermore, I am thankful to Benjamin Niedermann for enduring all unresolved cross-references and blank pages while proofreading this thesis.

The feedback of all these people had a great impact on my work and this thesis.

Declaration

I hereby declare that this thesis and all results presented in it are my original work and have not been submitted in any form to another university or educational instution for any award. Where information was derived from the published or unpublished work of others, this has been acknowledged.

Karlsruhe, April 2012

Max E. Kramer

Contents

Nomenclature

AMW	Atlas Model Weaver
AOM	Aspect-Oriented Modelling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
AST	Abstract Syntax Tree
ATL	ATLAS Transformation Language
BIM	Building Information Modelling
C-SAW	Constraint-Specification Aspect Weaver
CAD	Computer Aided Design
CNL	Controlled Natural Language
DSL	Domain-Specific Language
DSML	Domain-Specific Modelling Language
DSMTL	Domain-Specific Model Transformation Language
ECD	Exectuable Class Diagram
EMF	Eclipse Modeling Framework
EML	Epsilon Merging Language
GME	Generic Modeling Environment
GUI	Graphical User Interface
HOT	Higher-Order Transformation
IDE	Integrated Development Environment
IFC	Industry Foundation Classes
JPDD	Join Point Designation Diagram
JVM	Java Virtual Machine

LTS	Labelled Transition System
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
POJO	Plain Old Java Object
QVT	Query / View / Transformation
QVT-O	QVT-Operational
QVT-R	QVT-Relational
RDL	Requirements Description Language
SBVR	Semantics of Business Vocabulary and Business Rules
SPL	Software Product Lines
STEP	Standard for the Exchange of Product model data
UML	Unified Modeling Language
URI	Unique Resource Identifier

Part I.

Prelude

1. Introduction & Motivation

In Model-Driven Engineering (MDE) various activities require the modification of several areas of a model according to properties of these areas. Such activities may take the shape of refactoring tasks or search-and-replace tasks similar to those supported in text editors of Integrated Development Environments (IDEs). Others appear in more complex settings, such as model-completion transformations or model weaving in aspect-oriented environments. These activities are composed of atomic add, change, and remove operations similar to CRUD[1] operations in database systems. Although these tasks are largely independent of the problem domain, generic solutions that can be easily reused and customised for arbitrary kinds of models are rare. Existing solutions are restricted to certain modelling notations, do not support conditional application of changes, are not available for direct use and improvement, ignore domain-specific properties, or introduce accidental complexity. This thesis presents an extensible approach to generic model weaving that addresses these problems.

An example of such recurring model modifications is encountered when detailed information has to be added to an existing building model. Such a semantically rich, three-dimensional model is capable of capturing specific information at a high level of detail. Many of these details are, however, not included in the model because existing tools do not support the automatic addition of specific information at all places that fulfil the conditions of application. Thus, these details are formulated separately in a natural language text called building specification. The consequence of sourcing out these details into a specification text is that the building model does not contain all the information relevant for cost estimation and other analyses. For this reason the specification text has to be manually reinterpreted whenever these tasks are performed, which may be time-consuming and error-prone. This thesis proposes a way to add building specification information to building models using a flexible, controlled natural language. Sentences of this language can be translated to aspects that can be woven using a customisation of our generic model weaver.

1.1. The Need for Generic Model Weavers

There are different ways to carry out model modifications that affect various model areas at once according to encountered properties. But, to our knowledge, no existing

[1] CRUD - Create, Read, Update, Delete

solution provides a method that works for arbitrary models and exhibits a complexity that can be compared to the complexity of textual search-and-replace approaches.

General-purpose model transformation languages, for example, have not been designed specifically for these conditional modifications. In the terminology of model transformations such a search-and-replace type of modification is characterised by the involvement of a single modelling language (homogenous) and by the fact that a single model instance acts as input and output model at the same time (in-place). This transformation scenario is only one of the various scenarios that are supported by general-purpose model transformation languages. For that reason, these languages usually provide sophisticated functionality for transformations that involve models of different types that play different roles during the transformation. As a result, domain experts that simply want to add details to an existing model have to make efforts to master these powerful transformation languages. They have to reason about technicalities of the transformation that are not central to their task.

Model weaving approaches are another candidate for easing such modification tasks as they provide specific concepts for model changes that cross-cut the system's main decomposition. Currently available model weavers, however, are mostly bound to specific notations and tend to add a layer of complexity to these simple tasks just as general-purpose transformation languages do. This complexity results from the need for detailed weaving instructions, preparatory conversions to a formalism that supports weaving, or incomplete automation (see Chapter 11). Nevertheless, industrial domain-specific applications of model weaving, e.g. for communication infrastructure [CvdBE06] or robustness modelling [ABH11] are promising and suggest that further research is needed to overcome these shortcomings.

In this thesis we present a generic model weaver that tries to address these problems of genericity and complexity using customisable operations that are executed on the metamodel level. The weaver is a result of the consolidation of existing work on model weaving that considered, for example, the detection of areas to be changed (join points) or a declarative definition of aspects using derived modelling language variants. We incorporated these parts into an enhanced, yet simplified, approach to model weaving that can be used for arbitrary models. The goal was not to create a general weaver that can compete with every tailor-made weaver but to provide a generic base that can be reused and customised for arbitrary domains. To foster this reuse we developed an intensively documented, open-source implementation[2] that provides elaborate extension possibilities.

1.2. Integrating Specification Information into Buildings

In huge construction projects it is infeasible to manually construct a model that displays all elements at the highest of all levels of detail needed throughout the project. Nevertheless, this information is required for important analysis tasks, such as cost estimation, and therefore has to be obtained from another source of information. This is in stark contrast to the principle of MDE that regards transformable models conforming to well-defined metamodels as the main source of information during the design of a system. The construction industry cannot always respect these Software Engineering principles and expresses the details omitted in the building models in a natural language text called building specification. Together, the building model and the building specification serve as main input to all tasks requiring detailed information. This poses

[2] code.google.com/a/eclipselabs.org/p/geko-model-weaver

barriers to the automation of processes because the specification text has to be reinterpreted by experts whenever these tasks are performed. An example for the consequences of this lack of automation can be observed during the calculation of the quantities of material and work needed to construct a building. Although the model can be used to automatically retrieve costs for the rough structure, the calculation remains incomplete as longs as the details of the specification text are not manually taken into account. Therefore the process of cost estimation, which is vital to construction projects, is complex, time-consuming and requires a lot of skill and experience. We are confident that this and other analysis tasks could be performed faster and with a higher degree of precision and automation if a significant part of the details expressed in building specifications were directly available in the building model.

In this thesis we classify some of the concerns contained in building specification texts as cross-cutting and propose a way to integrate them into building models. We suggest a controlled natural language that is defined by domain-experts and produces aspect models that can be woven using the generic model weaver presented in this thesis. The application of our generic model weaver to the domain of building models serves as preliminary evaluation of its extension and customisation facilities. It shows that little additional work is needed to customise our generic weaver to this domain with outstanding specifics. In future work, this first result should be verified with new extensions for models of other domains.

1.3. Contributions

This thesis presents a generic, extensible, and practical model weaver, called GeKo, together with a demonstration of its use in different domains. The presented approach is generic because it is defined at the metametamodel level and operates on the metamodel level so that it can be applied to models of arbitrary type. It is extensible as it provides facilities for domain-specific solutions that can be used without modifications of the generic core weaving logic. Finally, it is practical because it provides detailed documentation and an implementation which can be used together with existing modelling languages and tools.

The contributions of this thesis are:

- the consolidation of existing work on model weaving into a generic weaving approach proving that practical generic model weaving can be defined on the metametamodel level (Chapter 3 – 5).
- the illustration of an extension mechanism for this weaver showing that little work is needed to customise the generic approach (Chapter 4).
- the provision of a thoroughly documented, extensible prototype implementation of this generic model weaver (Chapter 6 – 7).
- the classification of cross-cutting building specification concerns as domain-specific aspects for building models (Chapter 8).
- the proposition to combine existing techniques to express these concerns using a controlled natural language defined by domain-experts (Chapter 9).
- the preliminary evaluation of the extensions facilities of our weaving approach and its implementation in the context of building models (Chapter 10).

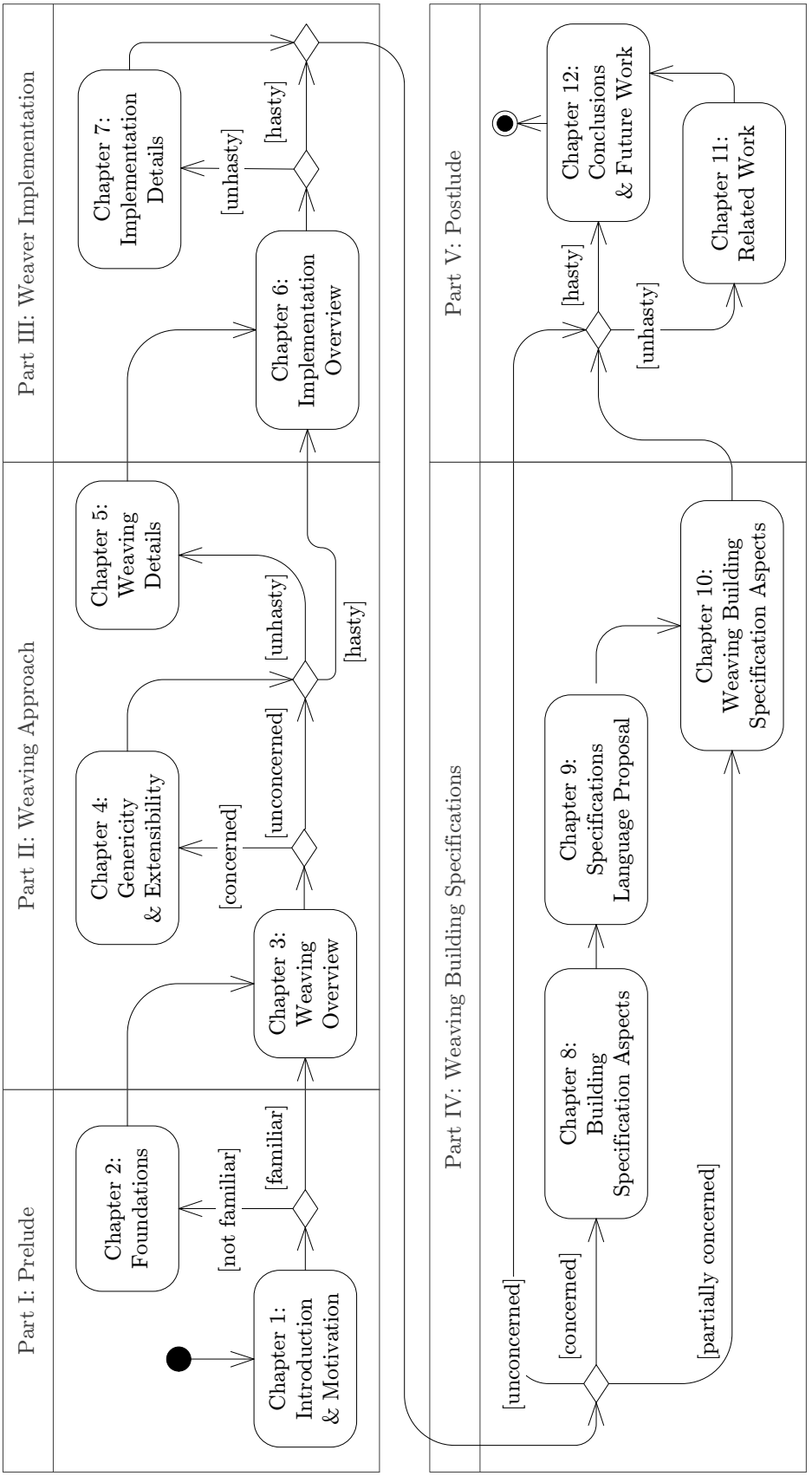


Fig. 1.1.: A UML activity diagram showing different ways to read this thesis.

1.4. Structure

This thesis is structured in three individual parts in order to ease selective reading: In the first part Chapter 2 provides the fundamental background for our work.

Part II presents the concepts and features of our approach to model weaving in an abstract way. Chapter 3 introduces our approach presenting its key characteristics and the overall workflow devised into weaving phases. Chapter 4 explains how we ensured genericity in the sense that the approach may be applied to arbitrary models and shows how generic weaving operations can be customised and completed through domain-specific extensions. Chapter 5 discusses the individual phases of weaving in detail and presents the formal foundations and weaving operations.

In Part III we explain how we implemented a prototype of our weaving approach in an extensible way. Chapter 6 presents the plug-in architecture of our solution together with its environment and library dependencies. Chapter 7 provides insights into specific problems, solutions, algorithms and data structures of our implementation.

Part IV shows how our generic weaving approach can be applied in order to integrate building specification information into building models. In Chapter 8 we show that some of the concerns expressed in building specifications exhibit characteristics of domain aspects. Chapter 9 proposes a solution to capture these concerns using a flexible language that can be used and defined by domain-experts. Chapter 10 illustrates the extension capabilities of our approach by showing how we were able to weave specification aspects into building models using a set of custom extensions.

Part V closes this thesis with two more chapters. Chapter 11 discusses related work on model weaving, domain-specific model transformation languages, and existing approaches to enriching building models with specification information. Chapter 12 draws some final conclusions and outlines possibilities for future work.

Readers of this thesis can choose a different reading path depending on their background, interests, and time schedule. In order to guide the readers through the dozen chapters of this thesis we provide a UML activity diagram depicting possible ways of reading in Fig. 1.1. First, every reader starts with the current introductory and motivating chapter. Then, depending on the knowledge of the domains of model weaving and Building Information Modelling (BIM), the foundations chapter can be skipped in parts or completely. The following overview chapter of our weaving approach is recommended for all readers. Afterwards, readers that are not concerned by the genericity and extensibility of our approach can skip the corresponding chapter. But we recommend that only hasty readers skip the subsequent detailed discussion of our weaving approach. Next, the short chapter introducing the prototype implementation of our weaving approach shall be worth reading for everyone, whereas the subsequent detailed discussion can be skipped by readers that are in a hurry. The following three chapters of Part IV can be skipped entirely by unconcerned readers. Those readers that are interested in an extension to our weaver but not particularly concerned with building specifications can directly move on to the chapter on weaving building specification aspects. Last, we recommend hasty readers to skip the chapter on related work, but we advise everybody to read the conclusions chapter.

2. Foundations

In this chapter we present the concepts that are fundamental to the work presented in this thesis. Starting with the paradigm of Model-Driven Engineering we explain the notions of metamodels, model transformations, and Domain-Specific Languages in Section 2.1. The terms that are specific to Aspect-Oriented Modelling are presented in Section 2.2. How the construction industry deals with models using the umbrella term Building Information Modelling is explained in Section 2.3. Finally, Section 2.4 explains how building models can be processed like software models using a technological bridge.

2.1. Model Driven Engineering

"Everything is a model." [Béz05]

In the field of Software Engineering and other engineering disciplines, Model-Driven Engineering (MDE) is a paradigm that elevates models to first-class entities during all phases of systems design. All engineering artefacts are expressed as models ([Béz05]), so that model-based techniques can be applied to them. This stands in contrast to Model-Based Development, which uses models as support or only during some of the steps of the development process.

As models are used in different ways in various contexts, we want to clarify the notion of models for the reader by providing a definition:

Definition 1 (Model) *A Model partially represents entities and relations of a system using a well-defined formalism in order to abstract details for a certain purpose ([Sta73]).*

This definition includes three fundamental properties of a model, which were identified by Stachowiak in 1973: reproduction, abstraction, and pragmatism [Sta73]. Note that the definition still permits different interpretations depending on the notion of formalism. For example, for verification and validation tasks the formal requirements for models tend to be more restrictive than in other contexts. For our work and our understanding of MDE the basic formal requirements are best expressed by demanding the use of metamodels that we describe next.

2.1.1. Metamodels

One possibility to approach the question of formalism for models are metamodels which express structural requirements for models. A metamodel represents the structural

possibilities for a family of models, for example, by defining what elements can be contained in a model and how they can be related. In addition to this possible kinds of entities and relations, a metamodel may impose arbitrary constraints on the members of a model family.

Metamodels as structural rules

We want to provide a definition of the notion of metamodel that is in line with our definition for models. To achieve this we have to clarify which system is represented by a metamodel, which formalism is used, which details are abstracted, and for which purpose this is done. A metamodel represents the structural system spanned by all possible members of a family of models. Sometimes a metamodel is also said to constitute the abstract-syntax of a family of models as it omits syntactical details and semantic information (cf. [HR04]). The purpose of a metamodel is to have a mean to create correct member instances of a family of models and to determine whether existing or modified models conform to the constraints of a model family. But as a practical definition needs to be more concise in order to provide a benefit to the reader we provide a condenseded definition:

Definition 2 (Metamodel) *A Metamodel represents all structural constraints that instances of it have to satisfy.*

A metamodel is expressed using a meta-modelling language that may in turn be represented as a metametamodel. This metametamodel is usually described with the concepts of the meta-modelling language itself also known as self-representation or self-description. The modelled system, the model, the metamodel, and the metametamodel form a group of four layers known as M0, M1, M2, and M3. These model layers and the relations between them are depicted in Fig. 2.1.

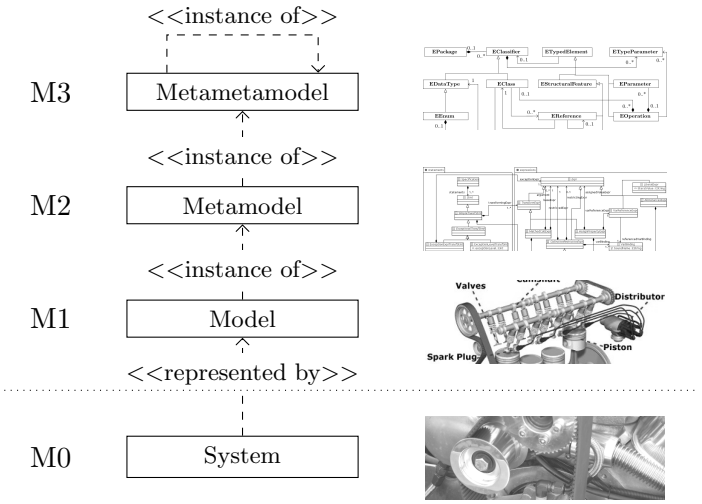


Fig. 2.1.: Layers of modelling in MDE

Just as in our previous discussion for models, the level of formalism for metamodels depends on the purpose. We would assume that the number and strictness of metamodel constraints is, for example, higher for model checking tasks than for code generation. In the next subsection, we present a modeling activity that can handle metamodels with low or high level of detail.

2.1.2. Model Transformations

In an environment in which all design artefacts are models it is essential to modify these models in an automatic way that guarantees structural integrity. In the context of MDE

such types of modifications are usually called model transformations. Arbitrary model modifications differ in two ways from model transformations. First, an arbitrary model modification may be performed or supported by human interaction. Second, it may result in models that exhibit a structure that violates the constraints for these models. Although in our work these constraints are usually expressed using metamodels we provide a more general definition for model transformations:

Automated, struture preserving modifications

Definition 3 (Model Transformation) *A Model Transformation is an automated modification of models that ensures conformance to the structural constraints for these models [SVBvS06].*

The structural integrity conserving property of model transformations relieves their users from the necessity of fixing syntactical errors. Nevertheless, a model transformation may result in undesired models if the metamodel does not contain all required constraints. Furthermore, the structural conservation makes it possible to define automised chains of transformations when all transformation preconditions are embodied in the metamodel.

Model transformations can be categorised depending on the number and types of models that are used as input and output for the transformation. When one or more model instances of a certain metamodel are transformed to one or more instances of a different metamodel we speak of an exogenous transformation. Similarly, in an endogenous model transformation the metamodels of the input models are the same as those of the output models. If an endogenous model transformation has only input models and no output models it is called in-place as the changes are directly applied to the input models. Out-place transformations have at least one model that is only used for output. This is always the case for exogenous transformations. Bidirectional model transformations can be used in both transformation directions as the role of input and output models can be interchanged. A detailed taxonomy of model transformations was created by Mens and Van Gorp [MG06].

During the last decade various frameworks and languages for model transformations were created. The Object Management Group (OMG) defined a standard for model transformations called Query / View / Transformation (QVT). Within this standard three transformation languages were defined: the imperative language QVT-Operational (QVT-O), the declarative language QVT-Relational (QVT-R) supporting bidirectional transformations, and a simpler declarative language QVT-Core, which should serve as translation target for QVT-R. All three languages conform to the OMG metamodel standard Meta Object Facility (MOF) 2.0 and were implemented in diverse tools.

Besides OMG's standard other model transformation languages such as the ATLAS Transformation Language (ATL)[JAB+06] and *Kermeta*[1] are also used in industry and research. ATL was developed by Obeo and the AtlanMod team of INRIA[2] in response to OMG's QVT request for proposal. As a hybrid language it offers imperative as well as declarative constructs. Kermeta was developed by the Triskell team of IRISA[3] as an imperative model transformation language that supports functional and aspect-oriented programming constructs. All presented languages are called general-purpose model transformation languages as they are not bound to a specific purpose or domain.

[1] Kermeta - A MetaModel Engineering language and workbench: kermeta.org
[2] INRIA - National Institute for Research in Computer Science and Control: inria.fr
[3] IRISA - Research Institute in Computer Science and Random Systems: irisa.fr

2.1.3. Domain-Specific Modelling Languages

A Domain-Specific Modelling Language (DSML) combines the conceptual and technical advantages of MDE with the raised abstraction level of domain-specific solutions. The goal of a Domain-Specific Language (DSL) is usually to provide persons that possess domain-specific knowledge the means to perform tasks that would require additional technical knowledge. In MDE the purpose of a DSML is to ease tasks such as design and analysis for domain-experts while ensuring automation capabilities and other benefits of MDE. Using DSMLs domain experts should be given the ability to produce models that can be transformed and refined using MDE tools. In contrast to general-purpose modelling languages such as UML a DSML is customized to the domain, so that common concepts can be expressed tersely and precisely.

A DSML is more than a domain metamodel

To which extend a metamodel can be seen as a DSML is disputable. According to our Definition 2, a DSML that is represented using a metamodel does not necessarily contain all information that would be necessary to describe its concrete syntax. Nevertheless, a metamodel may serve as description of the abstract syntax of a DSML. If the used meta-modelling language provides a standard representation for metamodel instances we may even have sufficient information to speak of a language. But as this language would express domain-specific entities using a general-purpose representation it could be misleading to speak of a DSML. Therefore, we think that the term DSML should only be used if the concrete syntax is also customised to the needs of the domain. This distinction is particularily important as one of the main advantages of DSLs is the ability to narrow the scope and exclude concepts and statements that are part of a general purpose language.

Definition 4 (Domain-Specific Modelling Language)
A Domain-Specific Modelling Language provides a domain-specific concrete syntax to represent models of a particular domain [CSN05].

An advantage of DSMLs is that they may ease the creation of models that are sufficiently detailed to be the only source of information and do not need to be further refined. Within the Software Engineering domain this advantage is better known as the power to enable full code generation. Kelly and Tolvanen [KT08] discuss this claim and present collected experiences with DSMLs for various software domains. A similar possible quality can be observed in the context of building models and its DSML as described in Chapter 9.

A special case of DSML are Domain-Specific Model Transformation Languages (DSMTLs), which combine the advantage of customisation for a domain with the structure preserving property of model transformations. To our knowledge DSMTLs have not been deeply investigated but this may also be due to the fact that not all of the domain-specific languages for model modifications that ensure structural properties are presented using the term DSMTL. There is, however, a generator framework for DSMTLs by Reiter et al. [RKR+06] that facilitates the creation of such languages using a template-based approach that targets general purpose model transformation languages. In Chapter 9 we present our first ideas for a DSMTL for Building Information Modelling (see Section 2.3). Other examples of DSMTLs such as a language for configuring features of a product line[AGGR07] are presented in Chapter 11.

2.1.4. Eclipse Modeling Framework

A MDE infrastructure for Eclipse

The Eclipse Modeling Framework (EMF)[4] is a set of plug-ins for Eclipse[5], a popular Integrated Development Environment (IDE). It consists of several subprojects that provide tools to define the abstract and concrete syntax of modelling languages. As well as its hosting workbench, EMF is platform-independent and provides various libraries that ease standard implementation work.

At the heart of EMF is its meta-meta-modelling language *Ecore*, which matches closely OMG's standard EMOF 2.0. Ecore supports generically typed metaclasses that can contain simple-typed attributes, complex-typed references, and operations. In Fig. 2.2 we present central entities and relations of Ecore in a simplified model. It shows that metamodels that are defined using Ecore are organised in packages that contain metaclasses and data types such as enumerations. The typification of attributes, references, operations and their parameters is also visible.

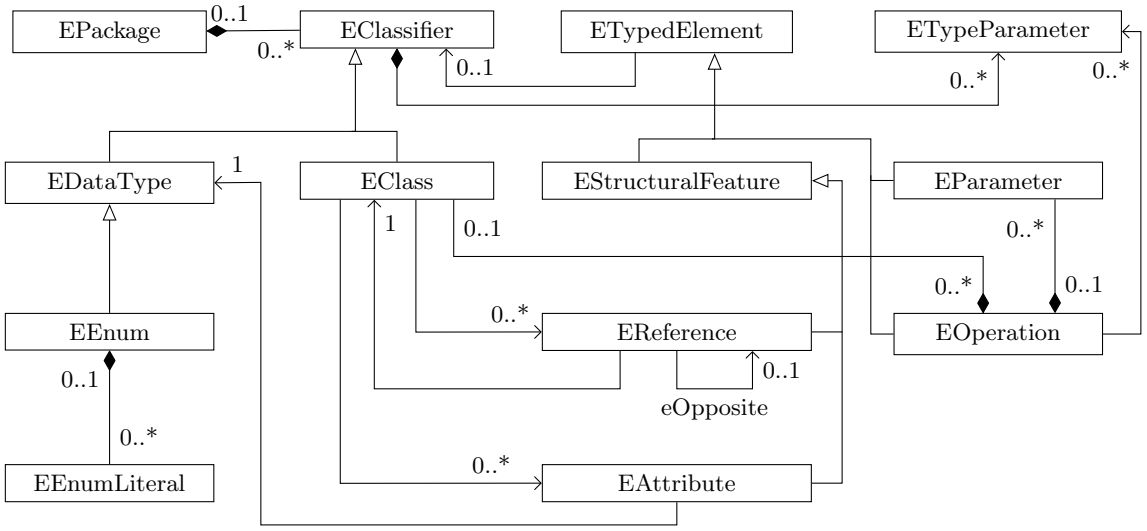


Fig. 2.2.: A simplified representation of central concepts of EMF's meta-modelling language Ecore (cf. [FBFG08]).

The default tools provided by EMF such as code generators, editors, and runtime platforms use Eclipse's built-in extension facilities to allow for customisations. Extensions can, for example, provide serialisation possibilities in addition to the default format XMI. In the last years an active community in research and industry created a diverse ecosystem of projects that use EMF as a foundation or interoperability interface.

2.2. Aspect-Oriented Modelling

Modelling concepts not only for cross-cutting concerns

Aspect-Oriented Modelling (AOM) allows modellers to describe cross-cutting concerns using concepts that are specifically designed for this purpose. A concern is said to be cross-cutting when it affects numerous areas of a system that belong to different parts of the predominant structuring. Some AOM approaches regard Aspect-Orientation as a design paradigm that should influence all design phases similar to other paradigms such as Object-Orientation or Service-Orientation. Other AOM approaches, like the one presented in this thesis, make it possible to take advantage of aspect-oriented concepts in

[4]EMF - Eclipse Modeling Framework: eclipse.org/modeling/emf
[5]Eclipse - An Integrated Development Environment: eclipse.org

ordinary environments and processes. Both perspectives have been proven successfull in industrial studies, e.g. by Rashid et al. [RCG+10], and in academic studies, for example, by Molesini et al. [MGvFGCB10].

The fundamental concepts of AOM are similar to those of other aspect-oriented techniques such as Aspect-Oriented Programming (AOP). A *pointcut* describes which conditions have to be satisfied at the points of a model *where* an aspect should apply. Therefore, a pointcut is similar to the search pattern in a search-and-replace task for a text. An *advice* defines *what* should be done whenever a part of a model matches the description of a pointcut. In the search-and-replace analogy it corresponds to the replace pattern that may refer to capturing variables defined in the search pattern. The model to which an aspect should be applied is called *base model*. Areas of a base model that match a pointcut are called *join points*. In AOM as well as in AOP there exist approaches with explicit and implicit join points. This terminology is common to many but not all AOM approaches. Explicit join points, for example, can take the form of annotations and might therefore also be named accordingly. A principle that is established by various proponents of Aspect-Orientation is called obliviousness. It demands that base models should not have to take into account that aspects might be applied to them. Therefore, explicit join points can be considered to break this principle.

When aspects are applied to a base model this process can usually be broken down into two parts. In the first step the points where the aspect should be applied (join points) are detected. The second step executes the changes that are described in the advice at the detected join points. This process of incorporating the properties of the advice into the base model is also called *model weaving* [GV07, Jéz08]. The order in which join points are processed during the weaving and how conflicts are resolved is important in case of overlapping join points.

At which level of abstraction aspect-oriented techniques should be applied is debatable. After an initial hype around AOP it was questioned whether it would be better to use aspect-oriented concepts at earlier stages of the development process. One of the reasons for this demand where technical details and difficulties that appear when code has to be woven. Another factor was the observation that many program aspects where rather part of the solution than the problem domain. Some researchers attributed these deficiencies to the aspect-oriented paradigm in general. Others developed AOM approaches that can be applied at higher levels of abstraction than the code level. Spanning from requirements elicitation to detailed design, these AOM approaches use very different notations but share various concepts such as pointcuts and join points. A case study presenting some AOM approaches at different levels of abstraction was presented by Alférez et al. [AAC+11].

In this thesis we present a generic approach to model weaving that was not optimized for a certain notation or a specific aspect-oriented development process. As a result it may provide less sophisticated aspect features than other approaches. But a resulting advantage of our approach is that its functionality can be used for arbitrary notations and development processes without any aspect-orientation. We discuss other generic approaches to model weaving in detail in Chapter 11.

2.3. Building Information Modelling

Semantically rich 3D building models

In the construction industry the term Building Information Modelling (BIM) [ETSL11] refers to building models that contains semantic information in addition to three-dimensional geometric information. According to Howard and Björk [HB08] the shift away from two-dimensional models towards these digital three-dimensional models

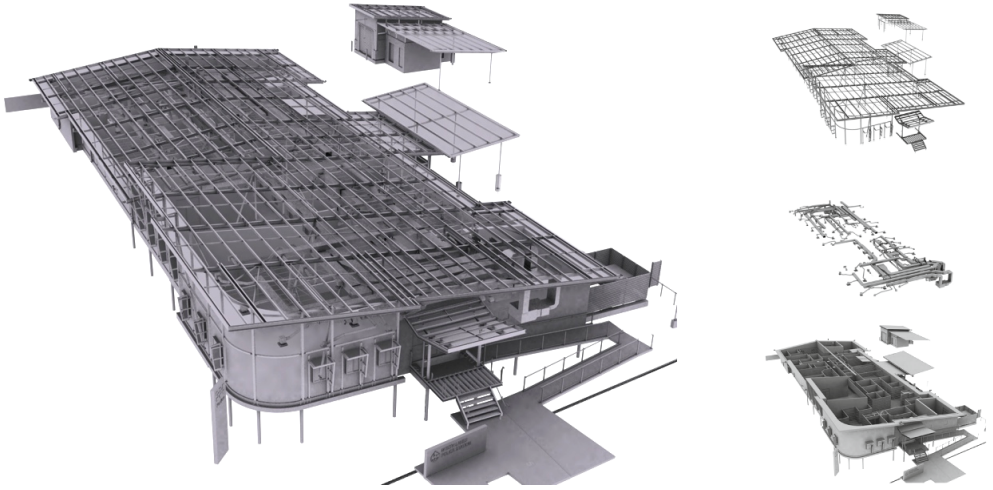


Fig. 2.3.: Combination of an architectural, structural, and mechanical model from Queensland Government Project Services showing a suburban police station.

started during the last decade. A potential benefit of BIM is that semantic information, such as used materials, can be used for automated tasks by the different stakeholders involved in a construction project. Although it would be possible to include all available information in a single common building model this is usually not done. Instead, partial models are used by contractors, which results in a heterogeneous system of models for a single building project, which poses challenges in terms of integration according to Shen et al. [SHM+10]. Fig. 2.3 presents a combination of the architectural, structural and mechanical model of a police station realised by the Project Services business unit of the Department of Public Work of the Government of Queensland, Australia.

Most Computer Aided Design (CAD) tools that support BIM use proprietary formats for representing and rendering building models. Nevertheless, interoperable CAD tools usually provide import and export functionalities for the de-facto standard Industry Foundation Classes (IFC), which we present in Section 2.3.1. We discuss the framework that we use to bridge the IFC and EMF technological spaces in Section 2.4.

2.3.1. Industry Foundation Classes

A de-facto standard for BIM

Within the Standard for the Exchange of Product model data (STEP), the BIM format Industry Foundation Classes (IFC) is defined using the schema language Express. STEP was designed as an ISO standard for engineering products that focuses on the exchange of information throughout their life cycle. IFC makes use of the geometry model of STEP and defines a wide range of ready-to-use objects for building modelling. In addition it provides an extension mechanism using proxy objects. IFC models support various serialisation formats such as ASCII text or XML files. In CAD tools IFC is usually used as exchange format as proprietary formats, which are optimised for performance, are used for the internal representation of building models.

2.3.2. Building Specifications

A text on details and standards

When buildings are designed with a size that cannot initially be tackled in full detail, a common technique is to begin with a rough model and to define details in a document called *building specification* [ETSL11]. This natural language text is also used when contractors disagree on questions such as how a building has to be build and therefore the specification documents have a legal character. But this is not the only reason why

building specifications are created. An important facet of building specifications is that detailed information that would have to be added at various places in a building model can be described. Together with the building model the building specification is used as the main input for various analysis tasks like cost estimation.

The type and level of detail of the information put into a building specification varies, for example, with regard to the properties of the project, the contractors, and the legal standards. A key difference to building models is that it is also possible to describe how work has to be conducted and not only what should be the result of it. Therefore, phrases that demand, for example, the provision of a certificate showing that some quality attributes are met can also be part of a building specification. Another peculiarity of building specifications results from the fact that they are written using natural language. On one hand, this makes it possible to define complex conditions that have to be satisfied for a requirement to apply. A specification sentence may, for instance, link the number and location of fire alarms to the apartment type in a project that involves various apartments of different type. On the other hand, like all natural language texts building specifications may be ambiguous and open to different interpretations. We discuss our proposition that building specifications contain cross-cutting concerns in detail in Chapter 8.

2.4. From Building to Software Models

Bridging instead of converting

In order to apply MDE techniques that are based on EMF to building models, we will use a bridge for the technological spaces EMF and STEP that was presented by Steel et al. [SDD11]. The goal of this bridge is to make building models accessible for the various approaches and tool platforms that have been created for EMF in the MDE community. Another possibility to reach this goal would be the application of format conversions, which would result in the loss of information because of the incompatibility of the involved formats. Technological bridges, such as the one for EMF and STEP respectively IFC, have the advantage that they represent a document of a technological space in the format of another technological space without explicit conversion or information loss.

The bridge between EMF and STEP is one example of a more general concept for bridging technological spaces. The notion of technological spaces and operational bridges between such spaces was proposed by Kurtev et al. [KBA02]. Later on Bézivin et al. [BDD+06] presented an example of such a bridge for model and ontology engineering. This bridge has been used and improved in various research projects and disciplines such as Biomedical Informatics [MCMTFBM09]. Other technological bridges such as one between the MDE tools from Microsoft and those built on top of Eclipse keep on being investigated [BCC+10].

For bridging EMF and STEP it is beneficial that both have a three-layered architecture, consisting of metametamodels, metamodels, and models. This makes it possible to implement the bridge using a promotion transformation in a three-stepped process. Fig. 2.4 visualises these three steps and the involved models and layers. In the first step an Ecore metamodel for the Express language is defined and an Express schema for IFC is loaded as an instance of this metamodel using an EMF Resource Implementation. Second, a model transformation is used to promote an Ecore metamodel for IFC out of the model instance gained from the first step. The trace information of this transformation is stored as an instance of an Express2Ecore metamodel. In the third and last step these traceability links are used in order to represent an IFC building model in the EMF space. A building model that was serialized as an IFC ASCII file is

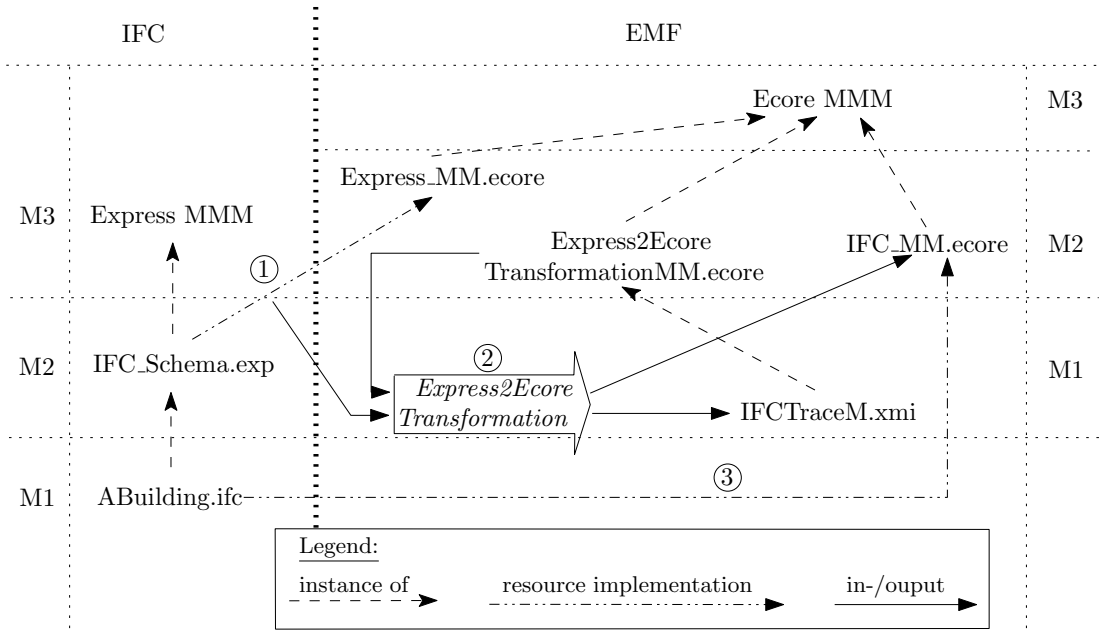


Fig. 2.4.: The models involved in the three steps of the IFC / EMF bridge aligned according to their meta-level in their technological space ([SDD11]).

made available as an instance of the Ecore metamodel for IFC which was gained from the second step.

The technological bridge between EMF and STEP provides benefits to stakeholders of both technological spaces. As EMF-based tools can be applied to IFC building models without explicit import and export operations the construction industry obtains additional means to leverage their models. The MDE world can as well profit from the gained accessibility of building models. BIM models present challenges to MDE approaches in terms of scalability and integration because of their comparatively huge size and differing perspectives of involved stakeholders.

Part II.

A Generic and Extensible Approach to Model Weaving

3. Overview & Key Characteristics

In this section we outline the overall architecture and key features of our generic model weaving approach. First, we describe five key features that characterise our approach in addition to the genericity and extensibility explained in Chapter 4. Then, we outline the individual phases of weaving that structure our approach.

3.1. Key Characteristics of our Weaving Approach

Our approach in two sentences

Our approach performs asymmetric model weaving based on implicit join points and declaratively mapped pointcut and advice models, which are defined using existing modelling languages. The induced composition operations are carried out in a way that is independent of the involved metamodel. We present these five key characteristics of our approach individually in the following sections.

3.1.1. Asymmetric Weaving of Ordinary Models

In our approach aspects defined by a pointcut model and an advice model are woven into a *base model*. The *pointcut model* of an aspect contains the model elements at which the aspect should apply. It expresses the constraints that a region of the base model has to satisfy in order to be affected by the aspect. The aspect's *advice model* contains the model elements that should be present after applying the aspect to such a region. It expresses the constraints that each affected region satisfies once the aspect was woven into the base model. The difference between an aspect's pointcut and advice model implicitly defines the changes that the weaving of the aspect will cause.

Such a model weaving approach is called asymmetric because the three input models for the base, pointcut, and advice have different roles. This means that the weaving result is not only determined by the individual contents of the input models but also by the role that they are given. Weaving the same models while interchanging their roles results in different woven models. This is not the case for symmetric approaches, which weave entities of the same role, for example, aspects with other aspects.

3.1.2. Implicit Join Points Allow Direct Use

In order to modify existing models at arbitrary points, our approach uses the well-known concept of implicit join points, which are identified using a join point detection mechanism. This means that a pointcut model is defined as a model snippet that can

contain any arbitrary standard model elements in order to define the regions at which base models should be changed. Therefore, our weaving approach can be directly used with existing modelling notations. No preparatory steps like annotating a model or designing and executing mark transformations are needed. In other words, our approach satisfies the controversial requirement of obliviousness [FF00], i.e. base models are oblivious to the aspects applied to them.

No genericity without implicit join points

Some approaches to AOM and AOP use explicit join points, which make it possible to control aspect application in base models. The possible consequences of explicit join points, such as avoiding aspect interference and the break of obliviousness, are widely debated in the community. Our weaving approach, however, is designed to be independent of the modelling notations to which it is applied. Therefore, explicit join points would have to be realised in a way that adds them to existing metamodels, for example, using annotations. Our approach could also support such explicit join points but as this was not required for building models we do not need to further discuss them at this point.

3.1.3. Aspect Definition Using Familiar Syntax

In our approach pointcut and advice models are defined using a variant of the original metamodel. In this variant, constraints, such as lower bounds and abstract metaclasses, are relaxed in order to allow the definition of incomplete model snippets. As a result, it is possible to define aspects with pointcut models that specify only the part of a model region that is relevant for the application of the aspect. That is, model elements that are irrelevant for the aspect but that would be required in order to meet the constraints of the metamodel can be omitted in the pointcut and advice model.

As the relaxed metamodel variant is a subtype of the original metamodel, our approach can be used to define aspects using existing editors that were designed for the original metamodel. Therefore, it is neither necessary to introduce aspect-oriented concepts into existing modelling languages to which our approach is applied nor do existing tools that manage these models have to be changed.

The concept of relaxed metamodel variants was previously presented by Ramos et al. [RBJ07]. To our knowledge, it has not been realised in a way that is fully independent of the involved metamodels. We are only aware of approaches that apply the metamodel relaxation on a specific metamodel in contrast to our approach, which derives the relaxed metamodel variants automatically.

3.1.4. Declarative Mapping from Pointcut to Advice

Specifying what to weave instead of how to weave

In our approach, users declaratively define which elements of the pointcut correspond to which elements of the advice. This indirect execution definition relieves the user from the need to explicitly specify weaving steps. Instead of expressing *how* an aspect should be woven it is sufficient to specify *what* should be woven. This is done by defining an ordinary m-to-n mapping from pointcut to advice elements in a mapping model that serves as further input model in addition to the base, pointcut, and advice model. In most cases it is, however, not even necessary to explicitly define this mapping as it can be determined automatically for pointcut and advice models with unambiguous correspondences (see Section 3.2.3).

The foundations of declarative weaving instructions have been presented previously by Morin et al. [MKBJ08] for the predecessor of the approach described in this thesis. As no other approach to model weaving uses such a mapping from pointcut to advice elements it remains a unique feature of this enhanced approach.

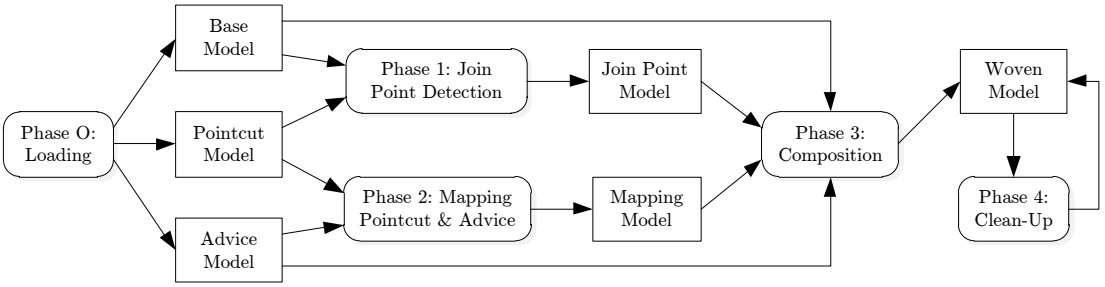


Fig. 3.1.: A UML activity diagram showing the five weaving phases together with the three input models, the two intermediate models, and the output model.

3.1.5. Metamodel-independent Operations

As our approach is defined on the metametamodel level, instances of any kind of metamodel can be processed. For example, instances of a building metamodel can be processed in the same way as instances of UML metamodel. This is possible because our weaving operations are based on the features[1] of the metamodel to which the model conforms. Such metaclass features can be attributes, which store primitive types, or references to complex types. Features in the form of attributes and references are part of several metametamodelling languages, such as Ecore and KM3 [JB06]. Therefore, our approach can be used for models conforming to metamodels that conform to different metametamodels.

Metamodel-independent operations for model weaving have already been proposed by Morin et al. [MKBJ08] but have never been realised in a completely generic way. To our knowledge, other approaches to model weaving implement operations that are conceptually generic but formulated in a way that is specific to used metamodels. This is not the case for the initial implementation[2] of the approach presented in this thesis. It is based on the metametamodelling language *Ecore*, which is the Eclipse Modelling Framework (EMF)'s variant of the OMG standard EMOF 2.0. The weaving operations of this implementation are defined in a metamodel-independent way as they directly operate on the features defined using the Ecore metamodelling language.

3.2. Weaving Phases

Our approach is structured in five different phases of weaving, which we visualise in Fig. 3.1 using an UML activity diagram. The overall control and data flow is as follows: After loading the base, pointcut, and advice model in the initial phase, these models are used as input for the phase of join point detection and the phase inferring a mapping from pointcut to advice elements. Note that detecting join points is independent of advice models and that the pointcut and advice models can be mapped in parallel as they are isolated from the base model. Both phases produce result models used for the central composition phase: a join point model and a mapping model. The woven model created in this composition phase acts as input and output of the final clean-up phase before it forms the overall result of the weaving process. All weaving phases are outlined individually in the following sections and discussed in detail in Chapter 5.

[1] Features in the sense of properties, not in the sense of functional units as in Software Product Lines.
[2] code.google.com/a/eclipselabs.org/p/geko-model-weaver

3.2.1. Loading

At the beginning of the weaving process all input models are loaded in order to make them available for the rest of the weaving, which is independent of the used serialisation and persistence technologies. We decided to mention this trivial phase because it can be customised to account for non-standard serialisations (see Section 4.2.2), which was necessary in the case of building models (see Section 10.1).

3.2.2. Join Point Detection

This phase of weaving identifies all regions of the base model that match the model snippet defined in the pointcut model. As an intermediate result we obtain for each join point a one-to-one mapping from elements of the pointcut model to elements of the base model. Depending on the structure and size of base and pointcut models this preparatory step can dominate the overall time required for weaving. For this reason, we decided to decouple it completely from the other phases in order to allow for possibly domain-specific customisations and optimisations of this import phase.

In our initial implementation we perform join point detection using the business logic integration platform Drools[3], which implements the Rete algorithm [For82]. For every pointcut model we generate Drools rules using a two-pass visitor and execute these rules on a knowledge base containing all elements of the base model. This is similar to the SmartAdapters approach presented by Morin et al. [MKKJ10]. The main difference, however, is that we do not generate rules for advice instantiation but separated this advice-specific step from the advice-independent process of join point detection. This allows for separate customisation and evolution of the weaving phases corresponding to these individual steps.

3.2.3. Inferring a Pointcut to Advice Mapping

In this step we infer a mapping from pointcut-model elements to advice-model elements. This mapping specifies how elements to be found prior to weaving correspond to elements to be present after weaving. As it is a m-to-n mapping it may relate multiple pointcut elements with multiple advice elements, which induces duplication and merge operations discussed in detail in Section 5.3.3.1 and Section 5.3.3.2.

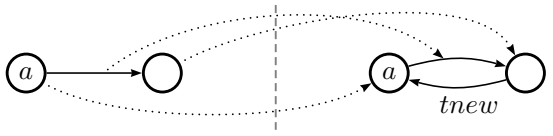
Automatic inference of unambiguous correspondences

To relieve the user from as much complexity as possible, we decided to infer the mapping automatically for unambiguous aspects. Such an unambiguous aspect is given when every pointcut element matches at most one advice element of the same type that has the same primitive attributes. Fortunately this happens to be the case for many weaving scenarios, such as the one presented in Fig. 3.2. It depicts an aspect for a Labelled Transition System (LTS), which encapsulate the core concepts that are fundamental to behavioural modelling notations such as UML state machines. The pointcut model of this aspect matches all states named a with an arbitrary transition to a state that can bear any name. As it is a strictly additive aspect the advice model contains exactly the same elements as the pointcut model. In addition, it contains a new transition *tnew* from the nameless target of the matched transition to a. How the unambiguous correspondence between these model elements is inferred is detailed in Section 5.2.

3.2.4. Model Composition

The central weaving phase composes the base and advice models by merging the values of the features of the metaclass instances contained in these models. Feature values of

[3] jboss.org/drools



(a) pointcut model (b) advice model

Fig. 3.2.: An example of a pointcut and advice model with an unambiguous mapping from pointcut to advice elements, which can be automatically inferred.

the advice are used to replace or complete base feature values but removal operations are deferred to the last phase of weaving. The basis for the model composition is the inferred or user-defined mapping from pointcut to advice elements and the mapping from pointcut to base elements, which is induced by a join point. Using these mappings elements of the base model that have to be created, merged, or duplicated are identified and these operations are performed directly by modifying the elements' metaclass feature values. These composition operations and exemplary weaving scenarios in which they occur are discussed in detail in Section 5.3. At the end of this central weaving phase, after all composition operations have been carried out, it is ensured that all newly created elements are added to their containers using the correct containment references.

3.2.5. Removal & Clean-up

In a last phase of weaving base elements that correspond to pointcut elements but that do not correspond to any advice elements are removed from the woven model. As other elements of the woven model may still refer to these removed elements this removal has to be followed by a clean-up operation. References to removed elements are deleted during this clean-up in order to keep the woven model consistent. If model elements violate lower bounds of reference features as a result of these removals they are removed as well. This is necessary to guarantee that woven models still conform to their metamodel. Details and examples for this removal of inconsistent elements are presented in Section 5.4.

3.3. Summary

In this chapter we introduced our approach to generic model weaving and gave an overview of its key characteristics and its individual phases of weaving. The characteristics concerned input symmetry, join points, syntax reuse, pointcut-advice correspondence, and the weaving operation's level of abstraction. We presented the underlying design decisions, mentioned how we realised them, and lay out the consequences. Using the individual phases of weaving, we presented the overall weaving workflow together with the involved models and intermediate results. In the following chapter we discuss two outstanding qualities of our approach, which we omitted in this introductory chapter: genericity and extensibility.

4. Achieving Practical Genericity through Extensibility

In this chapter we explain which properties render our model weaving approach generic and extensible so that it can be applied to and customised for arbitrary models. First, we describe how weaving operations that work on the features defined in the metamodel ensure that our approach can be applied to models conforming to arbitrary metamodels. Then, we illustrate how extension possibilities render this genericity practical through domain-specific customisations that complement generic weaving behaviour.

4.1. Genericity

In our approach genericity is achieved through weaving operations that are based on metamodel-features and that are induced by pointcut and advice model snippets that are defined using relaxed metamodel variants. How this metamodel-driven weaving is performed and why its aspects can be defined using existing languages and tools is detailed in the following sections.

4.1.1. Operating on the Metamodel Level

The key design decision that makes our approach generic is to transform models solely by operations formulated on the metametamodel level. These operations allow us to add, change, and remove elements of a metamodel instance using the features of the metamodel that are expressed using a metametamodelling language. Let us illustrate this using a small example. Suppose a single element b of a join point of a base model matches an element p of the pointcut model. Let this pointcut element p be mapped to an advice-model element a. This yields a correspondence between b and a resulting in a woven model in which the base element b exhibits the features of the advice element a. In order to perform this weaving it is irrelevant whether the model elements b and a are, for example, entities of a UML diagram or elements of a building model. It is sufficient to inspect and update the values of all the features that are defined in the metamodel for the metaclasses of b and a.

Syntactic vs. semantic weaving

This model weaving based on syntactic features defined in the metamodel has advantages and disadvantages. On the one hand, because we ignore concrete syntax and semantics during our abstract-syntax-based weaving operations we can weave models of any modelling notation. As a result, it is sufficient to provide a metamodel when

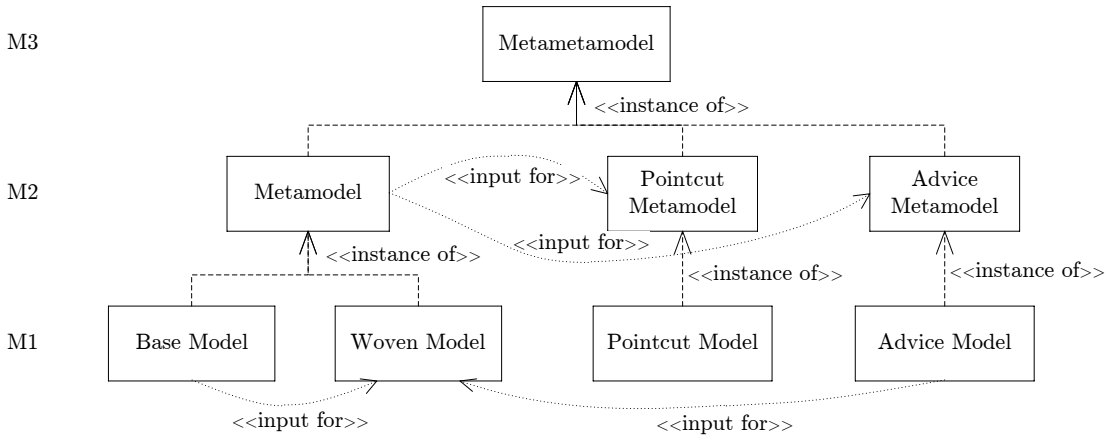


Fig. 4.1.: The models involved in the generic weaving process together with their metamodels and metametamodel aligned to the model layers M1, M2, and M3.

our generic weaving approach should be applied to models conforming to a metamodel to which it was never applied before. On the other hand, this syntactic focus means that semantic weaving is only possible when the default behaviour is changed using extensions. In cases in which the semantics of a modelling notation should be taken into account during the detection of join points, model composition, or any other weaving phases this has to be explicitly defined. As we are not aware of a generic approach to semantic model weaving we consider this need for extensions an acceptable hindrance.

4.1.2. Reusing Languages and Tools for Aspects

A key requirement for our metametamodel-centred approach is that users are able to formulate pointcut and advice model snippets for the metamodel of their choice. Such model snippets, however, do not need to satisfy all constraints defined in this original metamodel. For this reason, we create two metamodel variants with relaxed constraints, which are used to define pointcut and advice models. These relaxed metamodel variants are automatically obtained from the original metamodel as described by Ramos et al. [RBJ07]: invariants and pre-conditions are removed, all features of all metaclasses are specified as optional (lower bounds for references are set to zero), and all abstract metaclasses are made concrete (i.e. direct instances are no longer forbidden). All metamodel variants used in our weaving approach are presented in Fig. 4.1 together with the model instances conforming to them.

A convenient consequence of the use of metamodel variants for pointcut and advice models is that users are able to define aspects using the syntax that they are already familiar with. With our generic model weaver, defining an aspect means describing two model snippets (pointcut and advice) using the same notation as for ordinary models. No aspect languages or AOM concepts have to be applied. Additionally, the relaxed metamodel variants are supertypes of the original metamodel as they only relax constraints without removing or adding any elements. Therefore, it is not only possible to use the original syntax of a modelling language for pointcut and advice models but even the corresponding editors and other tools can be reused.

Complementing generic weaving by extensions

4.2. Extensibility

Even though our approach is generic, it may not handle all weaving circumstances for all metamodels in the way desired by its users. Therefore, we decided to give users

the ability to decide for each part of our generic approach whether the generic weaving behaviour should be used without modifications or whether it should be completed or replaced with custom weaving behaviour. In this section we briefly present the extension possibilities of our approach. How these extensions are used for a particular domain is illustrated in Section 10.1 using the example of IFC building models.

4.2.1. Extension Types

Different types of extensibility

Our model weaving approach provides extension possibilities differing in terms of co-existence with default behaviour. Some of the extension points that we provide can be used to change the default weaving behaviour. This means extensions to such extension points are executed instead of the generic default instructions. Other extension points can be used to perform custom work in addition to general weaving operations. Extensions to such extension points are executed after or before generic weaving instructions depending on the nature of the task. If multiple extensions to such an additive extension point are provided, the order of execution is handled based on a priority attribute. Each extension provides a priority level specifiying at which point it should be executed with respect to other extensions and with respect to the standard behaviour.

Extensions can also vary with respect to the level of detail provided by them. For some tasks we provide two extension points in order to give users the ability to provide simple as well as more elaborate extensions. When only parts of such a task have to be customised the remaining parts are handled in a generic way. As a result, users of our weaving approach can decide on a finer level of granularity how to customise the default behaviour.

Providing an extension to an extension point of the prototype implementation of our generic model weaver is largely independent of the extension type. A client class has to implement the Java interface provided by the extension point and the plug-in containing this client class has to declare this extension. The level of detail is determined by the interface corresponding to the individual variant of the extension point and a priority is simply returned using a corresponding method of the interface. Further details on the realisation of the extension mechanism are provided in Section 7.1.2.1.

4.2.2. Extension Points

This section presents the individual extension points at which our weaving approach can be adapted to specific needs. All extension points are discussed individually in order of their use during the weaving process:

EP 1 *Generating Metamodel Code*: After the preparatory derivation of relaxed metamodel variants for pointcut and advice models the default generator models for these variants can be modified. These generator models specify how a Java infrastructure, which realises all metamodel elements, is generated. This is necessary because the code that was generated for the original metamodel cannot directly be used for model instances of the relaxed metamodel variants. Extensions to this extension point can be used in order to generate code for the metamodel variants that exhibits the same features as the code for the original metamodel. This may be necessary in contexts in which special serialisations or functionalities such as locking are needed.

EP 2 *Loading*: The process of loading and storing models before and after the actual weaving can be customised using a simple and a detailed extension point. Simple extensions can override the retrieval of a model resource for a given Unique Resource Identifier (URI). Detailed extensions can, for example, perform custom actions directly

before a model is stored. They can also change the way all contents of a model are iterated or how root elements of a model are obtained.

EP 3 *Detecting Join Points*: Join points can be detected in a completely customised and separated step. This is possible because this weaving phase is independent of all other phases and just needs a base model and pointcut model as input. For every detected join point it returns an ordinary one-to-one mapping from a pointcut element to a base element. Extensions can provide domain-specific performance enhancements or functional customisations, such as support for semantic weaving as presented by Klein et al. [KHJ06]. This is necessary when join points do not always appear with the same syntax in the base model. Sequence diagrams, for example, can contain loops in which the occurrence of a join point is distributed over multiple iterations. Messages of such a join point do not have to appear syntactically after each other. Nevertheless, they should be detected if the semantics of the loop reveal that they will be sent after each other. An example would be a sequence of two messages in which the first message occurrs at the end of iteration i and the next message occurs at the beginning of iteration $i + 1$. When such a semantic matching is desired, a domain-specific extension should provide the necessary join point detection mechanism.

EP 4 *Ignoring Attributes*: It is possible to ignore specific features of metaclasses during join point detection and model comparison using an extension point. This makes it possible to detect join points or determine equivalence for two model elements in cases in which the values for these ignored attributes or references differ.

EP 5 *Identifying Aspect Mappings*: For the automatic inference of a mapping from pointcut elements to advice elements the calculation of unique identifiers can be customised. Whenever two elements have the same unique identifier it is inferred that these elements correspond. The matching algorithm presented in Section 7.3.2 ensures that it sufficient to change the way these identifiers are calculated in order to completely customise the mapping inference. An example showing how these identifiers induce the mapping from pointcut elements to advice elements is presented in Section 5.2.

EP 6 *Creating New Elements*: The creation of new base elements corresponding to advice model elements that do not have associated pointcut elements can be customised. Extensions to this extension point can change the way that elements and values of the advice model are copied to create new base elements. Additionally, such extensions can influence the decision whether a new element should be created or an existing element should be used. This can be used to ensure that elements introduced in advice models are, for example, instantiated globally or per join point. Details of this advice instantiation possibilities are discussed in Section 5.3.2.

EP 7 *Determining Containment*: Various meta-modelling languages require that a model element is unambiguously contained in exactly one model element or the root element of the model. With this extension point it is possible to customise the determination of containment references and containers for elements that are introduced into a woven model without containment that can be inferred from the advice model. Such elements are not contained within an advice element that corresponds to a base element at a join point. Therefore, the strategy to use the containment information available at a join point in order to add new elements does not work for such elements. Nevertheless, we try to determine a containment reference and container for such elements on a best-effort basis. In cases where this does not yield the desired results extension can use domain-specific knowledge to improve on this.

We visualise the presented extension points and the weaving phases in which they occur in Fig. 4.2. It is a variant of Fig. 3.1 from Section 3.2, which relates the extension points

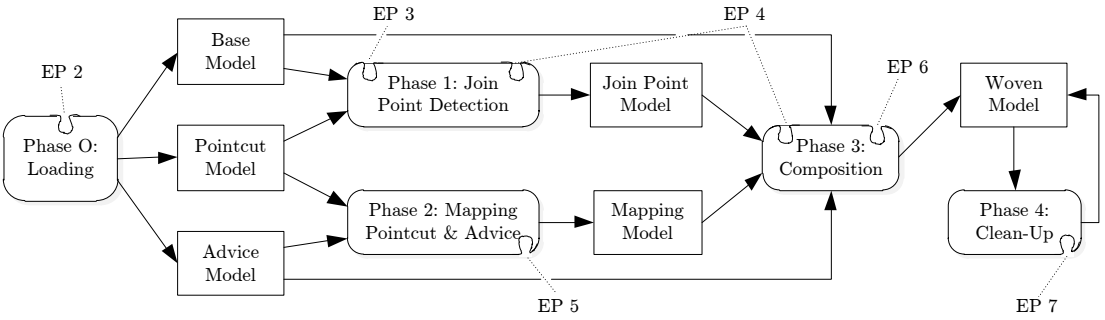


Fig. 4.2.: The weaving phases of our approach and their influencing extension points.

to the influenced weaving phases. Note that EP 1 (Generating Metamodel Code) is not depicted in the figure as it influences the derivation of relaxed metamodel variants. This preparatory step needs to be executed once for a metamodel and is not part of the weaving process for individual instances of that metamodel.

4.3. Summary

In this chapter we presented the two main qualities that distinguish our approach from other model weaving approaches: genericity and extensibility. First, we explained how we realised generic model weaving through operations on the metamodel-level and through aspects that are defined using automatically derived metamodel variants. Then, we argued that such generic weaving behaviour has to be customisable in order to be practically useful and presented the extension possibilities designed for this purpose. For each individual extension point we explained its necessity and showed how it is embedded in the overall weaving phases workflow. In the next chapter we discuss all these individual phases of weaving in detail, present their formal foundations, and provide illustrative examples.

5. Detailed Weaving Phases Discussion

In this chapter we discuss the individual phases of our model weaving approach in detail. First, we explain the preparatory phases, which detect join points and map pointcut to advice models, using a running Labelled Transition System (LTS) example. We chose LTS because it is a compact and well-known formalism that does not distract the reader from the essential weaving operations. Second, we present the central composition phase, the underlying formalisation, available advice instantiation strategies, as well as duplication and merge operations. These operations are illustrated using different examples for LTS and building models in order to highlight the applicability to different metamodels. Finally, the last weaving phase for removing model elements and cleaning up the woven model is discussed.

5.1. Join Point Detection

Weaving-independent pattern matching

The join point detection process finds all locations in a base model that satisfy the conditions defined in a pointcut model snippet. As a result a one-to-one mapping, which relates all elements of the pointcut model to the corresponding base-model elements, is obtained for each join point. How this mapping is obtained is irrelevant for the actual weaving operations and the structure of the woven model. Therefore, this step can be performed in complete isolation of other weaving steps and can be completely customised to specific domains. A symbolic representation of a join point in the context of all models that are involved in the weaving process is shown in Fig. 5.1. In this representation the unmapped elements in the base model (not filled) indicate that we are presenting the mapping for one out of multiple possible join points. Furthermore, different shapes represent model elements of different type. As the figure depicts the weaving process after join point detection, no mapping between pointcut and advice elements has been established and the woven model is still empty.

Let us illustrate the notion of join points using an example for LTS, which we will also use to explain the mapping from pointcut to advice elements and the set-theoretic composition formalisation. Fig. 5.2 presents the base model, pointcut model, and the two resulting join points of this example. The base model contains an initial state a with two transitions, which are triggered by $t1$ and $t2$ and target the states b and c. For the rest of this thesis we will use the input by which transitions are triggered to name the transitions of our LTS examples. Further transitions and states of the base model do not influence the detected join points but are displayed in order to show that

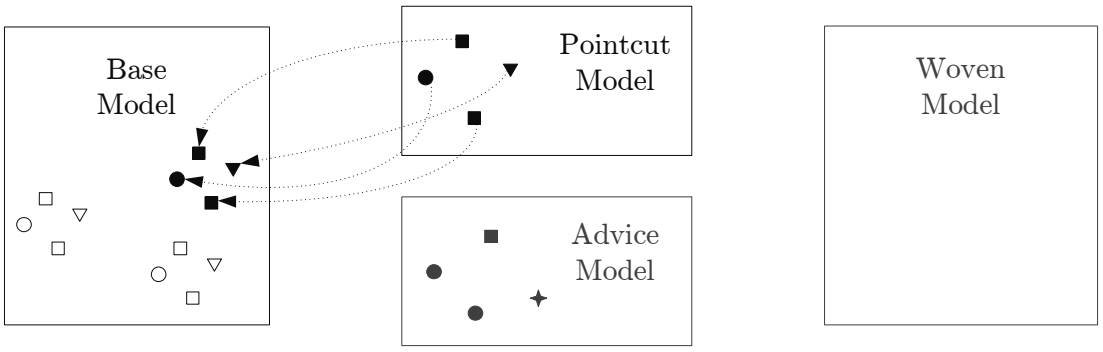


Fig. 5.1.: Symbolic representation of a join point mapping four pointcut elements to elements of a base model, which contains two more join points.

the base model is a complete LTS containing further elements. The pointcut model is an incomplete LTS snippet as it exhibits no initial state. It contains a state a with a transition, which is triggered on any input and targets a nameless second state. Recall that a join point has to map every element of the pointcut model to exactly one element of the base model. As the base model contains one state named a with two transitions we obtain two join points. One join point that identifies the pointcut transition with the base model transition $t1$, and the nameless target state with the base model state b (Fig. 5.2(c)). And another join point that matches the transition $t2$ and the target state c of the base model (Fig. 5.2(d)).

The default join point detection mechanism of our approach is based on the business logic integration platform Drools[1]. It provides the possibility to formulate rules that are executed on a knowledge base using the Rete algorithm[For82], which performs forward chaining starting with the rules. To perform join point detection we express a pointcut model as Drools rules and execute these on a knowledge base that contains all base-model elements. The algorithm that we use to generate these rules by visiting all pointcut elements twice is discussed in Section 7.3.1.

Let us continue to illustrate the overall join point detection using our running LTS example. Before we go into detail we present a visualisation of the metamodel for LTS in Fig. 5.3. It shows that an LTS has a name and consists of an arbitrary number of named States. Exactly one of these states is designated as initial state and an arbitrary number is given the role of final state. Each state acts as source for an

[1] jboss.org/drools

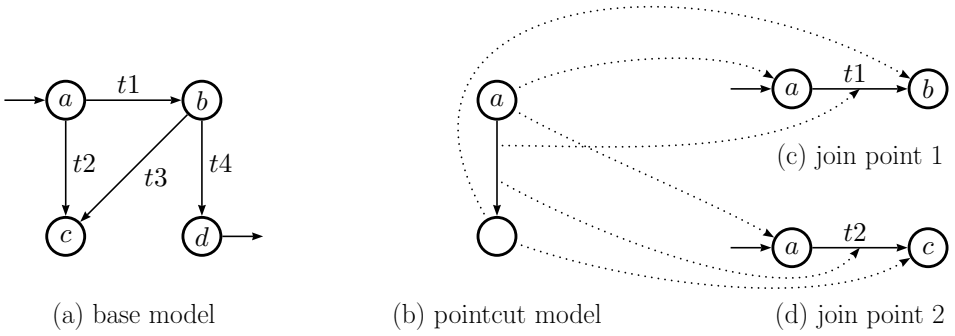


Fig. 5.2.: The base and pointcut of our running LTS example and the resulting two join points as involved elements and mappings from the pointcut to the base.

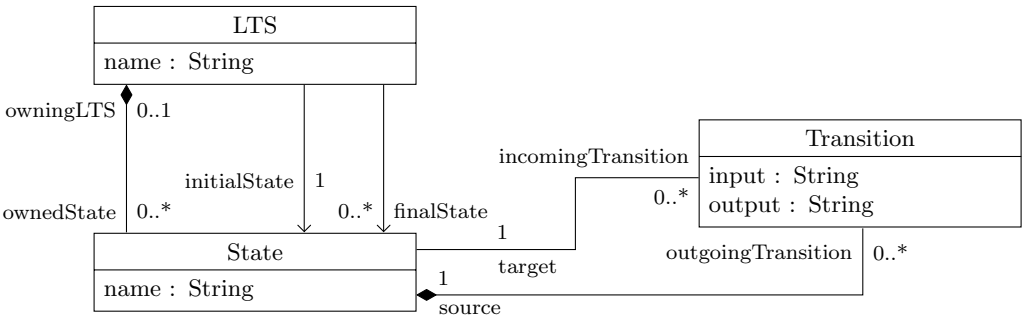


Fig. 5.3.: A UML class diagram showing the LTS metamodel based on [MKBJ08].

arbitrary number of outgoing transitions and serves as target for an arbitrary number of incoming transitions. These transitions are characterised by an input string and an output string.

Translating metamodel instances to matching rules

The rules that we generate for the pointcut model of our running example shown in Fig. 5.2(b) are presented in Fig. 5.4. They are divided into two parts handling attributes and references separately: In the first part, line 1 specifies that we are searching for an instance of the metaclass *State* having *a* as value for the attribute *name*. Then, line 2 demands an instance of the metaclass *Transition* with no restrictions for the attributes *input* and *output*. Last, line 3 requires another instance of the metaclass *State*, which is not restricted with respect to its only attribute *name*. In the second part, line 4 requires that state *a* links to the transition using its *outgoingTransition* reference. In line 5 the transition's opposite reference named *source* is required to point to the state *a*. Furthermore, the *target* reference is required to link to the nameless state. Likewise, the nameless state's *incomingTransition* reference, which is the opposite of the transition's *target* reference, has to list this transition as specified in the last line 6. All these requirements are met by the elements *a*, *t*1, and *b* for the first join point and by *a*, *t*2, and *c* for the second join point. The formal model composition process following the detection of join points for this small LTS example is presented in Section 5.3.1.

For other models and metamodels join point detection is performed in the same way: The application restrictions of a pointcut model are expressed as rules that list its elements and their attribute values and references. These rules are automatically generated based on the metamodel to which the model conforms. Actual detection is delegated to the Drools query mechanism that is given the rules and a possibility to access the elements of the base model.

An implementation of a predecessor to our model weaving approach, which was presented by Morin et al. [MKBJ08], did not support automatic join point detection. More recent implementations of related approaches, such as SmartAdapters by Morin et al. [MBNJ09], perform join point detection using the logic programming language Prolog or using the rules engine Drools. In contrast to our solution, they were not designed

```
$s0Decl: lts.State(name == "a")                                                    1
$s1Decl: lts.Transition()                                                          2
$s2Decl: lts.State()                                                               3

$s0: lts.State(this == $s0Decl, outgoingTransition contains $s1Decl)               4
$s1: lts.Transition(this == $s1Decl, source == $s0Decl, target == $s2Decl)         5
$s2: lts.State(this == $s2Decl, incomingTransition contains $s1Decl)               6
```

Fig. 5.4.: Join point detection rules generated for the LTS example shown in Fig. 5.2.

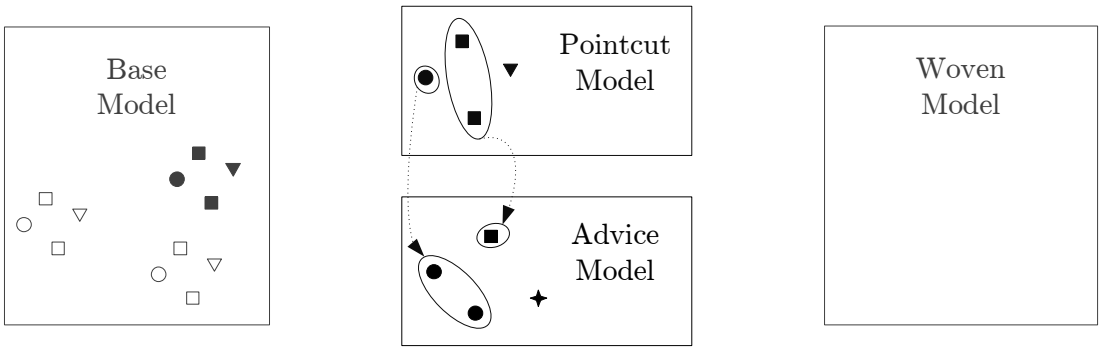


Fig. 5.5.: Symbolic representation of a mapping from pointcut elements to advice elements, which exhibits all four characteristic mapping situations.

to detect join points for models of arbitrary metamodels as they have no conceptual separation of simple attributes and general references to complex types. Therefore, our approach represents an improvement in automation and genericity to the process of join point detection for model weaving. We achieved a similar improvement in automation and genericity for the inference of correspondences between pointcut and advice elements, which we present in the following section.

5.2. Relating Pointcut and Advice Models

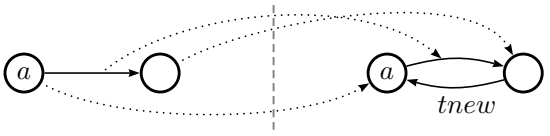
Correspondence-induced weaving operations

A unique feature of our model weaving approach is its ability to define weaving instructions using a declarative mapping from pointcut to advice elements. This many-to-many mapping defines how elements of the advice model represent elements of the pointcut model. Together with the join-point mapping from pointcut to base elements the mapping from pointcut to advice elements induces all weaving operations. These implicit weaving operations correspond to four characteristic mapping situations:

- Pointcut elements without a correspondence in the advice have to be *removed*.
- Advice elements without a correspondence in the pointcut have to be *added*.
- Pointcut elements corresponding to a single advice element have to be *merged*.
- Advice elements corresponding to a single pointcut element have to be *duplicated*.

In Fig. 5.5 we present a symbolic representation of the mapping from pointcut to advice elements for the weaving example that we already used in the previous section to illustrate the join-point mapping. It depicts all four characteristic mapping situations in the overall context of all models that are involved in the weaving process. Let us briefly explain the four mappings shown in the example. The triangle has to be removed because it has no correspondence in the advice model. For the star the opposite is true: It has no correspondence in the pointcut model and therefore has to be added. Both rectangles of the pointcut are mapped to a single rectangle of the advice which induces a merge. The single circle of the pointcut has to be duplicated because it corresponds to two circles of the advice.

The mapping from pointcut to advice elements can be defined as additional input for the weaving process or it can be inferred automatically for pointcut and advice models with unambiguous correspondences. Combinations of these two ways are also possible: An inferred mapping can be completed by a user-defined mapping and the other way round. The most comfortable way is to use the automatic inference in all cases in



(a) pointcut model (b) advice model

Fig. 5.6.: An example of a pointcut and advice model with an unambiguous mapping from pointcut to advice elements, which can be automatically inferred.

which this is possible and to specify mappings only when this is necessary and only for the ambiguous elements. We will now provide insights into the inference procedure for pointcut-to-advice mappings.

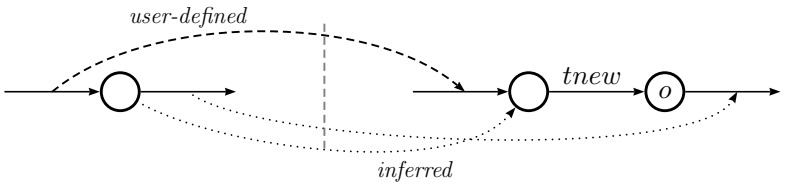
The default inference algorithm tries to map as many pointcut elements to advice elements on a best-effort basis using unique identifiers. Per default, these unique identifiers are calculated as concatenation of all values of string attributes. For every pointcut-model element this identifier is calculated in order to compare it to all identifiers of advice-model elements of the same type. If exactly one of these advice elements has the same identifier, a match is assumed and a mapping entry is inferred. If no or more multiple advice elements share the same identifier, no mapping is inferred for the pointcut-model element. Section 7.3.2 explains how this algorithm is realised in detail and Section 4.2.2 presents possible customisations through extensions.

Let us analyse the LTS example that we already presented in Section 3.2.3 and repeat in Fig. 5.6 in order to illustrate the mapping inference[2]. The purpose of this example is to detect transitions leaving the state a and to add new transitions in the opposite direction from the target to a. Its pointcut model matches a state a with a transition to a nameless state. The advice model repeats the detected pattern and contains an additional transition *tnew* from the nameless state to a. When an attempt for mapping the pointcut element a is performed, its identifier is calculated as the concatenation of the only string attribute: “a”. The only two advice-model elements sharing the same type have identifiers “a” and “”. Therefore, a mapping from the pointcut element a to the advice element a is inferred. Exactly the same process is repeated for the nameless states. Thus, another entry mapping these two states is inferred. The only remaining pointcut-model element is the transition from a to the nameless state. Its identifier is “” and the identifiers of the two transitions of the advice model are “” and “tnew”. Therefore, a third and last mapping entry for the transitions sharing the identifier “” is inferred. This demonstrates that our simple inference algorithm requires distinct identifiers for pointcut-model elements of a type that have to match the distinct identifiers for advice-model elements of the same type.

Completing user-defined mappings for ambiguous elements

To show how user-defined mapping entries can complete inferred mapping entries we provide another LTS example in Fig. 5.7. Its pointcut model matches every state with an incoming and an outgoing transition and the advice model introduces a state o dedicated to outgoing transitions. The two transitions of the pointcut model have the same string attributes, so that the same identifier “” is calculated for both. Therefore, our simple inference algorithm cannot deduce a complete mapping. Hence, the user has to define at least one mapping entry for a transition. In our example the user maps the incoming transition of the matched nameless state of the pointcut model to the incoming transition of the nameless state of the advice model. As only one transition and one state remain in the pointcut model, the rest of the mapping can be

[2]For simplicity, all LTS example figures depict pointcut-to-advice mappings as if they would map pointcut elements to advice elements and not sets of pointcut elements to sets of advice elements.



(a) pointcut model (b) advice model

Fig. 5.7.: The pointcut and advice model of an LTS example combining user-defined and automatically inferred mapping entries.

inferred automatically: The outgoing transition of the pointcut state is mapped to the outgoing transition of the advice model's new state *o* because they share the identifier "". Furthermore, the nameless state of the pointcut model is mapped to the nameless state of the advice model as they both share the identifier "" differing from the identifier "o" of the new state. This example demonstrates that a user only has to map pointcut elements for which more than one advice element is eligible as the remaining mapping entries can be inferred afterwards. Results that can be obtained when this aspect is woven into a base model are shown in Fig. A.1 and A.2 in the appendix.

Independent of the issue of automatic and user-defined mapping between pointcut and advice elements, the mapping itself is a key characteristic of our weaving approach that distinguishes it from other approaches. SmartAdapters (see Section 11.1.5), for example, relies on an imperative composition protocol, which specifies *how* an aspect has to be woven. MATA (see Section 11.1.7) works with stereotype values, which are used to declare, for example, elements that have to be added or removed. These stereotypes seem to be more declarative but they also specify *how* elements have to be woven. Aspects defined using our approach, however, neglect how weaving has to be carried out. Due to the mapping from pointcut to advice elements, it suffices to specify *what* elements are to be present before and after weaving. The correspondences imply the needed weaving operations, so that these do not need to be defined explicitly. This is a potential advantage in situations in which a user is able to express how a model should look like after the weaving but cannot or does not want to specify the necessary weaving and composition operations. How the actual weaving in the sense of model compositions is carried out in our approach is presented in the following section.

5.3. Model Composition

This section presents details of the model composition phase, which is central to our generic weaving approach. First, we provide a short description of the formalisation upon which all composition operations are built. Second, we discuss strategies for advice (re-)instantiation. Third, we present two composition scenarios that require duplication and merge operations. Before we go into details of the composition phase we complete our symbolic example for detecting join points and mapping pointcut to advice elements by providing the result of the composition phase. Fig. 5.8 depicts a symbolic representation of the woven model of this example and presents the correspondences between elements of the woven model and advice-model elements. Recall that all four characteristic mapping situations between pointcut and advice model are present. This induces the four displayed essential weaving operations: removal (triangle), addition (star), merge (rectangles), and duplication (circles).

A set-theoretic formalisation as operational basis

5.3.1. Composition Formalisation

This section introduces the essential concepts of a set-theoretic formalisation of our approach. The input to our weaving algorithm is a set B of base-model elements, a set

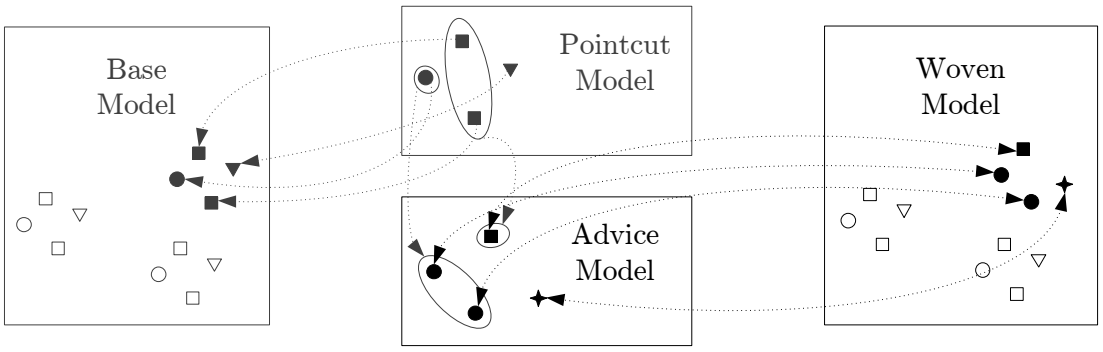


Fig. 5.8.: Symbolic composition representation induced by all models and mappings: removal (triangle), addition (star), merge (rectangles), and duplication (circles).

P of pointcut-model elements, and a set A of advice-model elements. From these inputs two mappings are calculated as intermediate results of the two phases of the previous sections (Section 5.1 and 5.2): A join-point mapping from pointcut to base elements and a pointcut-to-advice mapping:

Definition 5 (Join-Point Mapping) *Given a set B of base-model elements and a set P of pointcut-model elements we define the join-point mapping $j : P \to B$ as the injective function that maps every pointcut element to the base model element matching it at a join point.*

Definition 6 (Pointcut-to-Advice Mapping) *Given a set P of pointcut-model elements and a set A of advice-model elements we define the pointcut-to-advice mapping $m : 2^P \to 2^A$ as the partial function that maps pointcut elements to advice elements representing the same entities with 2^P and 2^A denoting the power sets of P and A.*

These intermediate mappings and the input models are used to calculate three sets and a bidirectional m-to-n mapping as formal foundation for the operations carried out during the weaving. Woven models are obtained through operations formulated for these sets and mappings. In Fig. 5.9 we present an exemplary visualisation of all sets and mappings of the formalisation. An initial description and a more complete description of the formalisation are provided by Morin et al. [MKBJ08] and Klein et al. [KKS+12b]. In the remainder of this section we discuss every set and mapping of our formal foundation individually and provide an example for the formalisation.

The first set of our formalisation contains all base-model elements that have to be removed during the weaving. These are all elements of the base model that correspond to an element of the pointcut model with no corresponding element in the advice model. We are convinced that this is intuitive for the user because the pointcut and advice models can be seen as pre- and post-conditions for the application of an advice: Every element that is explicitly mentioned in the pre-conditions but not mentioned in the post-conditions is considered an undesired element and has to be removed. The aspect models semantics and the resulting consequences have to be clear to users defining these aspects. To ensure that aspects do not remove elements unintentionally, implementations of our weaving approach should inform users when they define aspects that remove existing model elements.

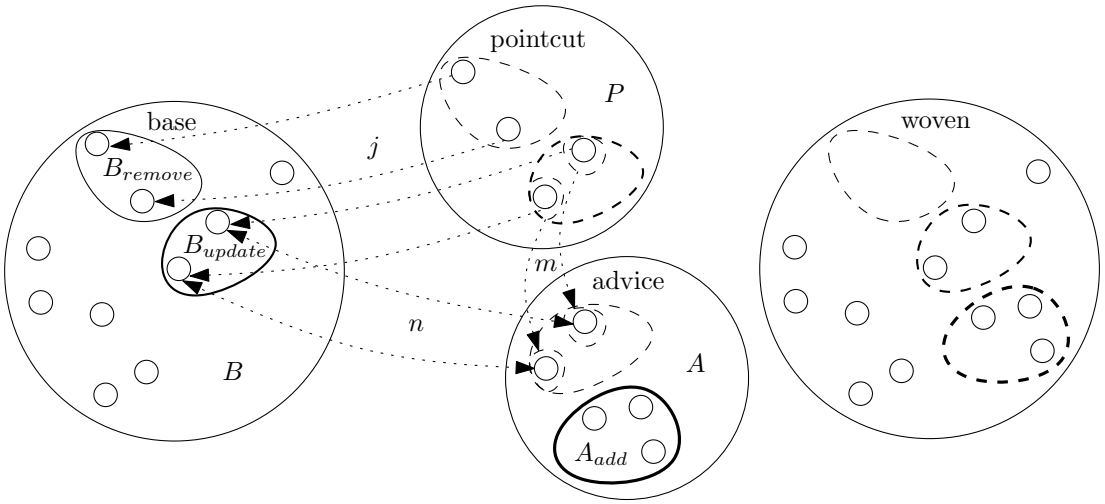


Fig. 5.9.: Formalisation visualisation showing involved models, sets and mappings.

Definition 7 (Base Elements to Remove) *Given a set B of base-model elements, a set P of pointcut-model elements, a set A of advice-model elements, a join-point mapping $j : P \to B$, and a pointcut-to-advice mapping $m : 2^P \to 2^A$, we define the set of base elements to remove:*

$$B_{remove} := \{b \in B \mid \exists p \in P : j(p) = b \wedge m(\{p\}) = \emptyset\}$$

We define a second set that contains all base-model elements that have to be updated during the weaving. These are all base elements that correspond to at least one pointcut element with a corresponding advice element. This is again in line with the user's intuition: Everything that appears in the pre- and post-conditions of an aspect application can potentially be modified because it is present prior to and after the weaving.

Definition 8 (Base Elements to Update) *Given the input sets B, P, and A and the mappings j and m as in Definition 7, we define the set of base elements to update:*

$$B_{update} := \{b \in B \mid \exists p \in P : j(p) = b \wedge m(\{p\}) \neq \emptyset\}$$

The third set of our formal foundation contains all advice-model elements that have to be added to the base model during the weaving. It is independent of a join point and contains all advice-model elements that correspond to no pointcut-model element. From the user's perspective this is straightforward: Everything that is not present in the pre-conditions of an aspect but occurs in the post-conditions has to be added.

Definition 9 (Advice Elements to Add) *Given a set P of pointcut-model elements, a set A of advice-model elements, and a pointcut-to-advice mapping $m : 2^P \to 2^A$, we define the set of advice elements to add:*

$$A_{add} := \{a \in A \mid \nexists p \in P : a \in m(\{p\})\}$$

The last element of the formal foundation for our weaving approach is a bidirectional m-to-n mapping that relates base-model elements with the corresponding advice-model

elements using the detected join-point mapping and pointcut-to-advice mapping. Once the three sets for elements that have to be removed, updated, and added are calculated the pointcut elements are irrelevant for the model composition. Therefore, it is sufficient and more convenient to work with a mapping that directly relates base and advice elements instead of working with two separate mappings that indirectly contain the same information via the intermediate elements of the pointcut.

Definition 10 (Base-to-Advice Mapping) *Given the input sets B, P, and A and the mappings j and m as in Definition 7, we define the bidirectional base-to-advice mapping n as the composition of the partial functions $n_{base-advice} : B \to A$ and $n_{advice-base} : A \to B$ defined as:*

$$n_{base-advice} : b \mapsto \{a \in A \mid \exists\, p \in P : j(p) = b \wedge a \in m(\{p\})\} \text{ and}$$

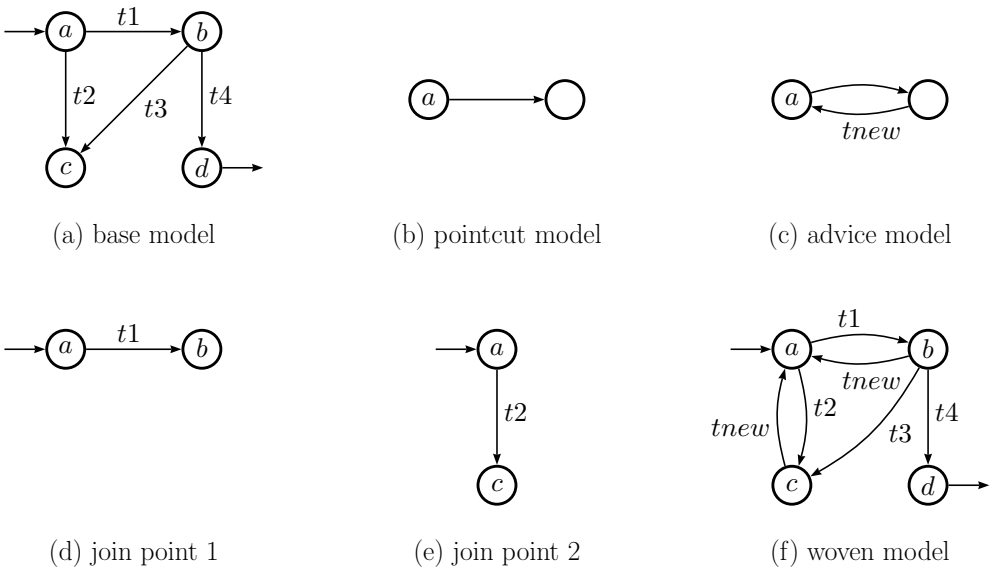
$$n_{advice-base} : a \mapsto \{b \in B \mid \exists\, p \in P : j(p) = b \wedge a \in m(\{p\})\}$$

Let us illustrate this formalisation using the running LTS example, which we already used to explain the detection of join points and the mapping of pointcut and advice models. Fig. 5.10 depicts the already discussed base, pointcut, and advice models together with the two corresponding join points and the resulting woven model. For the first join point (Fig. 5.10(d)) the three sets and the mapping formalising the weaving are as follows. The set of base elements to be removed is empty because all elements of the pointcut are mapped to advice elements: $B_{remove} = \emptyset$. The set of base elements to be updated contains all elements of the pointcut model as they all correspond to advice-model elements: $B_{update} = \{a, t1, b\}$. The set of advice elements to be added contains $tnew_a$ as this is the only advice element with no corresponding pointcut element: $A_{add} = \{tnew\}$. And the bidirectional mapping between base and advice elements $n_{base-advice}$ contains the entries $\{a\} \to \{a_a\}$ (where a_a denotes the state named a of the advice), $\{t1\} \to \{t_a\}$ (where t_a denotes the transition of the advice for which neither input nor output are specified), and $\{b\} \to \{s_a\}$ (where s_a denotes the nameless state of the advice). The entries of the opposite direction $n_{advice-base}$ are the same except for an inversion of domain and target. The sets and the mapping for the second join point (Fig. 5.10(e)) are identical except that every occurrence of b is replaced by c and every occurrence of $t1$ is replaced by $t2$. After the weaving process the woven result model contains the same elements as the base model plus two new transitions, which are triggered by the input $tnew$ and span from b to a and from c to a.

Exemplifying the formalisation

We will now explain how the set and mapping values for the first join point of our running example are established. The first join point can be expressed formally as a mapping $j : P \to B$ for $P = \{a_p, t_p, s_p\}$ such that $j(a_p) = a$, $j(t_p) = t1$, and $j(s_p) = b$. Similarly, the mapping from pointcut elements to advice elements can be formulated as $m : 2^P \to 2^A$ for $A = \{a_a, t_a, s_a, tnew_a\}$ such that $m(\{a_p\}) = \{a_a\}$, $m(\{t_p\}) = \{t_a\}$, and $m(\{s_p\}) = \{s_a\}$. Accordingly, B_{remove} is empty because for all $b \in B$ it holds that for all $p \in P$ such that $j(p) = b$ it is the case that $m(\{p\})$ is not empty. Furthermore, $B_{update} = \{a, t1, b\}$ and $n_{base-advice} = \{\{a\} \to \{a_a\}, \{t1\} \to \{t_a\}, \{b\} \to \{s_a\}\}$ because for $j(a_p) = a$ we have $m(\{a_p\}) = \{a_a\} \neq \emptyset$, for $j(t_p) = t1$ we have $m(\{t_p\}) = \{t_a\} \neq \emptyset$, and for $j(s_p) = b$ we have $m(\{s_p\}) = \{s_a\} \neq \emptyset$.

The formalisation is fundamental to our generic model weaving approach as it induces the individual weaving operations. It is based on a formalisation for a predecessor to our weaving approach, which was presented by Morin et al. [MKBJ08]. For our new version of the weaving approach we simplified the formalisation and increased its precision in terms of multiplicities. We excluded sets that induce no actions and factored out issues



(a) base model (b) pointcut model (c) advice model

(d) join point 1 (e) join point 2 (f) woven model

Fig. 5.10.: The base, pointcut, advice, join points, and woven result of our running LTS example.

relating to conditions in which advice elements are newly instantiated or reused by other join points. These advice instantiation strategies are discussed in the next section.

5.3.2. Advice Instantiation

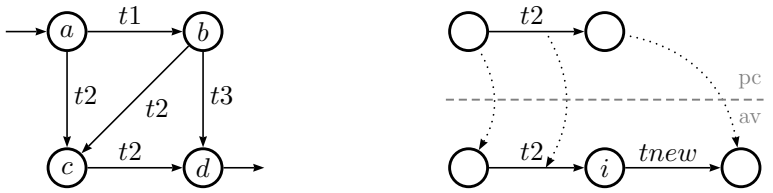
Three types of advice creation: global, per join point, and custom

In our approach we distinguish different strategies for advice instantiation as presented by Morin et al. [MKKJ10]. These instantiation strategies are similar to the way AOP approaches handle advice instantiation on a global, per-object, or per-control-flow basis. Whether or not an element of the advice model is newly instantiated during the weaving can be determined globally, per join point, or in a custom way for some of the elements of a join point. This model region for which an advice-model element is instantiated exactly once is called *advice instantiation scope*. We explain each instantiation scope individually and provide an example illustrating all different types.

If an element of an advice model that has to be added (i.e. a member of A_{add}) has a global advice instantiation scope, it is created once during the weaving so that all join points use this single instantiation. This scope can be used to share information across various join points or to avoid unnecessary system growth. For example, an aspect that is responsible for counting specific actions has to introduce a globally shared counter.

An advice-model element in A_{add} with a per-join-point advice instantiation scope is newly created for every detected join point. This scope is used as default when no scope is explicitly specified. It is possible to specify advice models containing elements with a global advice instantiation scope as well as elements having a per-join-point scope. References between those elements are also possible, but it is not possible to refer to per-join-point instantiations belonging to another than the current join point.

If an advice-model element in A_{add} has a custom advice instantiation scope it is newly created whenever a join point yields a new combination of the elements in the scope. This makes it possible to control advice instantiation using a comparatively fine level of granularity. An exemplary use of this scope would be an aspect that requires separate instantiations of an element for every individual subsystem. Another possibility is given in the following example that illustrates all different types of advice instantiation scope.

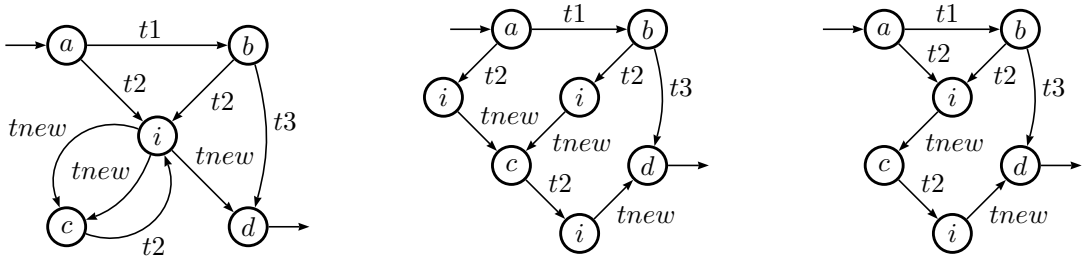


(a) base model (b) pointcut and advice model

Fig. 5.11.: The base, pointcut, and advice model for an advice instantiation example.

In order to demonstrate the effects of the three different types of advice instantiation scopes we provide an LTS example in Fig. 5.11 and 5.12. Fig. 5.11(a) shows the base model with four states linked by five transitions and Fig. 5.11(b) displays the pointcut and advice model introducing a new intermediate state i for every transition triggered by $t2$. The intermediate state i serves as new target for the matched transition. It has a new transition triggered by *tnew*, which targets the old target of the transition matched by the pointcut. As the intermediate state and its transition are both members of A_{add} the woven result differs depending on the chosen advice instantiation scopes for these elements. Fig. 5.12(a) shows the woven model that is obtained when a global advice instantiation scope is specified for i and the default per-join-point scope of its transition is left unchanged. It contains a single new state i, which serves as target for all three transitions triggered by $t2$. These transitions belong to the three join points involving a and c, b and c, and c and d. The three new transitions triggered by *tnew* target the former target of the matched transitions c and d. A different result is obtained when the default per-join-point advice instantiation scope is also chosen for the intermediate state i (Fig. 5.12(b)). For every matched transition triggered by $t2$ a new intermediate state i is introduced as new target. The result obtained when a custom advice instantiation scope is used for the intermediate state i and its transition triggered by *tnew* is shown in Fig. 5.12(c). When the transition that was matched in the pointcut targets a state that was not encountered as target before, a new intermediate state i is introduced. As a result, one instance of i is created for the two join points with transitions targeting c and another instance is created for the join point with the transition targeting d. Another weaving example with different advice instantiation strategies is provided in Fig. A.1 and A.2 in the appendix.

Our support for different advice instantiation scopes was inspired by the strategies proposed by Morin et al. [MKKJ10]. In contrast to their integration of instantiation strategies into the join point detection mechanism of SmartAdapters, we separated



(a) global scope (b) per-join-point scope (c) custom scope

Fig. 5.12.: Woven models for different advice instantiation scopes for Fig. 5.11.

these weaving steps. We do not need to share any information between join points during detection as we delegate instantiation to appropriate copiers that reuse existing instantiations whenever the advice instantiation scope specifies this (see Section 7.3.3). This separation makes it possible to have an extensible advice instantiation mechanism for which new advice instantiation strategies can be provided using extensions to the corresponding copier join point (see Section 4.2.2).

The set-theoretic formalisation and different advice instantiation scopes are two main elements of the compositional weaving phase of our approach. Another issue that has an important influence on the composition phase are the different ways in which pointcut and advice elements are mapped. Therefore, two exemplary ways to map aspect elements are discussed in the following section.

5.3.3. Composition Scenarios

One-to-many: duplication, many-to-one: merge

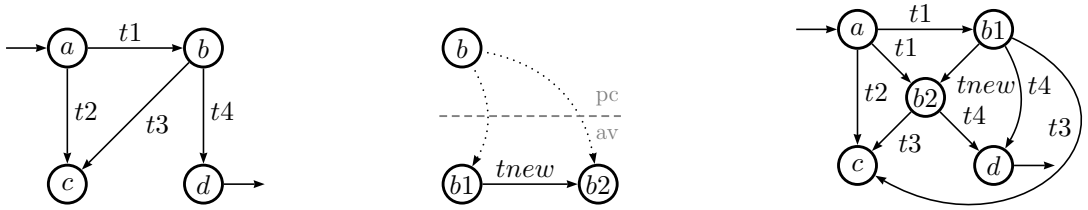
Based on the explanation of the mapping mechanism for pointcut and advice elements in Section 5.2, we present two composition scenarios in detail in order to complete the general discussion of the composition phase. These composition scenarios are induced by mappings for pointcut and advice elements that relate a single element to multiple elements or multiple elements to a single element.

5.3.3.1. Duplication

The first weaving scenario we present in detail involves the duplication of an element of the base model. Such a duplication operation is needed whenever an element of the pointcut model corresponds to more than one element of the advice model. For each join point the consequence for the model elements involved in the duplication-inducing mapping is as follows: All base elements representing involved advice elements have to exhibit all properties of the base element that matches the pointcut element of the duplication. This is achieved by introducing the attribute and reference values of the base element that matches the pointcut element into the base elements that correspond to the advice elements.

Fig. 5.13 illustrates such a duplication scenario with example models of an LTS. It depicts the base model that we also used to explain previous weaving phases and the formalisation. The aspect shown in Fig. 5.13(b) matches every state named b and replaces it with two states $b1$ and $b2$, which are linked by a transition triggered by $t2$. This replacement of one matched element with two new elements is defined by mapping the pointcut state b to the two advice states $b1, b2$ ($m(\{b\}) = \{b1, b2\}$). As the base model contains only one state named b we obtain a single join point mapping the pointcut element b to the base element b. In terms of the formalisation presented in Section 5.3.1 this means $B_{update} = \{b\}$, $n_{base-advice} = \{\{b\} \rightarrow \{b_1, b_2\}\}$, and $A_{add} = \{tnew\}$. The woven result obtained from weaving the base, pointcut, and advice model with this duplicating mapping is presented in Fig. 5.13(c). It shows that the new states $b1$ and $b2$ exhibit all properties of b. More precisely, b's incoming transition $t1$ and its outgoing transitions $t3$ and $t4$ are duplicated and added to the new states $b1$ and $b2$. The fact that the transition $tnew$ from $b1$ to $b2$ is also introduced during the weaving is independent of this duplication operation and a result of $tnew \in A_{add}$.

In order to demonstrate that such a duplication scenario occurs in real-world settings and to illustrate that the induced operations are independent of the metamodel we provide a duplication example for IFC building models. The purpose of this duplication aspect is to double cable ports. UML class diagrams of its pointcut and advice model are depicted in Fig. 5.14. The pointcut model presented in Fig. 5.14(a) shows that a cable



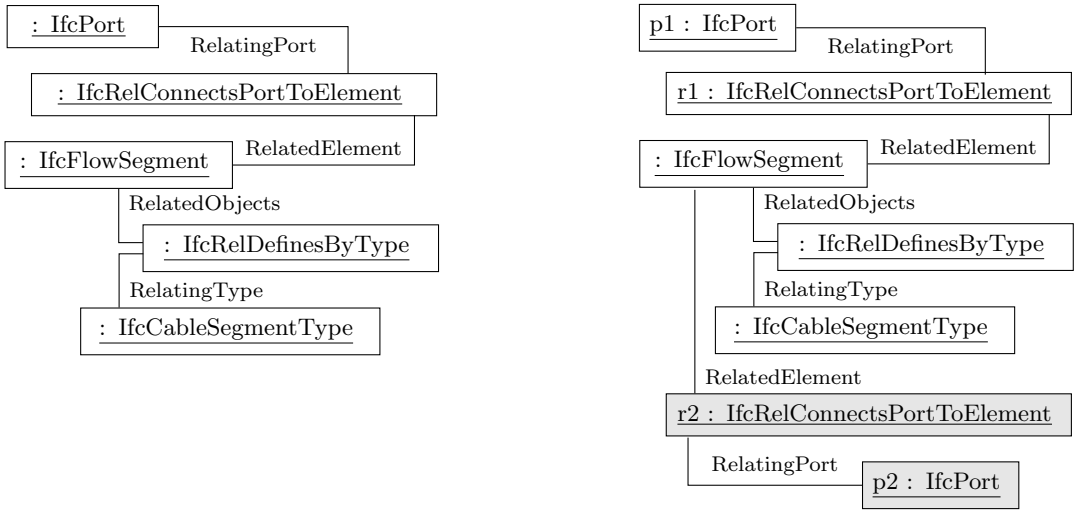
(a) base model (b) pointcut model advice model (c) woven model

Fig. 5.13.: Weaving an aspect into an LTS while duplicating the base element b.

port is represented as an `IfcPort` in the BIM format IFC (see Section 2.3.1). As such an `IfcPort` can be used to connect other building elements than cables, the pointcut model contains further restrictions. It specifies that the port is related to an `IfcFlowSegment` via an `IfcRelConnectsPortToElement`. Furthermore, the `IfcFlowSegment` is typed using an `IfcCableSegmentType` via an `IfcRelDefinesByType`. To achieve a duplication of such cable ports, the advice model presented in Fig. 5.14(b) contains the same elements as the pointcut model and an additional `IfcPort`. This port is connected to the same `IfcFlowSegment` using an additional `IfcRelConnectsPortToElement` relation. The mapping from pointcut to advice elements, which was omitted in the figure, relates the single `IfcPort` of the pointcut to both `IfcPorts` of the advice and the single `IfcRelConnectsPortToElement` to both instances of the advice. All other pointcut and advice elements have a one-to-one correspondence. As a result, all properties of the detected port and its relation are duplicated during the weaving of this aspect. We cannot provide a base or a woven example as this would require too much space.

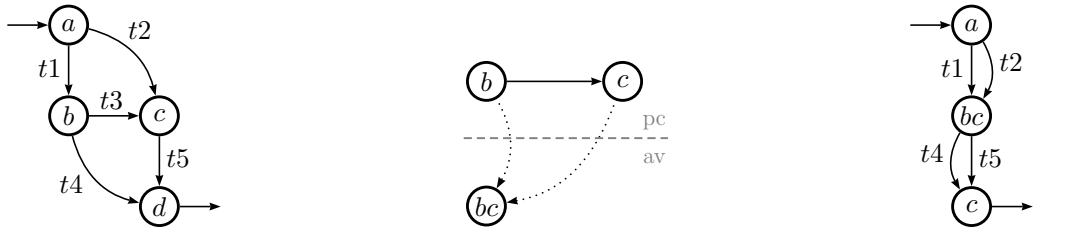
5.3.3.2. Merge

A composition scenario that can be seen as dual to duplication occurs when more than one element of the pointcut model corresponds to a single element of the advice



(a) pointcut model (b) advice model

Fig. 5.14.: An example aspect for IFC building models which duplicates cable ports and depicts the difference between the pointcut and the advice model in grey.



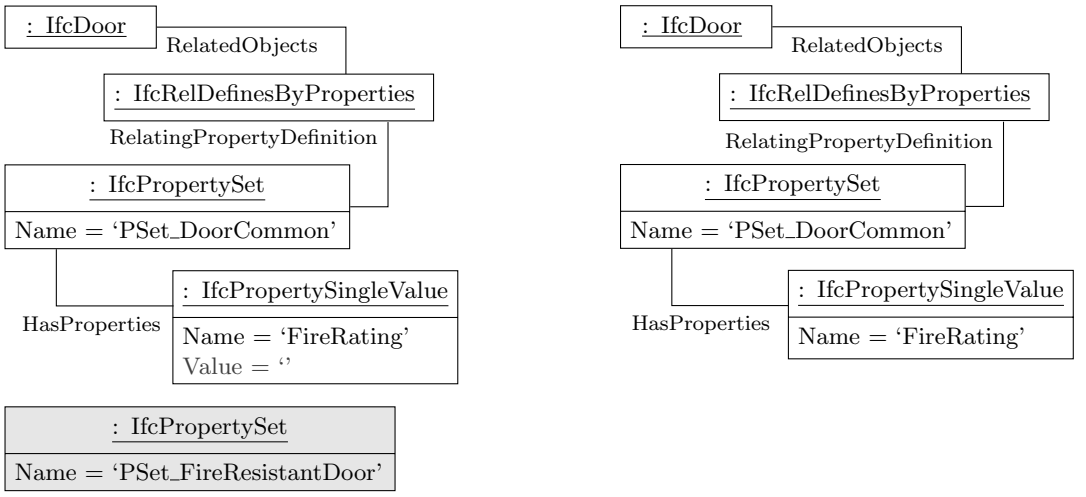
(a) base model (b) pointcut model advice model (c) woven model

Fig. 5.15.: Weaving an aspect into an LTS while merging the base elements b and c.

model. The resulting merge operation has to make sure that the involved advice element exhibits all properties of all the involved pointcut elements. To realise this, all attribute and reference values of the base elements matching the pointcut elements are introduced into the base element corresponding to the advice element.

We provide example models of an LTS to illustrate the merge scenario in Fig. 5.15. Its aspect shown in Fig. 5.15(b) matches every transition between two states named b and c and replaces them with a new state bc. This replacement of two matched pointcut elements with one merged advice element is induced by a mapping from the pointcut state b and c to the advice state bc ($m(\{b, c\}) = \{bc\}$). The only possible join point maps these pointcut elements b and c to the elements with the same names in the base model and the pointcut transition to the base transition triggered by $t3$. As a result, the mapping from base-model elements to advice-model elements relates the base elements b and c with the advice element bc: $n_{base-advice}(\{b\}) = \{bc\} = n_{base-advice}(\{c\})$. Additionally, our formalisation yields $B_{remove} = \{t3\}$ and $B_{update} = \{b, c\}$. The woven result obtained from weaving the base, pointcut, and advice model with this merging mapping is shown in Fig. 5.15(c). It indicates that the merged state bc exhibits all properties of the two states b and c. More precisely, b's incoming transition $t1$ and c's incoming transition $t2$ are merged into the woven model's resulting element bc. The same applies for b's outgoing transition $t4$ and c's outgoing transition $t5$. Independent of this merge another weaving operation has to be carried out: The transition from b to c has to be removed because the join point matches it to the transition of the pointcut model that has no correspondence in the advice model ($t3 \in B_{remove}$). Another merge example for an LTS is provided in Fig. A.3 in the appendix.

A similar merge scenario for IFC building models is shown in Fig. 5.16. The aspect of this example is used in order to ensure that every door with an unspecified fire rating obtains the properties of a fire resistant door. To achieve this, the pointcut model shown in Fig. 5.16(a) matches an `IfcPropertySet` that defines an `IfcDoor` using an `IfcRelDefinesByProperties`. The matched set of properties for a door is further restricted: It has to contain an `IfcPropertySingleValue` named "FireRating" with an unspecified (i.e. empty) value. The pointcut model also contains another `IfcPropertySet` named "PSet_FireResistantDoor", which will be used to apply fire resistance properties to matched doors. This application of properties is realised by merging the matched property set containing an unspecified fire rating value with the property set for fire resistant doors. In order to induce this merge the advice model contains the same elements as the pointcut model except for the property set for fire resistant doors and maps them accordingly (not shown in the figure). The reappearing pointcut elements are mapped to the directly corresponding advice elements and the fire resistance property set is mapped to the property set with the unspecified fire rating value. As in the



(a) pointcut model (b) advice model

Fig. 5.16.: An example aspect for IFC building models which merges properties and depicts the difference between the pointcut and the advice model in grey.

LTS merge scenario two pointcut elements (both matched property sets) are mapped to a single advice element (the property set that should remain). The effect is that all property values of both matched sets are present in the single property set of a woven model. A similar effect could have been achieved by matching only the underspecified door and listing all desired property values directly in the advice model. However, this solution would have been more verbose and bound to the concrete property values (e.g. fire rating = "AS 1905.1" or smoke stop = true).

The presented composition scenarios for duplication and merge close our discussion of the compositional part of our weaving approach. In the following section we present the remaining weaving phase, which removes elements and cleans-up elements that are inconsistent with respect to the metamodel.

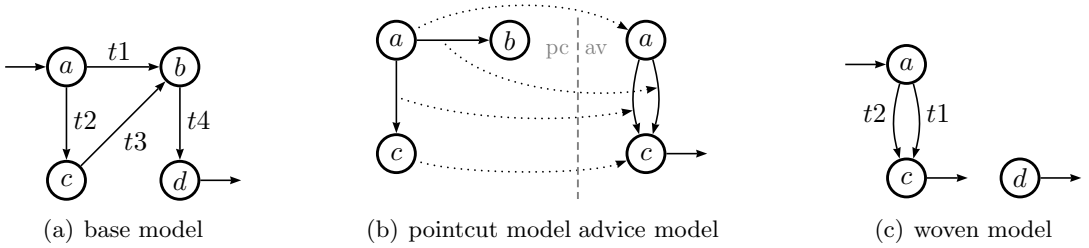
5.4. Removing Elements while Ensuring Metamodel Compliance

Iteratively removing inconsitent elements

When all model elements have been composed a last weaving phase removes model elements and ensures compliance to the metamodel by cleaning-up inconsistent model elements. As explained in Section 5.3.1, an element of the base model has to be removed during the weaving at a join point if this join point maps a pointcut element that has no correspondence in the advice model to the base element. Removing these unmatched elements may result in base-model elements that violate lower bound constraints of the metamodel because they cannot reference removed elements anymore. Therefore, we have to detect these inconsistent elements and remove them too. As this removal of inconsistent elements can produce new inconsistent elements, we have to continue this clean-up process until a point is reached at which all constraints are satisfied. This could theoretically lead to a clean-up that deletes all elements if no constraint-satisfying condition is reached before. But we are confident, that such a clean-up does not become necessary when aspects are designed with care using an interactive tool that issues a warning in cases in which an aspect has to lead to inconsistent conditions.

To illustrate this final removal and clean-up phase we provide an LTS example of a removal scenario in Fig. 5.17. Its aspect matches situations in which a state a has a

transition to a state b and a transition to a state c. The purpose of the aspect is to remove the matched state b while redirecting the transition that went to the state b to the kept state c (Fig. 5.17(b)). This is achieved by mapping the pointcut-model element b to no advice-model element and the transition from a to b to a new transition from a to c. In terms of the formalisation presented in Section 5.3.1 the only possible join point involves the base-model elements a, b, c, $t1$, and $t2$ and yields $B_{remove} = \{b\}$ and $B_{update} = \{a, c, t1, t2\}$. When b is removed from the woven model the transition $t3$, which originally went from c to b, violates the lower-bound constraint for its target attribute. It refers to no element instead of exactly one element as required by the LTS metamodel (see Fig. 5.3). The same applies to the source attribute of the transition $t4$, which originally went from b to d. Therefore, both transitions $t3$ and $t4$ are removed during the clean-up phase of the weaving. No element refers to the state d, but it is not removed during the clean up as it does not violate any constraint of the metamodel. The change of the target attribute for the transition $t1$ and the setting of the final attribute for c are independent of the removal and clean-up operations. In contrast to the scenarios for duplication and merge we cannot provide a realistic removal and clean-up example for IFC building models as building specifications do not specify removals. Another LTS example with removal is provided in Fig. A.4 in the appendix.



(a) base model (b) pointcut model advice model (c) woven model

Fig. 5.17.: Weaving an aspect for a small LTS while removing the base element b.

5.5. Summary

In this chapter we discussed the individual weaving phases of our approach to generic model weaving in detail using LTS and IFC examples. We explained how join points are detected and that they can be formally expressed as a mapping from pointcut to base elements. Then, we presented the weaving inducing mapping from pointcut to advice elements and explained its four characteristic mapping situations for removal, addition, merge, and duplication. For the model composition phase, we presented the set-theoretic formalisation, discussed advice instantiation strategies, and exemplified the merge and duplication of model elements. The presentation of the last weaving phase, which removes elements and cleans up inconsistent references, concluded the chapter.

After presenting our weaving approach in a conceptual way, the next part of the thesis provides insights into our weaver implementation. We start with a description of the architecture and environment of our implementation in the next chapter, before we continue with a detailed discussion of our solutions.

Part III.

A Practical Model Weaver Implementation

6. Architectural Overview

In this chapter we introduce the implementation of our generic and extensible model weaver. Section 6.1 describes which environment and libraries we chose and explains the corresponding rationale. Section 6.2 outlines the structure of our implementation and presents the individual plug-ins with their interdependencies.

6.1. Environment & Libraries

The implementation of our model weaver benefits from the runtime environment that it is deployed in and from external libraries. Both are presented in the following two sections in order to discuss the underlying design decisions and resulting consequences.

6.1.1. Environment

We implemented our generic and extensible model weaver as a set of plug-ins for the IDE Eclipse. These plug-ins are based on the Eclipse Modelling Framework (EMF) and were written in the *Java* programming language. In the following the most important features of this environment and the rationale for choosing it are explained.

Fundamental design decisions

Eclipse's extension possibilities based on plug-ins and explicit extension points are the foundation for the implementation of the extensibility and genericity of our weaving approach. Each plug-in for Eclipse is automatically a bundle of the modular component-based Open Services Gateway initiative (OSGi)[1] framework. We use this OSGi base to start new Eclipse plug-ins at runtime in order to execute code that was generated for user-chosen metamodels. Furthermore, Eclipse plug-ins can declare custom extension points and execute the code of extensions that registered for these extension points. This feature is heavily used throughout our implementation in order to allow for domain-specific customisations of the generic weaving process.

EMF provides an infrastructure for typical tasks of MDE and includes the wide-spread metamodelling language Ecore, which gives us the possibility to support a variety of models. As various MDE tools and model transformation engines are based on EMF this ensures a good interoperability of our implementation. It also allows us to concentrate on the key weaving functionality as the EMF API provides standard functionality, e.g. for serialising and storing models. The fact that many important modelling notations,

[1] OSGi - The Dynamic Module System For Java: osgi.org

such as the OMG standard Unified Modeling Language (UML), can be represented using Ecore makes our implementation accessible.

We decided to implement our weaver using Java in order to give as many researchers and developers as possible the facility to extend or modify our implementation without any technological breaks. Java is wide-spread in the Eclipse and EMF community and these platforms are also implemented using Java. This allows for straight debugging of our implementation without any conversions to intermediate representations or similar barriers. Other languages based on the Java Virtual Machine (JVM), such as Kermeta or *Scala*[2], may have resulted in more concise code but would have reduced the number of possible adopters.

As we explain in Section 12.2.2 a possible field for future work could be the transition to the programming language *Xtend*[3]. It is an extension to the Java programming language and requires less code repetition through features like type inference while compiling to ordinary Java code. Xtend could increase the code quality for our implementation, for example, with its functional programming constructs in areas where intensive collection manipulation is done. We could also use *Xtext*[4], the framework for defining extensible DSLs upon which Xtend is built, to define a small language for accessing the numerous extension points and extensions of our implementation. Both solutions would still satisfy our need for direct Java debugging and reusability as Xtend and DSLs created with Xtext can be compiled to plain Java code.

6.1.2. Libraries

In addition to the internal libraries of EMF our implementation uses only one external library: the business logic integration platform *Drools*[5] which realises the join point detection phase.

One of the possibilities to use Drools is to formulate rules for a collection of Plain Old Java Objects (POJOs) in order to obtain the objects that satisfy the restrictions of the rules. In contrast to engines for logical programming languages, such as Prolog, Drools uses the forward-chaining Rete algorithm [For82] instead of a backward-chaining solution starting with the goals. This data-driven inference algorithm provides improved performance compared to backward-chaining systems. We use Drools query possibilities to execute join point detection rules on a knowledge base that contains all elements of the base model. How we obtain these rules from visiting a pointcut model is explained in Section 7.3.1. In Section 11.1.5 we discuss the SmartAdapters weaving approach of Morin et al. [MKKJ10], which uses similar Drools rules. There we also explain that our approach differs in that we do not generate rules for advice instantiation in order to allow for separate customisation or evolution of these weaving phases.

Our implementation does not require any other external libraries in addition to Drools for two reasons: First, it does not need complex data structures. Second, it uses the rich APIs of the underlying EMF and Eclipse platform. The only non-standard data structures that we use in our weaver implementation is a bidirectional m-to-n mapping, which we presented in Section 7.4. We created our own custom implementation as existing libraries, such as Apache Commons Collections[6] or Google's Guava[7], do not provide an implementation that can directly be used for our purposes.

[2]Scala - A scalable general-purpose programming language: scala-lang.org
[3]Xtend - An extension to the Java programming language: eclipse.org/Xtext/xtend
[4]Xtext - A framework for creating extensible DSLs: eclipse.org/Xtext
[5]Drools - A Business Logic integration Platform: jboss.org/drools
[6]Commons-Collections - Augments the Java Collections Framework: commons.apache.org/collections
[7]Guava - Google Core Libraries for Java: code.google.com/p/guava-libraries

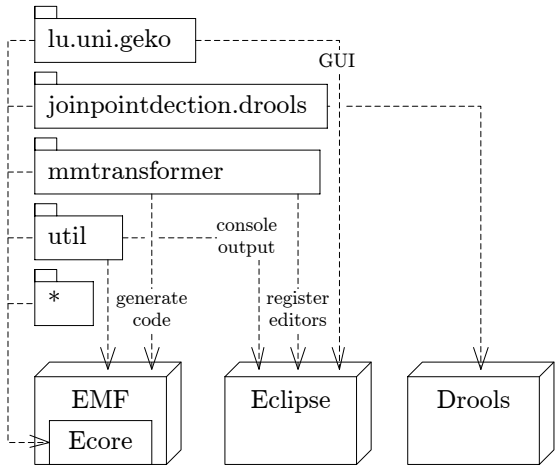


Fig. 6.1.: Visualisation of the environment and libraries of our implementation.

External dependencies

Fig. 6.1 displays the frameworks and libraries used in our implementation together with the plug-ins depending on them. EMF's metametamodelling language Ecore also provides the implementation classes and interfaces for models conforming to metamodels that are specified using Ecore. Therefore, all plug-ins of our weaver implementation depend on the Ecore related part of EMF as depicted in the figure. The remaining dependencies are as follows: Our main plug-in `lu.uni.geko` indirectly depends on Eclipse to provide the Graphical User Interface (GUI) for our tool. All other plug-ins of our implementation are prefixed with this specifier, which we omit in the rest of this thesis. The default plug-in for join point detection `joinpointdetection.drools` depends on Drools. EMF functionality apart from the main Ecore related API is used in the `mmtransformer` plug-in in order to generate infrastructure code for relaxed metamodel variants and corresponding editors. It also depends on Eclipse as it registers the generated metamodels and editors for the GUI. Finally, the main utility plug-in of our weaver `util` also depends on advanced EMF functionality and Eclipse's GUI API for console output. These and all remaining plug-ins of our implementation are presented in detail in the following section.

6.2. Interacting Plug-Ins

Internal dependencies

We implemented our generic model weaver as a set of Eclipse plug-ins that interact with each other through explicit interfaces. These plug-ins and their structural interdependencies are illustrated in Fig. 6.2. Note that the depicted interdependencies are the result of calls to comparatively small subroutines and do not reflect the overall control flow, which is discussed in Section 3.2. The main plug-in `lu.uni.geko` realises the facade design pattern by Gamma et al. [GHJV95] in order to provide access to user-related functionality of all other plug-ins at a unique location via commands. Therefore, it depends on all remaining plug-ins. All these plug-ins depend on the plug-in `util` as it centralises all project-independent utility functionality and structure. Except for this general plug-in all other plug-ins depend on `common`, which encapsulates all functionality and data structures used by multiple plug-ins but unlikely to be applicable in other projects. The last of these plug-ins that are required by numerous others is `resources`. It provides convenient access to model resources of all kinds to all plug-ins except for the two previously mentioned project-specific and project-independent utility plug-ins.

Let us shortly present the remaining six out of the ten plug-ins of our implementation. We already mentioned in the previous section that the `mmtransformer` plug-in

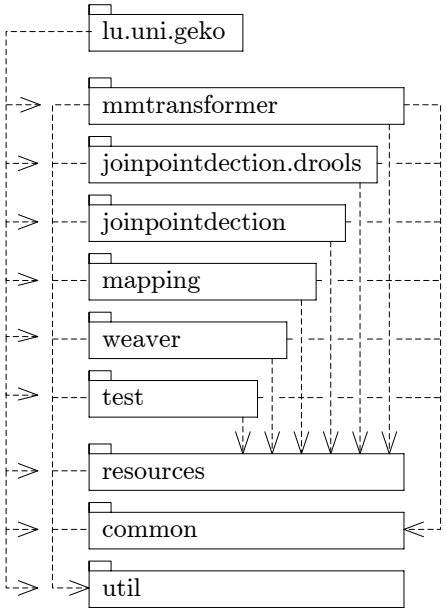


Fig. 6.2.: The plug-ins implementing our weaver and their structural inter-dependencies.

generates metamodel variants and corresponding code whereas the plug-in `joinpoint-detection.drools` detects join points using the Drools libraries. The general plug-in `joinpointdetection` defines an interface for join point detection mechanisms in order to abstract from the individual realisation and therefore it does not provide own functionality. Code for loading, resolving, and completing mappings that relate pointcut and advice models is located in the plug-in `mapping`. The main weaving logic for merging, duplicating, and removing model elements as well as advice instantiation code is bundled in the plug-in `weaver`. Last, the plug-in `test` facilitates testing of customised functionality for extension developers by providing a unit test skeleton. This testing infrastructure can be used to compare weaving results with woven archetype models, which serve as examples of correct weaving.

The dependence relationships between the plug-ins of our implementation are the result of a compromise between test-driven programming and the attempt to achieve good extensibility. Apart from dependencies to the util, common, and resources plug-ins the individual plug-ins and weaving phases are independent of each other. The rationale for this strict separation is that we wanted to ease the replacement of steps or whole phases of the weaving process. Furthermore, we had to develop a rapid prototype that handles our test weaving scenarios but provides no functionality that is not yet used. As a compromise, we decided to separate our implementation into several individual plug-ins while not providing any extension points that are neither used by the generic core nor by the extension for building models.

6.3. Summary

In this chapter we presented the overall structure and environment of our model weaver prototype. We explained the main design decisions together with the corresponding alternatives, consequences, and rationale. After a description of the interaction with the Eclipse environment and with used libraries, we presented the internal structure of the plug-ins realising our weaver. Detailed information on implementation solutions is provided in the following chapter.

7. Detailed Implementation Discussion

In this chapter we present detailed solutions for the realisation of our generic model weaver along used patterns, algorithms, and data structures. Section 7.1 explains know design patterns and custom solution patterns that we applied throughout our implementation. Barriers that had to be overcome while implementing our weaver are described in Section 7.2. Selected algorithms are detailed in Section 7.3 using pseudo-code. Finally, custom data structures that we developed for our weaver are presented in Section 7.4.

7.1. Solution Patterns

In order to ease maintenance and extension tasks, we used renowned design patterns and developed custom solution patterns for problems that recurred during the implementation of our model weaving approach.

7.1.1. Applying Known Design Patterns

We applied three structural, a behavioural, and a creational design pattern presented by Gamma et al. [GHJV95]: facade, bridge, object adapter, visitor, and singleton.

The structural *facade* design pattern is used in the plug-in `lu.uni.geko` for the class `ActionsFacade` in order to provide a unified interface to all functionality available to users. To sustain this design, we formulated the rule that individual plug-ins should not access functionality of other plug-ins directly if the functionality is also accessible for end users of the weaver. In such cases a new method should be added to `ActionsFacade` and this method should be called by commands of the GUI and depending plug-ins.

Decoupling library implementations

In order to decouple the implementation of used APIs from their abstract functionality we employ the structural *bridge* design pattern. In the plug-in `lu.uni.geko.util` eight of these bridges encapsulate functionality of Eclipse, EMF, and Java. As a general rule we created a new method in a bridge whenever a characteristic sequence of calls to an API occurred multiple times. This reduces the number of points at which our implementation has to rely on external libraries, which should reduce maintenance effort in case of library updates or replacements. The application of this design pattern should also reduce the effort needed to implement extensions for our weaver as it reduces the amount of knowledge that is necessary to use these APIs by providing own methods for frequent tasks. We tried to increase this effect by separating functionality based on a single library into several bridges according to a more fine-grained task classification.

For EMF, for example, we provide four different bridges that encapsulate general functionality of Ecore-models, resource-related functionality, creational factory methods, and model-independent functionality, such as file-path conversions.

In order to be able to treat simple extensions like their sophisticated counterparts we used the structural *object adapter* design pattern, which is also known as *wrapper*. As presented in Section 4.2.2, we provide two different possibilities to extend each of the tasks of loading models and adding new model elements to woven models. When the corresponding code is called no distinction between the simple and detailed variants has to be made as simple extensions are automatically wrapped into an object adapter that adds default behaviour.

The behavioural *visitor* design pattern is realised in the Drools variant of the join point detection plug-in in order to derive rules for the Drools engine from a pointcut model. In a first pass, all elements of the pointcut model are visited in order to declare a variable with according attributes for each element. In a second pass, all references of the pointcut model elements are resolved. They are added as separate variables with corresponding references to the variables that were generated during the first pass. As the internal state of the pointcut model elements is available via public getter methods we were able to use a simplified version of the design pattern without explicit accept methods.

Last, we had to use the creational *singleton* pattern at various places of our implementation in order to restrict the instantiation of classes to a single object. This was particularly important in the context of resource loading in order to ensure that different weaving tasks do not have access to different instances of the same model resource.

7.1.2. Developing Custom Solution Patterns

Where it was not possible or appropriate to implement our weaver using well-known design patterns we organised our implementation with a regularity that emphasises the commonalities of recurring problems and solutions. In this section we explain some of these recurring problems and present our solutions and the according design rationale.

7.1.2.1. Extension Facilities

The highest regularity in our implementation is exhibited by the infrastructure required to provide extension possibilities. The reason for this redundant regularity is that Eclipse requires a combination of declarative and imperative code for extension points and extensions. This makes it difficult to factor out structural commonalities. In order to ease maintenance of our implementation and to provide a guideline for new extensions, we employed a general solution pattern at every individual point of extension.

Rules for extension points

An extension point itself is always named like an acting subject and marked with a suffix in order to avoid name clashes with existing entities such as classes. Thus, all extension point names, such as `SimpleAdderExt` or `UIDCalculatorExt`, end with one of the suffixes `-erExt` or `-orExt`. This name is also used for the Java interface that every extension has to implement.

The XML schema definition of an extension point for our implementation is structurally identical for all extension points except for two points of variation. Fig. 7.1 presents a simplified template for these schemas. It shows that every extension of an extension point has to refer to at least one client element. This client element has to provide the fully qualified name of the class that implements the interface of the extension

```
<schema targetNamespace="?qualifiedPluginName?" xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="extension">
      <complexType>
         <choice minOccurs="1" maxOccurs="?upperBound?"> <!-- variation point 1/2 -->
            <element ref="client"/>
         </choice>
         <!-- [...] -->
      </complexType>
   </element>
   <element name="client">
      <complexType>
         <attribute name="class" type="string" use="required">
            <annotation>
               <appinfo>
                  <meta.attribute kind="java" basedOn=":?qualifiedExtensionPointName?"/>
               </appinfo>
            </annotation>
         </attribute>
         <attribute name="priority" type="string"/> <!-- variation point 2/2 -->
      </complexType>
   </element>
</schema>
```

Fig. 7.1.: A simplified template representation of the XML schema for extension points for our implementation showing two points of variation.

point. Whether more than one client can be provided in a single extension and whether extensions have to provide a priority depends on the individual extension point. So far all extension points of our weaver implementation that permitted more than one extension used the priority attribute in order to call registered extensions in order of decreasing priority.

Management of registered extensions and delegation of execution to the code of these extensions is always encapsulated in a utility class named like the extension point except for the prefix `Main`. These utility classes obtain the registered extensions, wrap simple extensions using the object adapter pattern as explained above, sort all extensions in order of decreasing priority, and pass arguments to calls to extension code. Although we tried to reduce possible code redundancy as far as possible by providing appropriate utility methods we are convinced that the best way to do this would be using a small DSL for extensions and extension points. In Section 12.2.2 we explain how a language that replaces declarative and imperative extension code could be used in the future to reduce redundancy.

7.1.2.2. Convenience Representations for Weaving Data

Wrapping central weaving concepts

Although we followed the principle of agile development that states that only strictly necessary code should be implemented[1] we decided to introduce convenience representations for highly used weaving data. This was done in order to increase the readability of our code and to facilitate later changes. The three convenience representation classes `JoinPoint`, `Advice`, and `AdviceEffectuation` are depicted in Fig. 7.2 as a UML class diagramm. At its centre is the class `EObject` that is used as super-class for all modelled objects, such as metaclass instances, attribute values, or enumeration literals. Some of the most important methods of the convenience classes are also shown in the class diagram in order to illustrate the benefits of semantic information in data structure

[1] YAGNI - You Are not Going to Need It

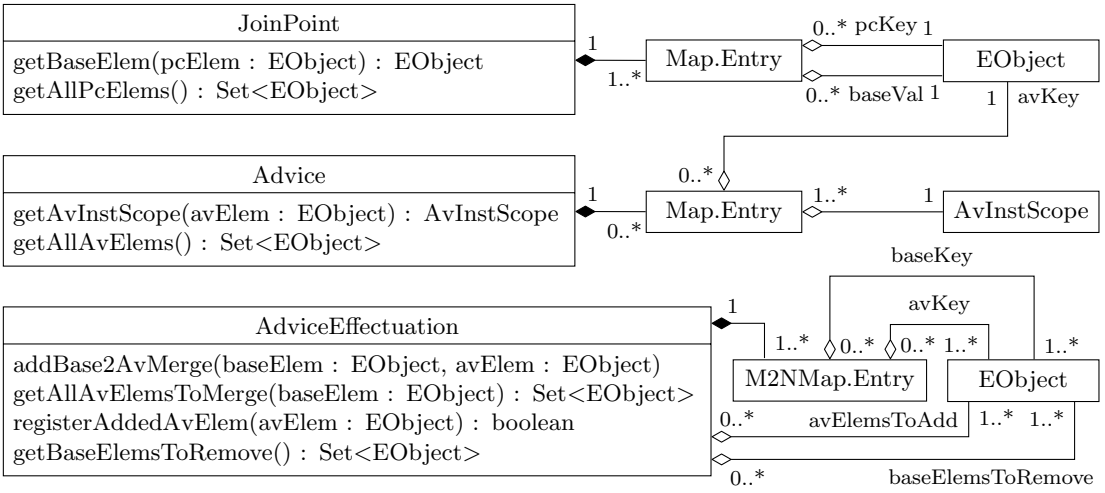


Fig. 7.2.: A UML class diagram showing the complete structure and a part of the method interface of the convenience classes for weaving data.

names. The method `getBaseElement(pcElement)` of the class `JoinPoint`, for example, is far more self-explanatory than the method `get(key)` of a standard `Map`. Each depicted convenience class is explained individually in the following paragraphs.

A `JoinPoint` is a convenience representation for a unidirectional 1-to-1 mapping from elements of a pointcut model to elements of a base model. The advantage over a common `Map<EObject,EObject>` and corresponding calls to the Java API is that semantic information is obvious and does not need to be repeated in comments or variable names.

An `Advice` provides access to all elements of an advice model and to the advice instantiation scope linked to them. Such an instantiation scope specifies under which conditions an advice element has to be re-instantiated for a particular join point (see Section 5.3.2). The convenience wrapper `Advice` encompasses only information of an advice model and therefore it is independent of a specific join point or base model.

All data needed for the effectuation of an advice at a certain join point is encapsulated in the convenience representation `AdviceEffectuation`. It contains the formal sets and mappings introduced in Section 5.3.1, such as the bidirectional m-to-n mapping that relates base and advice elements, the set of advice elements to be added, and the set of base elements to be removed. This encapsulation creates a single point of responsibility for join point-specific weaving data and eases the parameter passing between individual phases of weaving.

7.1.2.3. Separating Helper Modifications

Due to shortcomings of EMF's Ecore API, which we discuss in the next section, we had to extend and change some of the behaviour provided by its utility classes. The individual modifications of existing library functionality were realised in strict separation of each other in order to increase understandability and maintainability. As a result, every class of the package `lu.uni.geko.util.ecore` changes the default implementation of a helper class for exactly one isolated concern.

Fig. 7.3 presents a UML class diagram that illustrates the inheritance hierarchy of our Ecore helper classes. It shows that our strict separation of concerns resulted in comparatively long class names and numerous intermediate classes. In a domain model such a design would be considered of low quality. In this context, however, it is necessary

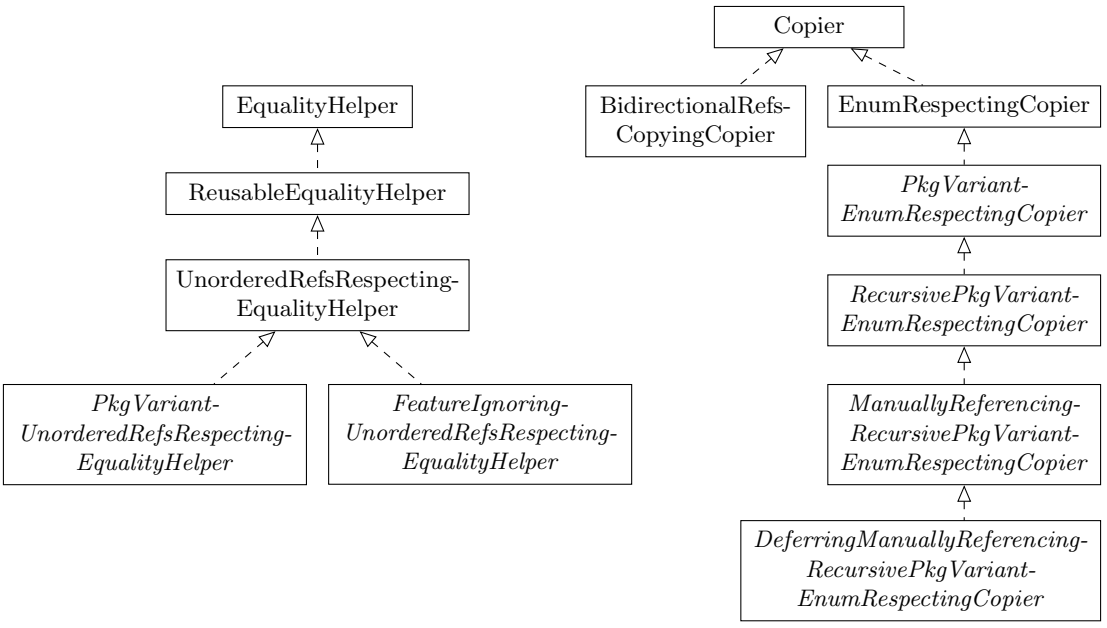


Fig. 7.3.: A UML class diagram showing the hierarchy of Ecore helper modifications.

for these solution-specific helper classes that realise complex model traversals. Let us illustrate this necessity using the inheritance chain that ends with `EnumRespecting-Copier` and spans over five levels of inheritance. The code and JavaDoc documentation of these five classes comprises 461 lines. As only the leaf class is used separately it would have been possible to create a unique class instead of five. The resulting class, however, would probably have posed barriers to maintenance and evolution of the code.

7.2. Overcoming Existing Barriers

In this section we present some of the barriers that we had to overcome when implementing our weaving approach. This is done in order to give the reader the possibility to understand why we could not opt for different solutions. We start with the reasons for the modifications of Ecore helpers that we mentioned above, continue with factors that resulted in a switch from Kermeta to Java, and finish with insufficiencies of the Drools library.

7.2.1. Ecore Utility Helpers

After presenting the separation of individual modifications of EMF's Ecore helper classes in the previous section we explain now why these modifications where necessary. In the package `org.eclipse.emf.ecore.util` the class `EcoreUtil` contains two nested classes `EqualityHelper` and `Copier` that provide functionality to compare and copy model elements. Both helper classes provide functionality needed for our weaver implementation but could not directly be reused in our context.

The class `EqualityHelper` can be used to determine whether two model elements are structurally equal. During comparison this class populates a mapping that relates elements for which the equality relationship has already been established. This comparison, however, is done in a way that restricts the use of this helper class for our weaver implementation in two ways.

Reusing comparison results

First, the class `EqualityHelper` was designed in a way that makes it necessary to newly instantiate the class for every comparison operation as it is assumed that a model element is equivalent to at most one model element. When models are woven using our approach it is, however, necessary to compare numerous model elements that may already have been compared or contain compared model parts. Therefore, it is important to avoid comparisons of model subgraphs that have already been compared. In our implementation we achieved this by creating the class `ReusableEqualityHelper` that extends `EqualityHelper` and overrides small parts of its behaviour. These modifications allow us to store an m-to-n equivalence mapping instead of the original 1-to-1 mapping. As a result, it is possible to use a single instance for multiple comparisons while ensuring that subgraphs for which equivalence was already analysed are not traversed a second time.

Respecting unordered references

Second, when values of references are compared the attribute `unordered` is not respected by the implementation of `EqualityHelper`. Let us consider two instances of a metaclass with a reference for which the `unordered` attribute is set to `true`. Suppose both instances have exactly the same feature values except for the values of the unordered reference. If both elements refer to the same elements but list them in a different order the implementation of `EqualityHelper` considers them to be unequal. As this is not conformant to the semantics of the `unordered` attribute we created a class `UnorderedRefsRespectingEqualityHelper` which changes this behaviour. Whenever values of an unordered reference are compared the order of values is ignored and it is only checked that the same values are present. This modification is particularly important for testing our weaver implementation because of the non-determinism of the join point detection. Different orders of join points can, for example, result in woven models that are equivalent but contain newly added elements in varying order. If the underlying reference is unordered our helper modification ensures that these woven models are considered equal.

7.2.2. Kermeta

We initially started to implement our generic weaving approach using the model-transformation language Kermeta (see Section 2.1.2). During the work on this early prototype, however, we discovered that an interoperability restriction made this language inadequate for our purposes. This restriction makes it impossible to use custom model loading and storing functionality without spending substantial effort on modifications of Kermeta's runtime environment.

No custom loading with Kermeta

A main goal for our implementation is real genericity in the sense that the weaver implementation can be used with arbitrary models that conform to an Ecore metamodel. This requirement makes it necessary to provide users the possibility to load and store their models using their own code or frameworks. Kermeta, however, was designed in order to relieve the user from the necessity to specify how a model is loaded or stored. When custom model loading and storing functionality is to be used in Kermeta this functionality has to be available as a Kermeta or Java method. The first possibility of loading and storing models using custom methods written in Kermeta would restrict the applicability of our weaver to models for which Kermeta serialisation implementations exist. This is rare and specifically not the case for building models, which we load using the technological bridge for IFC and EMF (see Section 2.4). Therefore, this possibility could not be pursued for our weaver implementation. The second possibility of loading and storing models using Java methods would involve passing complex-typed parameters between Java and Kermeta. Such parameters would involve conversions from the runtime format of Java to that of Kermeta and vice-versa. But this is not supported by the current version of the Kermeta runtime. Therefore, we tried to add support

for Java parameters of complex type to the Kermeta framework. We had to abandon this plan because a successful implementation within the time constraints of this thesis seemed out of reach. As a result, the second possibility for custom model loading on top of Kermeta proved also to be infeasible for our purposes. Altogether this practically forced us to implement our weaving approach using Java although theoretically Kermeta might have been more appropriate for this task.

7.2.3. JBoss Drools

The business logic integration platform Drools which we use for join point detection and presented in Section 6.1.2 posed a technical problem to our implementation.

No runtime-generated classes with Drools

Drools supports rules that involve POJOs generated at runtime, but the definition of these rules has to take into account whether the corresponding classes were made available at compile time or at runtime. As a result, we were unable to use the rules that work flawlessly for statically registered metamodels in a dynamic setting where the metamodel implementation is only available at run-time. This is unfortunate because users of our weaver should have the ability to weave models independent of the point of time at which the corresponding metamodel is available. Drools, however, currently forces us to require users to statically register metamodel code before starting our weaver. We keep on working on this issue in order to free users from the resulting inconveniences in the future.

If it becomes apparent that Drools does not meet our requirements, it could be that supporting another join point detection engine, such as EMFQuery (see Section 12.2.2), is the only remaining solution. This option may also become interesting in case Drools does not scale to large input models or a multitude of join points.

7.3. Algorithms

This section presents some of the algorithms that we used to implement our generic weaving approach. Chapter 5 explained conceptually *what* individual steps are required for weaving whereas this section focuses on *how* these steps are actually performed.

7.3.1. Pointcut Rules Generation

A two-pass visitor for pointcuts

The first algorithm that we use in our weaver generates rules for the logic framework Drools as a preparatory step for join point detection (see Section 5.1). In Algorithm 1 we present a pseudocode representation of this pointcut rules generation algorithm. It is devised into a main routine `generatePointcutRules` and two subroutines `declareStructure` and `declareReferences`. This is due to the fact that the rules are generated by visiting all elements of the pointcut model twice in a depth-first manner.

The main routine `generatePointcutRules` initialises an empty rule and retrieves all model elements that are direct child elements of the pointcut model, which was passed as an argument (line 3). After that, both subroutines `declareStructure` and `declareReferences` are called individually using these first layer elements as arguments. Finally, the obtained rules are appended to the overall rules and returned as the result of the algorithm.

During the course of the subroutine `declareStructure` (line 8) all passed elements are inspected individually and variables that define the structure of each element are declared (line 11). These variables ensure that only elements of the correct type are detected. In order to restrict the set of matching elements further, these variables are constrained using the name and value of each attribute of each element (line 11).

Algorithm 1 Pointcut Rules Generation

```
GENERATEPOINTCUTRULES(pcModel) {                                                    1
   rules ← ""                                                                       2
   firstLayerElements ← direct content elements of pcModel                          3
   rules.append(declareStructure(firstLayerElements)) // first pass                 4
   rules.append(declareReferences(firstLayerElements)) // second pass               5
   return rules                                                                     6
}                                                                                   7

DECLARESTRUCTURE(elements) {                                                        8
   rules ← ""                                                                       9
   for all elements e {                                                             10
      declare a variable for the structure of e                                     11
      for all attributes a of e {                                                   12
         name ← name of attribute a                                                 13
         value ← e's value for a                                                    14
         restrict the variable using the name and the value                         15
      }                                                                             16
      children ← direct content elements of e                                       17
      rules.append(declareStructure(children)) // recursion                         18
   }                                                                                19
   return rules                                                                     20
}                                                                                   21

DECLAREREFERENCES(elements) {                                                       22
   rules ← ""                                                                       23
   for all elements e {                                                             24
      declare a variable for the references of e                                    25
      for all references r of e {                                                   26
         value ← e's value for r                                                    27
         var ← the structural variable for value // from the first pass             28
         restrict the variable by referring the structural variable for r           29
      }                                                                             30
      children ← direct content elements of e                                       31
      rules.append(declareReferences(children)) // recursion                        32
   }                                                                                33
   return rules                                                                     34
}                                                                                   35
```

As a result, the join point detection will only yield elements of the base model that exhibit the same attribute values as the elements of the pointcut model. We make sure that all elements of the pointcut model are considered by recursively calling this subroutine for all elements that are contained in the passed elements (line 18). Finally, the obtained rules are returned. For the main routine this means that rules for the structure of all nested elements of the pointcut model are created before the next subroutine `declareReferences` is executed.

In a second pass performed by the subroutine `declareReferences` (line 22) all references of pointcut model elements are inspected in order to create according join point detection rules. The overall structure of this subroutine is very similar to that of `declareStructure`: In a depth-first manner all passed elements and their children are visited using recursion (line 32) and at the end the obtained rules are returned. The subroutines mainly differ in the processing of the individual elements (lines 26 – 30). This is because the references that are inspected in the second subroutine refer to complex types that cannot be directly expressed as it is the case for the attribute values of simple type from the first pass. Thus, when a reference of an element is inspected,

Algorithm 2 Pointcut Advice Correspondence Inference

```
INFERPC2AVMAPPING(pcElements, avElements) {                                  1
   mapping ← ∅                                                               2
   for all pcElements p {                                                    3
      compute potentially corresponding subset of avElements                 4
       // per default these are all advice elements of same type             5
      pcID ← calculate unique identifier of p                                6
      potential match ← nil                                                  7
      ambiguous ← false                                                      8
      for all potentially corresponding advice elements a {                  9
         avID ← calculate unique identifier of a                             10
         if (avID = pcID) {                                                  11
            if (potential match = nil) {                                     12
               potential match ← a                                           13
            }                                                                14
            else {                                                           15
               ambiguous ← true                                              16
            }                                                                17
         }                                                                   18
      }                                                                      19
      if (not ambiguous ∧ potential match ≠ nil) {                           20
         add correspondance (p,a) to mapping                                 21
      }                                                                      22
   }                                                                         23
   return mapping                                                            24
}                                                                            25
```

we obtain the variable that was used in the first pass to define the structure of the referenced element (line 28). This structural variable is then used to restrict the new reference variable that was created for the referring element (line 29).

Altogether, we create two variables for each element of the pointcut model during the execution of the two subroutines. This clear separation of structure and references is needed because Drools does not permit rules that refer to variables that are declared after the reference.

7.3.2. Relating Pointcut and Advice Elements

In another preparatory step carried out before weaving correspondences between pointcut and advice elements are obtained from the user or automatically inferred (see Section 5.2). For all elements of the pointcut and advice model that have not been mapped to a corresponding counterpart by the user a mapping is inferred using Algorithm 2. This means that the same algorithm is used for the complete inference of a mapping and for the completion of a partial mapping provided by the user.

Matching elements based on identifiers

The inference algorithm starts with an empty mapping and processes every passed pointcut element individually. For a pointcut element it is computed which of the passed advice elements could possibly correspond to it (line 4). If no extension is provided the result defaults to all advice elements that have the same type like the pointcut element. Then, the unique identifier for the pointcut element is calculated. Per default this unique identifier is a concatenation of all values of attributes of string type. After that, it is determined for each of the potentially corresponding advice elements whether it has the same unique identifier. If this is true for an element but was not true for another advice element before, the advice element is stored as a potential match (line 13). If the same identifier was calculated for more than one advice element this means that

Algorithm 3 Copy Advice Element

```
COPYAVELEMFORBASE(avElement) {                                                    1
   avInstScope ← get advice instantiation scope for avElement                    2
   if (avElement already copied within avInstScope) {                            3
      copy ← existing copy for avElement                                         4
   }                                                                             5
   else {                                                                        6
      copy ← empty base element container of same type like avElement            7
      register copy for avInstScope                                              8
      for all attributes a of avElement {                                        9
         value ← get avElement's value for a                                     10
         set corresponding attribute in copy to value                            11
      }                                                                          12
      for all references t of avElement {                                        13
         refElem ← get avElement's value for r                                   14
         refElemCopy ← copyForBase(refElem) // recursion                         15
         set corresponding reference in copy to refElemCopy                      16
      }                                                                          17
   }                                                                             18
   return copy                                                                   19
}                                                                                20
```

the current aspect is ambiguous in the sense that a mapping from pointcut to advice elements cannot be calculated based on the identifiers (line 16). This means that it is sufficient to check whether exactly one potential match was determined once all advice elements have been inspected (line 20). If this is the case a correspondence for the pointcut and advice element can be added to the mapping (line 21).

Altogether, the algorithm returns a mapping that contains an entry for every pointcut element for which exactly one advice element with the same unique identifier could be determined. This means that the algorithm only produces 1-to-1 mappings. In cases where more than one pointcut element or more than one advice element should be part of a mapping entry this has to be specified by the user.

7.3.3. Copying Advice Element for the Base

When new elements are added to the base model during the weaving, copies of the corresponding elements of the advice model have to be created. How these copies are obtained based on the advice instantiation scope (see Section 5.3) is presented in Algorithm 3.

Recursively copying contained elements

Initially it is determined whether the advice element that has to be copied was already copied for the current advice instantiation scope (line 3). If this is the case this existing copy will be returned as result (line 4). Otherwise, a new empty element of the type of the advice element is created and registered as copy for the advice element and the current advice instantiation scope (line 8). Then, the attribute values of the empty copy container are set to the corresponding attribute values of the advice element (lines 9 – 12). Next, each of the advice element's references to complex types is processed in two steps. First, a copy of the referenced element is obtained using recursion (line 15). Then, the value of the inspected reference is set for the copy of the advice element in consideration. When all attributes and references have been processed the copy is returned as result.

For this recursive algorithm it is crucial that new copies are registered before recursive calls occur during the processing of references. This is the case because an advice

element may directly refer to itself or indirectly refer to elements that refer back to the advice element. If a copy would only be registered once the copying process is finished these circular references would have to be handled separately. The presented algorithm, however, is robust with respect to these references as new copies are already taken into consideration while their references are copied.

7.3.4. Adding Advice Elements to the Base

After an advice element has been copied for introduction into a base model the copy has to be added to the containment hierarchy of the base model. This is done iteratively using the main routine `addAvElements` of Algorithm 4 and supported by its subroutine `getContainment`.

Best-effort containment

At the beginning of the algorithm, helper variables for the number of iterations, the set of already added advice elements, and the set of base variants of these already added advice elements are initialised (lines 2 – 4). Then, as many iterations as advice elements have to be added are performed. In each iteration an attempt to add each of these elements is performed on a best-effort basis. Such an attempt of addition consists of three steps. First, the base variant of the advice element to be added is obtained. If the advice element has not been copied so far this step results in copying as discussed in the previous section. Second, it is determined whether this base variant was already added in an earlier iteration (line 8). As nested containment relations may make explicit additions unnecessary, it is then checked whether the currently considered advice element is indirectly contained in an advice element that has already been added by the algorithm. In such a case the current advice element is already contained in the woven model and requires no further processing. The same applies if the advice element itself was already added by the algorithm, so this is checked too. If both checks reveal that the current advice element is neither indirectly nor directly an already added element the subroutine `getContainment` is called (line 14). Once this call returned a container and a containment reference for the current advice element, these two results are used to actually add the element to the woven model. Afterwards, the helper data structures are updated: the current advice element is added to the set of already added advice elements and the base variants of it as well as its children are added to the set of already contained base variants. Finally, whenever an iteration over all advice elements is finished the iteration count is increased by one.

The subroutine `getContainment` (line 25) is responsible for inferring the container and containment reference to be used for addition of a given advice element. Initially, it is checked whether the advice element is directly contained within the root element of the advice model. If this is the case an attempt for unambiguously inferring the container and containment reference is performed. This is similar to the inference of unambiguous pointcut to advice mappings (Algorithm 2). First, inference helper variables that store a potential match and detect whether ambiguity occurred are initialised. Then, all containment references of the base model's root element are inspected. Whenever the advice element that was passed as an argument to the subroutine matches the type of a containment reference or one of its subtypes the reference is a possible match (line 31). If this is not the first reference that yields a match a containment reference cannot be chosen unambiguously which hast to be noted (line 36). Once all containment references of the base root element were inspected this information is used to check whether no more than one matching reference was encountered. In this case, the base root element and the stored matching reference serve as container and containment reference (lines 40 – 43).

If the first check of the subroutine `getContainment` reveals that the advice element that was passed as an argument is not a direct child of the advice model's root element

Algorithm 4 Add Advice Elements

```
addAvElementsToBase(avElements) {
   iteration count ← 0
   added advice elements ← ∅
   added base variants ← ∅
   while (iteration count < |avElements|) {
      for all avElements avElem {
         base variant ← get base variant of avElem
         if (base variant is a member of added base variants) {
            add avElem to added advice elements
         }
         else {
            if (avElem not indirectly contained in an added advice element
                  ∧ avElem not a member of added advice elements) {
               (container, containment reference) ← getContainment(a)
               try to add avElem to container using containment reference
               add avElem to added advice elements
               add base variants of avElem to added base variants
               do the same for avElem's children
            }
         }
      }
      iteration count ← iteration count + 1
   }
}

getContainment(avElement) {
   if (avElement is a direct child of advice model root element) {
      potential match ← nil
      ambiguous ← false
      for all containment references r of the base root element {
         refType ← get referenced type for r
         if (refType is avElement's type or super-type) {
            if (potential match = nil) {
               potential match ← r
            }
            else {
               ambiguous ← true
            }
         }
      }
      if (not ambiguous) {
         container ← base root element
         containment reference ← potential match
      }
   }
   else {
      container ← base variant of container of avElement
      containment reference ← reference containing avElement
   }
   return (container, containment reference)
}
```

it is not necessary to guess a containment. The base variant of the parent of the advice element and the reference used for containment can serve as container and containment reference (lines 45 – 48).

7.4. Data Structures

As we mentioned in Section 6.1.2 our implementation did not require any non-standard data structures except for a many-to-many mapping and its bidirectional variant that we present in this section. To our surprise, standard libraries like Apache Commons or Google Guava do not provide an implementation of a bidirectional mapping that relates multiple keys to multiple values and vice-versa. Therefore, we had to develop our own implementation in order to realise the conceptual mapping from pointcut to advice elements described in Section 5.2.

7.4.1. Many-to-Many Mapping

Additional integrity constraints

Although implementations of unidirectional many-to-many mappings are available we decided to use a custom implementation that meets all our requirements and provides convenience methods while giving us full control. In contrast to existing implementations, such as Guava's `MultiMap`, our notion of many-to-many mapping requires that an element occurs at most once as a key or value of an entry. This means, given an entry that maps a set k_1 of keys to a set v_1 of values and an arbitrary other entry that maps a set k_2 of keys to a set v_2 of values, the intersection of the key sets $k_1 \cap k_2$ and the intersection of the value sets $v_1 \cap v_2$ has to be empty. In our implementation we ensure this constraint by throwing an appropriate exception in case of violating add operations for a many-to-many mapping.

Fig. 7.4 shows a UML class diagram that displays the classes and interfaces that we created to realise uni- and bidirectional many-to-many mappings with an extract of their methods and attributes. The interface for unidirectional many-to-many interfaces `M2NMap` provides two operations for adding new mapping entries. Both operations throw an exception if they are called using a key or value that is already mapped. In accordance with this constraint we only provide a restricted remove operation. It removes a mapping from a key to a value if no other key is involved in the corresponding mapping. If it would have been needed, we would also have defined a remove operation for a set of keys and a set of values. Note, however, that such an operation could only ensure our constraints for many-to-many mappings if it would remove an entry that maps exactly the given keys to the given values and does not involve any additional keys or values.

We provided a default implementation for unidirectional many-to-many mappings called `HashM2NMap`. It uses a conventional `HashMap` to map a set of keys to a set of values as shown in Fig. 7.4. This class also provides a caching mechanism, which avoids unnecessary iterations over all mapping entries when all values for a given key shall be returned.

7.4.2. Bidirectional Many-to-Many Mapping

To our knowledge, no implementation of the bidirectional many-to-many mapping that is used in our weaving approach during the merge of base and advice elements (see Section 5.3.3.2) is available. Therefore, we created an own implementation based on the unidirectional many-to-many mapping that we presented in the previous section.

As shown in Fig. 7.4, the interface for bidirectional many-to-many mappings `BiM2NMap` extends the interface `M2NMap` and adds an operation that returns all keys that map to

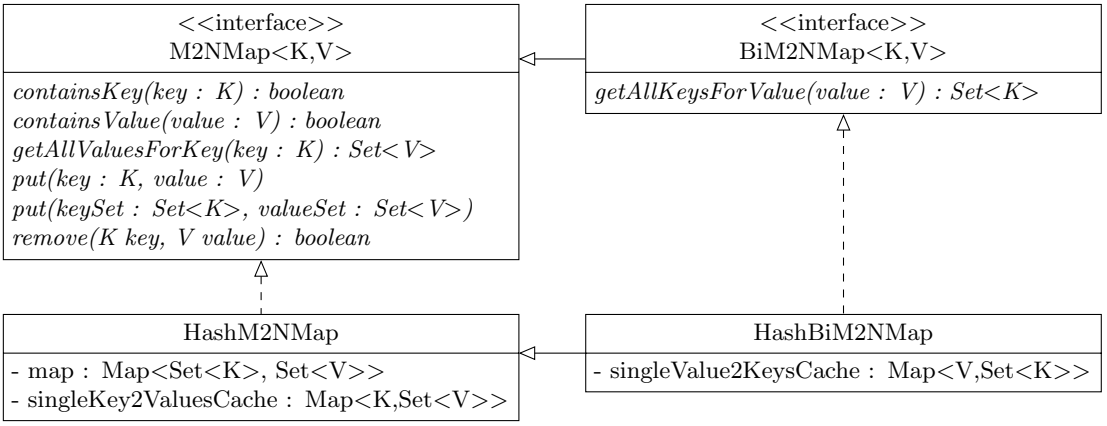


Fig. 7.4.: A UML class diagram showing the interfaces and classes for uni- and bidirectional many-to-many mappings and some of their attributes and methods.

a given value. The default implementation `HashBiM2NMap` extends its unidirectional counterpart `HashM2NMap` and expands the caching mechanism. In analogy of the unidirectional cache the bidirectional variant avoids iterating over all entries in cases where all keys for a given value shall be returned.

7.5. Summary

This chapter presented a selection of solutions to problems that we encountered during the implementation of an extensible prototype for our generic model weaver. We explained how we strived for regularity by using known design patterns and custom solution patterns. We also discussed technical barriers posed by employed libraries and platforms and we explained how we overcame them. Furthermore, we provided a language-independent presentation of detailed algorithms used at critical points of weaving. Last, we pointed out why we had to develop own data structures for unidirectional and bidirectional many-to-many mappings and explained their design.

In the next part of the thesis, we discuss cross-cutting concerns contained in building specification texts and explain how we used them to evaluate our extensible model weaver. The following chapter picks up a discussion on domain-specific aspects and argues why building specifications contain such aspects.

Part IV.

Building Specifications as Domain-Specific Modelling Aspects

8. Building Specification Aspects

In this chapter we show that building specifications contain cross-cutting concerns and therefore can be treated as aspects in the sense of Aspect-Orientation (see Section 2.2). The purpose of a building specification is to provide additional details for an existing building model. How these details are expressed depends on the concern itself and on external factors such as contractors and standards. That some of the building specification concerns are aspects of the problem domain is not obvious. In the literature no consensus on the question whether such domain-specific aspects exist could be established. Therefore, we discuss some of the relevant publications on domain-specific aspects in Section 8.1 before we show in Section 8.2 that building specifications contain such aspects.

8.1. Domain-Specific Aspects

"Domain models are aspect free." [Ste05]

In 2005 Steimann [Ste05] formulated the thesis that domain models contain no aspects. He discussed various types of aspects, provided a semi-formal proof of his thesis, and defined criteria for disproving counterexamples. All his statements are based on the assumption that aspects are necessarily characterised by two properties: First, an aspect controls the areas to which it applies (quantification). Second, other entities than aspects are unaware of possibly applying aspects (obliviousness). We will now discuss and analyse Steimann's argument in detail.

The first context for aspects that Steimann discusses are roles, which he characterises as properties that crosscut types in one or both of two possible ways. According to Steimann, a role can lead to objects of different types having identical properties and it can lead to objects of same type having different properties. After this observation Steimann claims that these identical properties are "usually" realised differently for different types. From this proposition he concludes that roles cannot be represented by aspects. Although such a polymorphic implementation may be used in various cases, it is not a necessary prerequisite for arbitrary roles, nor does it need to be true for most roles. Furthermore, it may be argued that this statement evolves from a perspective that is rather concerned about the implementation and not suitable for an argument on models.

Similar to his first generalisation on roles, Steimann postulates that these have to be determined by collaboration. Again, this may be correct in various situations but does

not describe the general case. For example a role that encapsulates the ability to float on water does not need another role to make sense. Such a role can exhibit a property, for example a notion of uplift, in isolation of other roles. Steimann uses his claim of strictly collaborative roles to argue that aspects cannot realise roles as a result of their necessary obliviousness. The empirical results of Hoffman and Eugster [HE08], however, showed that at least for AOP a combination of oblivious aspects and explicit interfaces for cross-cutting concerns results in the best code quality. This puts in doubt whether aspects have to be oblivious if they are to express cross-cutting concerns.

Steimann closes his considerations on role aspects with a summarising role definition that he considers incompatible with aspects. He denotes roles as named types that specify properties that are determined by collaboration with other roles and typically implemented in a polymorphic fashion. This allows him to conclude that aspects cannot represent roles as they are neither types, nor introducing polymorphic realisations, nor collaborative as this would contradict obliviousness.

The core of Steimann's arguments is a semi-formal proof of his thesis that domain-models are aspect-free. As a motivation, he provides some traditional example aspects for AOP and generalises from these that standard aspects are rather part of the (programming) solution than of the problem domain. Similar to his first argument on roles, it is questionable whether this observation from AOP should be used for arguments on domain-models. We separate our analysis of the proof into four parts. The first two parts are observations, the third part transfers the second observation to AOM, and the last part draws the conclusion.

First-order domain models vs. second-order aspects

First, Steimann states that domain-models are first-order models similar to first-order logic because propositions about propositions would describe the model and not reality. It is not clear to us why a model may not describe concerns of the reality in a way that reasons about entities of the reality using the descriptions of the model. If a model is nothing different than a partial representation of the reality that uses a well-defined formalism (see Definition 1, p. 9), we cannot exclude the possibility that this representation makes use of its own concepts in order to describe the reality. Therefore, we even have to consider the possibility that models may exists that cannot be described *without* second-order statements. A detailed analysis of this question would be out of scope of this thesis, but the reader may find it convincing that in cases where an infinite part of reality is to be described a first-order modelling language may not be sufficient. But even if second-order models may not be necessary, it is questionable whether all domain-models have to be first-order just because the same reality might also be represented without second-order statements.

In the second step of his proof Steimann notes that aspects of the AOP language AspectJ [KHH+01] are second-order concepts as they can contain variables and expressions that range over first-order concepts of Java such as classes and methods. This observation from the world of AOP is transferred to AOM in the third step of the semi-formal proof: Without any further support or evidence Steimann postulates that "this applies equally to aspect-oriented modelling". But we are convinced that the fact that this observation is true for AspectJ does not necessarily mean that it has to be true for every aspect concept of every AOM approach.

Steimann draws his final conclusion that pure domain models are aspect-free in the last part of his proof using his observation from the first step and the transferred observation of the second and third step. If domain models are first-order and aspects second-order, then it follows that domain models contain no aspects. As we doubt the correctness of the premises we cannot endorse the conclusion.

In order to clarify his argument Steimann provided falsification criteria in the form of requested properties for counterexamples. He demanded that counterexamples have to contain aspects that are aspects in the aspect-oriented sense and thus no subroutines or roles. Furthermore, these examples must not be artefacts of the technical solution but have to be part of the problem domain. Last, examples should be chosen arbitrary in order to provide the possibility to find further ones.

"Domain models are not aspect free." [RM06]

Although Steimann proposed a strong thesis we are only aware of a single publication by Rashid and Moreira [RM06] that argued against it. Rashid and Moreira refuted individual statements of Steimann, such as the incompatibility of aspects and roles, the non-polymorphism of aspects, and the first-orderness of aspect, but their main contribution concerned a characterisation of aspects. They questioned Steimann's thesis that aspects cannot represent roles using the work of Hannemann and Kiczales [HK02], which showed improvements for code quality metrics when roles of design patterns where realised using AOP. Furthermore, Rashid and Moreira challenged Steimann's claim of non-polymorphism by pointing out several AOP techniques that allow different implementations of an aspect in a polymorphic way. They were able to invalidate the thesis on the second-order nature of aspects using multi-dimensional AOM approaches. As proposed by Tarr et al. [TOHS99] multi-dimensional approaches allow composition and decomposition of systems along various dimensions instead of a single one. This means for AOM that no distinction between base concerns and cross-cutting concerns is possible and thus all concerns are first-order.

Furthermost, Rashid and Moreira provided an alternative to Steimann's classification of aspects by quantification and obliviousness. To this end, they also analysed aspect-oriented approaches for requirements engineering, which are mostly discussed using the term Early Aspects. Rashid and Moreira argued that, independent of the fact that some of these approaches satisfy the obliviousness and quantification property, aspects are rather about abstraction, modularity, and composability. We will come back to these three properties in the following discussion on the aspectual nature of building specifications.

8.2. Building Specification Concerns as Aspects

This section shows the presence of domain-specific aspects in building specifications based on the precedingly presented discussion on domain-specific aspects. Using two examples from a construction project, which was realised for the Government of Queensland, Australia, we illustrate the aspectual nature of building specification concerns. By this means we contribute supporting details that we could not provide in an initial position paper due to size constraints [KKS12c].

Building models exhibit a dominant decomposition paradigm that is closely coupled to the geometry as it recursively divides projects into sites, buildings, storeys, and, finally, spaces or rooms. For example in building models encoded using the IFC format (see Section 2.3.1) every individual model element has a specific geometric position and an explicit relation to a space that belongs to a storey of a building on a site. The spatial dimension is the dominating decomposition paradigm not only for the internal organisation but also for various visualisations of building models. This even holds for partial models for example when a construction project is first decomposed into structural, architectural, mechanical, and electrical perspectives and then spatially organised.

Having identified the main decomposition paradigm of building models allows us to show that building specifications contain concerns that crosscut this decomposition in

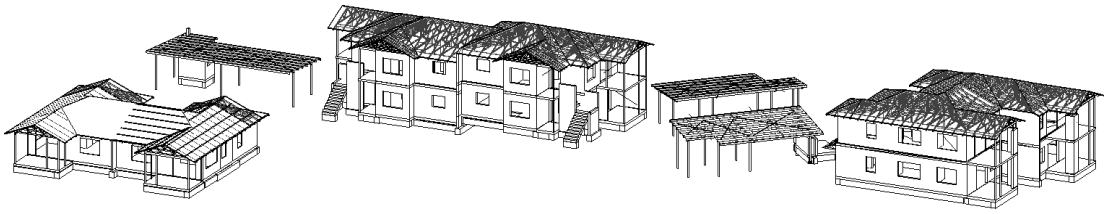


Fig. 8.1.: Structural model of a housing complex with a cross-cutting specification concern requiring an additional fire alarm for two-storey apartments.

an aspectual way. In the context of this thesis we analysed for example a project for a housing complex that involves multiple buildings with multiple apartments. A view on the structural model of this project is shown in Fig. 8.1. The building specification for this project contains the following sentence:

> Where the dwelling is a two storey type, provide an additional smoke alarm in the storey not containing bedrooms, adjacent to the internal stairs.

Crosscuting spatial decompositon

The concern expressed with this sentence affects various apartments of different buildings and contains a condition that depends on multiple spaces (bedrooms and stairs) of different storeys. Therefore, it crosscuts the main decomposition. Furthermore, at least two of the properties that Rashid and Moreira consider characteristic for aspects are satisfied: First, the concern abstracts the individual layout of the apartments as well as the exact location of the bedrooms and stairs. Second, it reasons about the overall size of apartments and about the locations of bedrooms and stairs in isolation from other properties (modularity).

Coming back to the discussion on domain aspects, we can show that this exemplary specification concern satisfies the conditions that Steimann required for a falsification of his thesis. The concern exhibits none of the properties of a subroutine or role and thus is an aspect in the aspect-oriented sense according to Steimann. Regarding the second condition that excludes technical aspects, one might argue that a building model and the corresponding building specification already represents the solution to a "problem" respectively the realisation of a requirement. But this is irrelevant for the given example as it is easy to reformulate the concern in a problem-oriented way. Without any references to parts of the solution the example sentence could be rewritten to:

> All bigger apartments have to be secured against fire according to their size in proximity of the location of beds.

This reformulation shows that the specification concern is indeed part of the problem and not the solution domain. Steimann's third and last requirement for domain aspects is an arbitrary choice of counterexamples that we satisfy by providing another example.

An abstract, modular, and composable example concern

In the same construction project another cross-cutting specification concern demands insect-screen doors at all external doors. First, we consider Steimann's falsification requirement that asks for counterexamples that are no roles. At first sight, one could argue that being an external door could be modelled using a role for ordinary doors. But a closer look reveals that the concern is not about properties that are shared by external doors. It simply demands the presence of another entity (insect door) whenever a given entity (door) occurs in a specific context (in an outside wall) without changing the properties of the original entity. The second requirement is met as the occurrence of insects as well as the requirement to avoid them is certainly part of the problem and not the solution domain. To proof compliancy with the requirement of arbitrariness, various further concerns could be listed but this is not within the scope of this thesis.

We can also show for our second example concern that it satisfies the characteristic properties of aspects according to Rashid and Moreira. First, the concern abstracts the individual properties, types, and occurrences of doors throughout the project. Whether a door provides access to the external area on the ground floor or whether it is adjacent to a balcony on the first or second floor has no influence on the concern. Second, the fact that a door faces outwards is used in isolation of other properties of the door or other building elements (modularity). Last, the concern can be realised in a way that may allow its use together with realisations of other concerns as it does not change modelled elements in an unexpected way (composability).

8.3. Summary

In this chapter we analysed the relation between building specifications and different perspectives on the notion of domain-specific aspects. Based on an analysis of supporting and opposing arguments for the thesis that domain models are aspect free we showed that building specifications contain domain-specific aspects. We did not claim that all building specification concerns can be treated in an aspect-oriented way. Nevertheless, the existence of domain-specific aspects is fundamental for our proposition of an aspect language for building specifications, which we present in the next chapter.

9. Proposing an Expert-Driven Building Specifications Language

After we showed in the preceding chapter that building specifications contain domain-specific aspects we are now prepared to analyse how these aspects can be expressed in a way that facilitates automatic introduction into building models. To this end we propose an approach that lets domain experts define a DSMTL (see Section 2.1.3) for building specifications. In Section 9.1 we argue why this language should be a form of restricted English that is also known as Controlled Natural Language (CNL). Some initial ideas on how this DSMTL for building specifications could be realised using interpretation patterns are presented in Section 9.2. Section 9.3 illustrates our proposal using an exploratory example.

Note that we do not attempt to provide a plan for the realisation of a language for executable building specifications. This thesis focuses on investigating whether and how model weaving aspects can be used to introduce domain-specific information into models. An automated integration of building specification concerns into building models is only one of multiple possible applications for our generic approach to model weaving. The work presented in this chapter is the result of a first exploratory step on the way to the question how BIM aspects for our weaver could be obtained in the future. We hope that our suggestions give the reader and future researchers an idea of the context and conditions in which executable building specification aspects could be created.

This chapter is embedded in the overall structure of this thesis as follows. The motivation for integrating building specifications concerns into building models is given in Section 1.2. In Section 10.1 we explain how the aspects that could be produced according to the propositions of this chapter can be woven into a building model using our generic weaving approach. How an illustrative example for such aspects is woven is presented in Section 10.2. Another DSMTL for building models that was presented by Steel and Drogemuller [SD11] is discussed in Section 11.3.1.

9.1. A Controlled Natural Language

Control the language to ease analysis

We suggest using a CNL for the definition of executable building specifications because the syntactical nature of building specifications and their integration into the overall construction process match the characteristics of such a language. CNLs such as Attempto Controlled English by Fuchs and Schwitter [FS96] try to combine the

advantages of natural languages with these of machine-processable languages. They are characterised by a controlled grammar and restricted vocabulary which increases precision and reduces ambiguity and complexity in order to facilitate automatic analysis. The structure, regularity, and precision of building specification texts and the characteristics of the domain experts that formulate them as well as their role in the construction process cause the applicability of CNLs.

Although the structure and design of CNLs can strongly reflect the intended purpose some common problems and solution patterns can be identified. The vocabulary can be divided into a fixed set of functional keywords and an extensible or even replaceable set of content words of the domain. For further flexibility extension mechanisms may deal with unknown words that are neither keywords nor registered content words. To reduce ambiguity syntactical rules can demand clear formulations and interpretation rules can clarify the semantics of possibly misunderstood parts of the language. Depending on its purpose, a CNL can also exhibit advanced concepts such as composite sentences or references to objects of preceding phrases. An example of such a sophisticated CNL that even supports tabular forms is Semantics of Business Vocabulary and Business Rules (SBVR)[1], a standard that was defined by the Object Management Group (OMG) in the context of the Model-Driven Architecture (MDA) standard.

Building specifications exhibit a level of detail and a structural and syntactic regularity that contributes to the definition of a corresponding CNL. Although building specification texts can also contain rather vague workflow instructions for contractors they are predominantly technical documents that exhibit a high level of implicit formalisation. The specificity of building specifications also results from their legal character, which makes them important for negotiations and disputes with contractors. Avoiding ambiguities is a renowned objective in the context of building specifications that does not need to be newly introduced with a CNL. Furthermore, the commonalities of construction projects are already used to create building specifications with a consistent textual structure. Some of the recurring details such as the default thickness for external, internal, and fire doors are not project-specific but nevertheless mentioned in the specification text. This structural and syntactical regularity limits the size of the grammar and vocabulary and ensures that efforts to design a CNL will not be lost for an individual project but can pay off in various construction projects.

A language for domain experts and MDE laymen

A key factor for our proposition to use a CNL for building specifications is the special situation of the domain experts that define these. As building specification texts play a crucial role during the realisation of construction projects they are formulated by experts that have to possess a substantial amount of domain knowledge. These domain experts would also be the possible users of a building specifications CNL in a process that directly integrates the appropriate concerns into the building model. It has to be expected that these experts are programming laymen and not familiar with MDE concepts such as model transformations. Therefore, learning a programming or model transformation language will require more effort than understanding the restrictions of a domain-specific CNL. Furthermore, it has to be taken into account that the stakeholders that are currently writing building specification texts and who are to write model transforming instructions expressed using a CNL will not be the principal beneficiaries of the workflow change. Thus, it can be expected that these persons will have less interest in adopting the new technique than it would be the case in a setting where the employees that are affected by a change are also profiting from it.

The integration of building specification texts into the established overall workflow during construction projects demands for an approach to executable building specification

[1] SBVR - OMG standard CNL for business descriptions: omg.org/spec/SBVR

texts that fits into the context. According to Eastman et al. [ETSL11] building specifications are commonly an inherent part of construction processes. Therefore, it could be beneficial to include executable specification details that were expressed using a CNL in the traditional building specification document. In order to further bridge the gap between specification texts and models it would also be possible to provide traces that relate integrated concerns of the specification text with the corresponding parts of the model and vice versa. This could increase traceability and allow ordinary use of sentences of the CNL by humans regardless of their simplified structure or additional execution purpose. Because of these possibilities we are convinced that such an approach will have less negative impact on existing tasks and workflows than a solution that executes specification concerns without a human-readable trace.

Transformative alternatives to a generative CNL

For the integration of specification concerns into building models a CNL is only one of various possible approaches. Instead of directly formulating some of the concerns of building specifications using a CNL it would also be possible to continue formulating these concerns using ordinary natural language and to process them afterwards. As this approach would require advanced natural language processing techniques, it is presumably more complex than the human-supported removal of ambiguity that occurs when using a CNL. Another approach, which would also keep building specifications in their common natural language format, could use modelling experts for the manual or semi-automated integration of modelling concerns. This would require the modelling experts to read and understand the building specification in order to perform the corresponding actions using appropriate model editors.

We argue that both solutions are less appropriate than a CNL as they suffer from two problems. First, writing and understanding building specifications requires a lot of domain knowledge and therefore programs and modelling experts will have difficulties to accomplish these tasks. When using a CNL the only stakeholders that have to grasp the meaning and consequences of specification concerns are designated domain experts. Second, like all natural language texts, common building specifications can be open to multiple interpretations especially when implicit knowledge is not transferred from the author to the addressee of a text. In contrast to the second alternative, in which modelling experts interpret text written by domain experts, a solution involving a CNL supports the expert authors of specification concerns in expressing implicit knowledge. Because of this complexity and unexpressed knowledge we are convinced that such transformative approaches are less appropriate for building specifications than an approach that supports the direct creation of controlled instructions.

We argued for a CNL for executable building specifications because of the syntactic regularity of parts of the texts. In the following section we explain why the semantic discrepancy between building specifications and building models demands advanced solutions such as expert-driven semantics.

9.2. Expert-Knowledge as Interpretation Patterns

In order to tackle the semantic richness of building specification concerns and in order to close the gap between building specifications and concepts of building models we propose the use of expert-defined interpretation patterns. Before we present our suggestions in detail we explain the observations that led to them. First, building specifications are characterized by a multitude of domain-specific abstractions that are not available in building models. These abstractions have to be available in a language that expresses specification concerns as transformation instructions for building models. Second, it can be expected that domain experts that formulate building specifications know best about these abstractions and about the effects that specifications phrases should have on

building models even if they may not be able to design an appropriate language. Experts that are more skilled in designing languages probably have difficulties to understand these domain-specific abstractions and effects. Third, the variety of concerns that can be expressed in building specifications requires significant effort when attempting to design a complete language and delays the possibility for an initial evaluation. Therefore, we propose to integrate the domain experts into the design of such a language with an approach that lets them define interpretation patterns similar to these presented by Lugato et al. [LMT+04].

Sentence and interpretation pattern definition workflow

Before we go into details about interpretation patterns we explain their use and definition in the overall process of formulating executable building specification phrases in order to illustrate their purpose. For the time being it is sufficient to know that in our suggested approach interpretation patterns combine a syntactical pattern with the resulting semantics in the form of a weaving aspect. The formulation process for an individual concern and the needed interpretation patterns is depicted in Fig. 9.1 as a UML activity diagram. We will now explain the activities of the diagram individually.

Analyse Initially it has to be analysed whether the building specification concern can be expressed as an instruction for the corresponding building model (Fig. 9.1–1).

Choose If the concern cannot be introduced into the building model, it is formulated in natural language as part of the ordinary specification text (Fig. 9.1–2b).

Formulate If the concern can be represented as model integration instructions because it only relates to model elements, it is formulated using the CNL (Fig. 9.1–2a).

Parse After formulating a CNL sentence, the domain expert directly checks whether this sentence can be parsed (Fig. 9.1–3).

Reformulate If parsing is not possible, the expert has to reformulate the sentence until it contains only keywords and content words of the domain (Fig. 9.1–2a). This process could be supported by an appropriate tool that suggests existing terms and expressions.

Interpret Once the sentence can be parsed the expert trys to interpret it using the existing interpretation patterns (Fig. 9.1–4).

Evaluate interpretation If the semantic of the sentence can be determined using the existing patterns, the domain expert is given the possibility to evaluate the derived interpretation (Fig. 9.1–7). This evaluation could include a preview of the aspect that was generated for the sentence. It could also display the detected join points for the generated aspect and the building model of the current construction project in order to give the expert an idea of the places of application.

Next concern If the domain expert finds that the sentence leads to the intended interpretation, the process for the current concern is over and the domain expert can proceed with the next concern.

Evaluate patterns If the interpretation is insufficient, the domain expert continues with an examination of the existing interpretation patterns (Fig. 9.1–5). This activity also happens in situations in which the sentence can be parsed but not interpreted.

Reformulate If the existing interpretation patterns are sufficient for the expressed concern, the expert should reformulate his sentence (Fig. 9.1–2a). A reformulation is followed by another attempt for parsing.

Define patterns If the expressed concern cannot be interpreted using the existing interpretation patterns, the expert has to define the required interpretation patterns (Fig. 9.1–6). After the definition of a new interpretation pattern the parsing result is still valid and a new attempt for interpreting the sentence is directly performed.

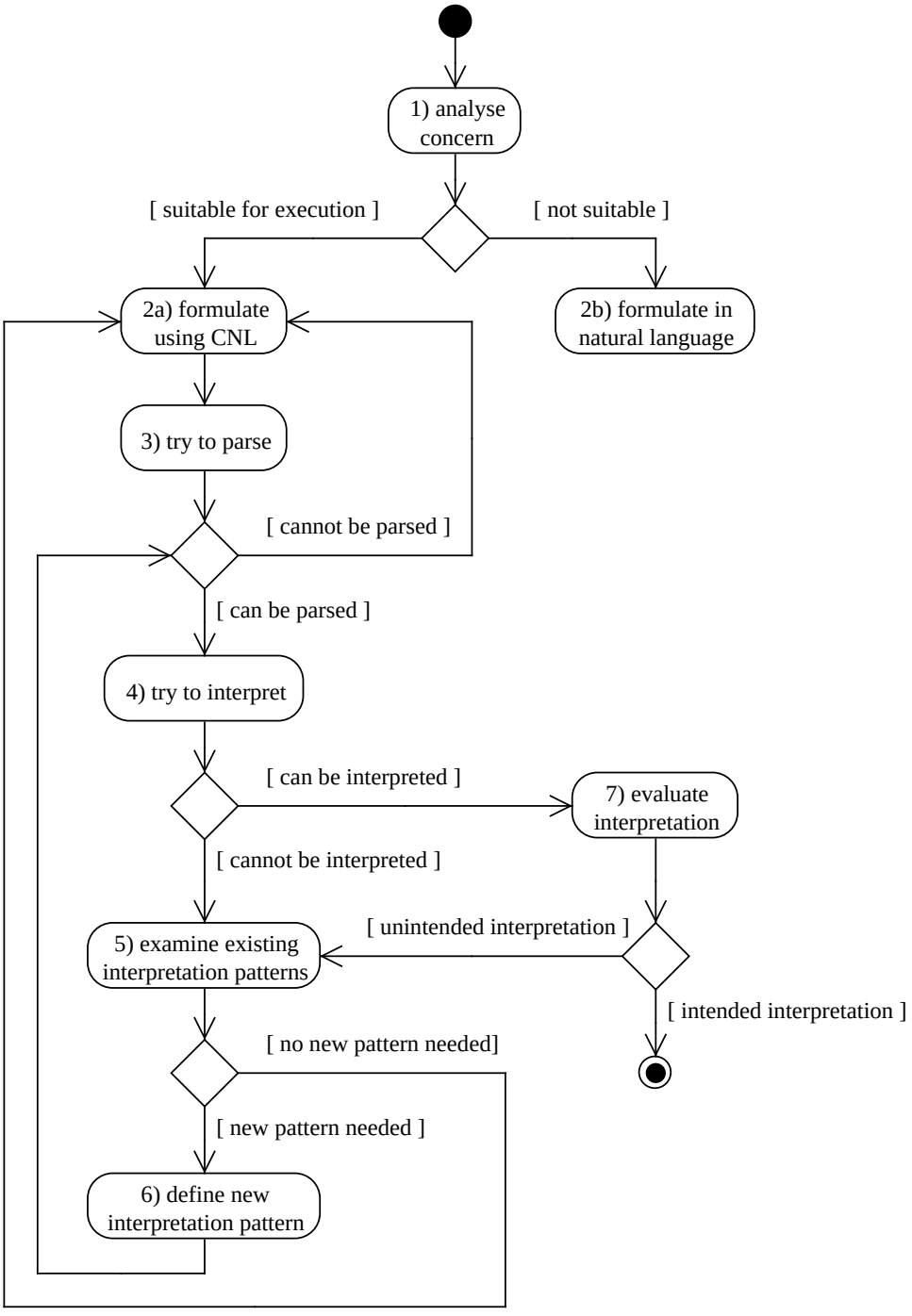


Fig. 9.1.: Activity for formulating, parsing, interpreting, and evaluating a building specification concern using a CNL based on interpretation patterns.

Once the formulation process for a suitable concern has been started it always ends with a CNL phrase that can be interpreted based on the possibly updated pool of interpretation patterns. The overall structure of this process gives domain experts the ability to write executable building specifications incrementally while increasing the expressive power of the language they are using. Note, however, that special attention has to be paid to the composability of newly created interpretation patterns in order to avoid situations in which different interpretation patterns lead to different semantics for the same sentence.

Realising expert-defined interpretation

We suggest to realise an interpretation pattern as an Abstract Syntax Tree (AST) pattern for the building specifications CNL together with an aspect for our model weaving approach (see Part II). The AST pattern should have the possibility to bind values to variables so that these values can be used in the pointcut and advice model of the linked aspect. In this context it may be important to analyse whether it is sufficient to use these values without modifications or not and whether other techniques for the definition of sophisticated interpretation patterns would be needed. Furthermore, the requirements of the AST pattern description language have to be examined. However, during the explorative investigations described in this section we could not focus on these interface questions as we were mainly concerned with the overall feasibility of an approach based on a CNL and interpretation patterns.

Our proposals for the realisation of an interpretation engine for a building specifications CNL are influenced by the way Lugato et al. [LMT+04] realised their interpretation patterns for requirements engineering. In an industrial context they generated test cases from use cases that they obtained by interpreting requirements formulated using a CNL called Requirements Description Language (RDL). RDL did not allow for nested sentences or references to other sentences so that it was possible to interpret each sentence individually. Although domain experts where the ones that suggested interpretation patterns, they were not the ones to actually define their executable semantics. Every interpretation pattern was implemented individually as a routine that compares a given AST with the expected structure while storing matched variable values. To determine whether a given RDL sentence can be interpreted, all existing patterns where tried in a fix order and the overall matching process was stopped as soon as a single pattern matched. The resulting use case was then created using the variable values stored during the processing of the AST of the RDL sentence. Using this approach Lugato et al. were able to interpret 70 % of the requirements of two medium-sized case studies (5 - 15 KLOC each) with 16 individual interpretation patterns.

We suggest a generalisation of the approach of Lugato et al. in order to give domain experts the ability to define the syntax and semantics of their own interpretation patterns. As mentioned before, the definition of the syntactical part of an interpretation pattern would require a language for describing the AST pattern and assigning variables that can be used in the realising aspects. At design time the description of an AST pattern would have to be compiled to code that compares a given AST to it and binds the involved variables. The definition of the semantics of an interpretation pattern, however, could be done using plain aspects for our weaving approach. No compilation of aspects would be needed as it would be sufficient to replace variable placeholders in the pointcut and advice model with the matched values before weaving.

The purpose of our suggestions for a building specifications CNL based on interpretation patterns is the provision of a more detailed context and motivation of our work on weaving building model aspects. In order to render these suggestions more tangible we illustrate the processing of an example sentence in the following section.

9.3. An Exploratory Example

This section presents one of the example concerns of a building specification that we used to partially evaluate our suggestions for a building specifications CNL based on interpretation patterns. We explain the concern, sketch the patterns needed to interpret it, demonstrate their step-wise application and present the obtained pointcut and advice models.

As a full-blown example would not ease but hinder the understanding of our propositions we decided to use a simplified building specification concern formulated as:

> All windows of the ground floor have to be made of high grade steel.

Without any knowledge of the IFC metamodel for building models one may analyse that this example sentence relates to three concepts for building models. First, the main subject of the sentence is the entirety of "all windows". Second, this entirety is restricted using the concept of "ground floor" in order to phrase a necessity for all windows that occur on this floor. Third, the requirement that all these entities are realised using the material "high grade steel" is expressed.

Parsing rules and interpretation patterns

In order to ease the formulation of interpretation patterns that apply to a wide range of formulations we suggest the definition of rules for content words that could also be named parsing rules. Such content word rules (see Fig. 9.2 for examples) would allow for the transformation of convenient formulations into concepts that are used in interpretation patterns. If the content word vocabulary of the CNL would also be implemented using such content word rules this would allow for later customisation of the parsing process. In contrast to interpretation patterns the search pattern of content word rules should not be related to the syntactic structure of the CNL but be expressed on the level of characters respectively strings and not involve variables. Whether appropriate use of such content rules may increase the composability and generality of the used interpretation patterns has to be checked. It could also be of interest whether such rules may help to separate the customisation of the parsing process from the domain experts' definition of the semantics.

The content word rules that would be needed for our running example are presented in Fig. 9.2 in the order in which they can be applied to the example sentence from left to right. Rule I) defines that for every occurrence of the word "window" an instance of the metaclass `IfcWindow` of the IFC metamodel has to be created. The second rule structurally differs from the first one as it specifies that the term "ground floor" has to be replaced by an instance of the metaclass `IfcBuildingStorey` for which the attribute `Elevation` is set to `0`. Rule III) defines that the term "high grade steel" has to be replaced by an instance of the metaclass `IfcWindowStyle` that references the corresponding enumeration literal `HIGH_GRADE_STEEL`. Note that we propose to automatically generate rules like the first and the last one from the IFC metamodel. This would require some derivation logic that includes for example string operations

```
I)   'window[s]' = IfcWindow()

II)  '[the] ground floor' = IfcBuildingStorey(Elevation=0)

III) 'high grade steel' = IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')
```

Fig. 9.2.: Content word rules for window example sentence.

```
1) p:IfcProduct 'of' s:IfcSpatialStructureElement
     ↦
   p s r:IfcRelContainedInSpatialStructure(RelatedElements<-(p), RelatingStructure=s)

2) 'All' p:IfcProduct more1:* 'have to' more2:*
     ↦
   Aspect(Pointcut=p more1, Advice=more1 p more2)

3) o:IfcObject '[be²] made of' t:IfcTypeObject
     ↦
   o t r:IfcRelDefinesByType(RelatedObjects<-(o), RelatingType=t)
```

Fig. 9.3.: Interpretation patterns for window example sentence.

that remove prefixes such as "Ifc" or operations for replacing separation characters such as "_". If the CNL should also allow synonyms or plural forms this content word rule generation would also require some linguistic techniques. But, as the IFC metamodel contains more than 600 metaclasses, these investments can be expected to pay off.

After application of the presented content rules our example sentence reads as follows:

> "All" `IfcWindow()` "of" `IfcBuildingStorey(Elevation=0)` "have to be made of" `IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')`

In this form the sentence can be interpreted using the three interpretation patterns that we present in Fig. 9.3. Note that we display a simplified representation of our interpretation patterns that does not refer to elements of the grammar but directly to concepts of the IFC metamodel in order not to burden the reader with the technicalities of ASTs. This is remarkable as it may be beneficial to let domain experts define patterns in a similar way as they are presumably familiar with the domain metamodel but not with the AST of the CNL. The first interpretation pattern defines that a word sequence that combines an `IfcProduct` with an `IfcSpatialStructureElement` using the word "of" should be interpreted as a relation of type `IfcRelContainedInSpatial-Structure`. This relation should link both model elements of the pattern using the attributes `RelatedElements` and `RelatingStructure`. The second pattern contains the word "All" followed by an `IfcProduct` and some arbitrary elements plus the words "have to" and other arbitrary elements. It states that such a sentence should result in an aspect consisting of a pointcut model and an advice model. The advice model should contain the product and the elements before the string "have to". This advice model should also contain the elements before the string "have to" plus the product plus the elements after "have to". The third interpretation pattern is similar to the first one as it defines an interpretation as a relation. Whenever an `IfcObject` is followed by the words "made of" and a `t:IfcTypeObject` this has to be interpreted as a relation of type `IfcRelDefinesByType`. This relation should link the object and the type using the attributes `RelatedObjects` and `RelatingType`. In order to applicable in different contexts the words "made of" may be preceded by any conjugation of the verb "be".

A possible interpretation

One of multiple possibilities to apply the presented interpretation patterns to our example sentence is shown in Fig. 9.4. Initially all three content word rules are applied so that only the words "All", "of", and "have to be made of" and the model elements `IfcWindow`, `IfcBuildingStorey`, and `IfcWindowStyle` remain. To understand the following applications of interpretation patterns some further type information for these model elements is needed. According to the inheritance relations of the IFC metamodel every instance of the metaclass `IfcWindow` is an `IfcProduct` and every instance of the meta-

```
'All windows of the ground floor have to be made of high grade steel.'

     apply I, II, and III
     ------------------->

'All' IfcWindow() 'of' IfcBuildingStorey(Elevation=0)
  'have to be made of' IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')

     apply 1
     ------->

'All' p:IfcWindow() s:IfcBuildingStorey(Elevation=0)
  IfcRelContainedInSpatialStructure(RelatedElements<-(p),RelatingStructure=s)
  'have to be made of' IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')

     apply 2
     ------->

Aspect(
  Pointcut = p:IfcWindow() s:IfcBuildingStorey(Elevation=0)
             IfcRelContainedInSpatialStructure(RelatedElements<-(p),RelatingStructure=s)
  Advice = s:IfcBuildingStorey(Elevation=0)
           IfcRelContainedInSpatialStructure(RelatedElements<-(p),RelatingStructure=s)
           p:IfcWindow() 'be made of' IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')
)

     apply 3
     ------->

Aspect(
  Pointcut = p:IfcWindow() s:IfcBuildingStorey(Elevation=0)
             IfcRelContainedInSpatialStructure(RelatedElements<-(p),RelatingStructure=s)
  Advice = s:IfcBuildingStorey(Elevation=0)
           IfcRelContainedInSpatialStructure(RelatedElements<-(p),RelatingStructure=s)
           p:IfcWindow() t:IfcWindowStyle(ConstructionType='HIGH_GRADE_STEEL')
           IfcRelDefinesByType(RelatedObjects<-(p),RelatingType=t)
)
```

Fig. 9.4.: Application of interpretation patterns for window example sentence.

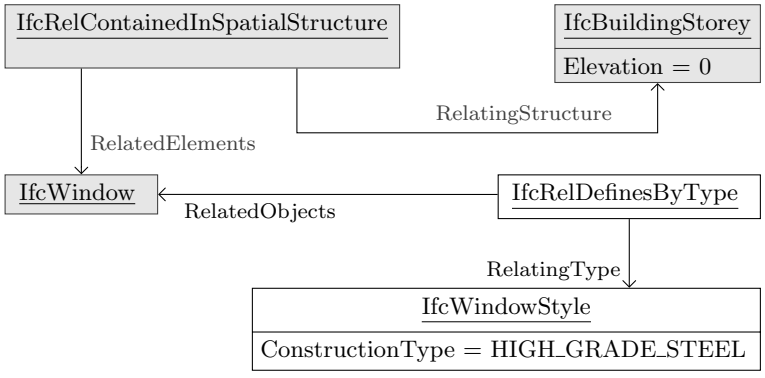


Fig. 9.5.: Resulting aspect for window example sentence (pointcut in grey).

class `IfcBuildingStorey` is an `IfcSpatialStructureElement`. Therefore it would be possible to apply interpretation pattern 1) as well as interpretation pattern 2). In our example we continue with pattern 1) thus replacing "of" with an instance of the relation metaclass `IfcRelContainedInSpatialStructure` that links the elements that represent the window and the storey. The following application of interpretation pattern 2) replaces the overall sentence structure with an aspect for our weaving approach. In this aspect the pointcut model contains all model elements that correspond to the words that occurred before "have to". The advice model of the aspect contains the same elements plus the words "be made of" and the `IfcWindowStyle` model element, which corresponds to the part of the sentence after "have to". Due to the inheritance relations every `IfcWindowStyle` is an `IfcTypeObject` and every `IfcWindow` is an `IfcObject`. Hence, interpretation pattern 3) can be applied: The remaining words "be made of" are replaced with an instance of the relation `IfcRelDefinesByType` that links the window and its style. As a textual representation may impede understanding we depicted the resulting weaving aspect also visually in Fig. 9.5. To avoid repetition we combined the visualisation of the pointcut and advice model while marking model elements that are part of the pointcut but not part of the advice model in grey.

9.4. Summary

In this chapter we outlined a possible way for obtaining model weaving aspects for building specification concerns using a CNL based on interpretation patterns. We explained the semantic gap between building specifications abstractions and abstractions present in building models and showed how this gap could be closed using an expert-defined specification language. The suggested approach was illustrated with a step-wise explanation of a possible parsing and interpretation process for a concrete example. How the obtained aspect can be woven into building models is explained in the next chapter.

10. Weaving Building Specification Aspects

In this chapter we explain how building specification aspects can be woven into building models using a customisation of our generic model weaver. First, Section 10.1 describes the extensions that customise our prototype implementation in order to account for the specifics of BIM models. Then, the effects of these extensions are illustrated in Section 10.2 using the aspect that we obtained from the building specification example of the previous chapter.

10.1. Extending our Generic Weaver

We customised the prototype implementation of our generic model weaving approach for IFC building models using five extensions. The first extension affects a non-recurring preparatory step and the four other extensions belong to the weaving process for individual models. These four extensions are depicted in Fig. 10.1 together with the corresponding extension points described in Section 4.2.1 and the affected weaving phases. They ensure that the specific properties of IFC building models are taken into account during the individual steps of weaving. We will now present the individual extensions in the order that they are used during the weaving process.

The two IFC building models extensions that are used first during the weaving process are necessary because we cannot use default XMI serialisation for these models. EMF (see Section 6.1.1) provides a default serialisation for models conforming to metamodels defined in the metamodelling language Ecore. Our prototype implementation uses these serialisation facilities to load and store models. Building models specified in the IFC format, however, are serialised as ASCII text files. Using the technological bridge by Steel et al. [SDD11] (see Section 2.4) these building models can be loaded as instances of an Ecore metamodel for IFC. As the implementation for instances of this metamodel and the loading and storing procedure differ from the default metamodel code and default loader, we provide two extensions to account for these specifics.

The first extension extends EP 1 in order to ensure that the code generated for the relaxed metamodel variants exhibits the same properties as the code for the original IFC metamodel. It propagates the properties of the bridge's code generator model into the code generator models for the relaxed pointcut and advice metamodel variants. An example of such a property is the superclass relationship, which specifies that every

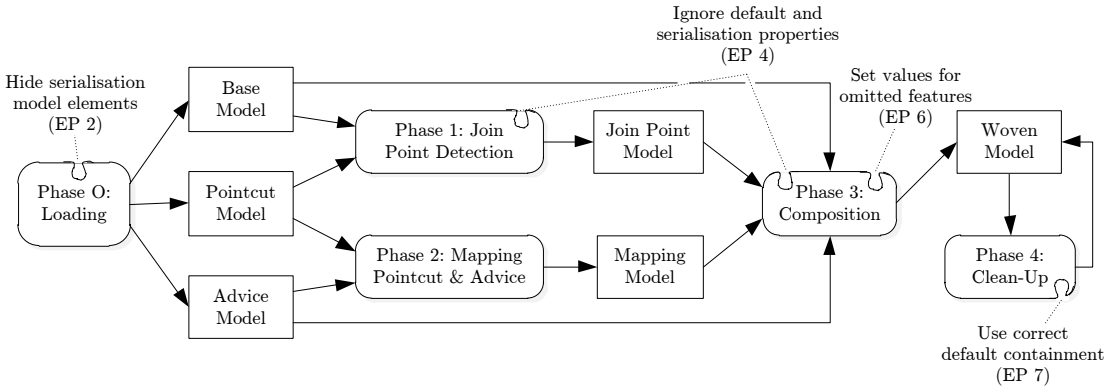


Fig. 10.1.: The extensions of the IFC customisation for our weaver and the influenced weaving phases and models.

implementation class for a metamodel element extends a class of a library that was used to implement the technological bridge.

In the same serialisation context we provide a second extension, which extends EP 2 in order to customise the loading and storing processes for building models. It provides a customised resource loader taking into account that actual building model elements are wrapped into elements that model the serialisation format. This is necessary, for example, in order to hide the elements that model the serialisation when an iteration over all contents of a model is performed. These serialisation model elements and the actual building model elements are instances of the Ecore metamodel for IFC. Therefore, we do not have to customise all loading and storing infrastructure but can reuse most of the generic facilities provided by GeKo and EMF in this IFC extension.

A third extension, which extends EP 4, makes sure that the weaver ignores specific values of metaclass features during join point detection and model comparison. Specifically, when creating a pointcut model and specifying that a feature's value is not significant, it is important to avoid the case where the implementation for the IFC metamodel automatically applies the default value for this feature. This ensures that a pointcut model that does not specify a value for a certain feature matches all values for the feature and not only the default value. Furthermore, certain features, such as globally unique identifiers, that are only necessary for the serialisation of IFC elements, should be ignored during the comparison of woven models. This ensures that model elements can be considered equal even if they exhibit different global identifier values because they belong to different models. When woven models are compared during regression testing or during the creation of new aspects, this possibility for semantic comparison is particularly useful.

EP 6 is extended in a fourth extension, which takes care of features that are required in the model but inconvenient to express in the aspect. While an advice model element is copied for inclusion into the woven model the extension ensures that default values for these omitted features are set. An example for such a feature omitted in aspects is the owner history, which has to be set for every element of an IFC model. This feature references the person responsible for making changes to the building model. All elements introduced during the weaving have the same value for this feature because the same person is responsible for all these elements. Therefore, omitting the owner history feature in pointcut and advice elements and adding it automatically during the weaving is more convenient than mentioning it for every individual element. Another feature that is set by this extension is the globally unique identifier, which we already

mentioned for the previous extension. When advice model elements are added to a woven model we cannot use the identifier that was used for the advice model because this could lead to collisions in which two elements share the same identifier. Instead, a new identifier is calculated and set for such newly introduced elements in a way that ensures that no other elements of the woven model have the same identifier.

Our fifth and last extension for IFC building models applies at the very end of the weaving process and overrides default behaviour for EP 7. This extension is responsible for determining the correct containment reference for elements that are added to the woven model during the weaving. If such an element is not implicitly contained in a building element as a result of the references specified in the advice model, it has to be added explicitly by the weaver. Per default, the containment reference for this explicit addition is chosen using the best-effort algorithm presented in Section 7.3.4. This metamodel-independent algorithm does not yield the correct reference for IFC building models. Therefore, the extension has to use its knowledge of the IFC metamodel to return the correct containment reference. Let us use this extension to illustrate an advantage of our approach resulting from the decision to support pointcut and advice definition using model snippets: IFC building models may exhibit deeply nested hierarchies. A window, for example, may be indirectly contained in a hierarchy of containers starting with a storey and spanning over a building container, a building, a site container, a site, a project container, and a project. If pointcut and advice models could not directly contain model elements that occur anywhere in this hierarchy, this would mean that every pointcut and advice model would need to specify the whole hierarchy beginning with the building project. Our approach, however, makes it possible to add typically nested building elements directly at the first level of pointcut and advice models. If new elements are added during the weaving, we use information available at the join points to hook these new elements into the containment hierarchy of the nested building elements. For elements for which no containment information is available at the join point our extension ensures that these elements are correctly added to the hierarchy.

We implemented the five presented IFC building model extensions for our generic model weaver in less than 10% of the lines of code needed for the complete generic implementation[1]. This result and the practical experience of providing a set of domain-specific extensions to our own generic approach makes us confident that this strategy is generally suitable for modifying domain-specific models. The fact that only little additional work was needed to customise the weaver for models with outstanding specifics suggests the conclusion that applying our generic approach requires less effort than the development of domain-specific model weavers. This, however, needs to be confirmed by future experiments that involve new extensions for models of other domains.

10.2. An Illustrative Example

Let us illustrate the presented extensions for IFC building models using the building specification example sentence of the previous chapter.

> All windows of the ground floor have to be made of high grade steel.

The result of parsing and interpreting this sentence using a CNL for building specifications is the aspect depicted in Fig. 9.5 on page 85. Its pointcut model contains two objects representing a window and a storey and a spatial containment relation for these two objects. The advice model repeats these objects as well as the relation and lists a new object representing the style of a window together with a type definition relation

[1] 367 lines of program logic code used for the extension and 5,083 lines of generic program logic.

linking this style object with the window object. We will now detail how the individual extensions for building models affect the weaving of this aspect.

At the beginning of the weaving the base model is loaded using the customised resource loader provided by the extension to EP 2. Thanks to the technological bridge for IFC and EMF, the base model, which is serialised as an ASCII text file, appears as an instance of the Ecore metamodel for IFC. The elements that model the serialisation format are skipped by the resource loader of the extension when the `IfcProject` root element of the building model is requested for join point detection.

The other model influencing the process of join point detection is the pointcut model, which is serialised using the XMI default, and therefore can be loaded using the standard resource loader. The implementation classes for the pointcut-model elements were generated according to the customisations for the generator model extension point EP 1. Therefore, the objects representing base and pointcut elements share the same structure despite of their different serialisation techniques. This common implementation base makes it possible to directly compare elements of the pointcut model with base-model elements. Once the pointcut model elements where loaded, they are used to generate the join point detection rules. The extension to EP 4 ensures that the features for which no values were specified for the three pointcut elements `IfcRelContainedInSpatialStructure`, `IfcBuildingStorey`, and `IfcWindow` are ignored during this generation. This guarantees that all windows of the ground floor are detected regardless of their values for the features that are purposely omitted in the pointcut.

After successful join point detection, the objects `IfcRelDefinesByType` and `IfcWindowStyle`, which are only part of the advice model, have to be added at each join point. While these elements are copied for inclusion in the woven model the extension to EP 6 ensures that features that were omitted in the advice model are completed before they are added to the woven model. For the object `IfcWindowStyle` and its relation `IfcRelDefinesByType` the default owner history is added and new global unique identifiers that do not collide with the identifiers of the woven model are calculated.

The last weaving step that is affected by the building models customisations is the handling of containment. Both new elements `IfcRelDefinesByType` and `IfcWindowStyle` are first level members of the advice model and not implicitly contained in other advice-model elements. This means that they have to be explicitly added to the containment hierarchy of the woven model. The extension to EP 7 returns the correct containment reference so that both are added to the woven model's root element `IfcProject` using the correct containment reference named `contents`. No clean-up step is required because no elements were removed during the weaving. This makes the addition of the two new elements the last step of weaving for our example.

10.3. Summary

In this chapter we showed how our generic model weaving approach can be used to weave aspects representing building specification concerns. First, we presented the individual extensions that complete or change the generic weaving behaviour in order to account for the specifics of building models. Then, we illustrated how these extensions affect the individual steps of the weaving procedure using an exemplary building models aspect.

The end of this chapter is also the end of Part IV, which presented building models and specifications as an application for our generic weaving approach. In the following, last part we complete this thesis by discussing related work, summing up its contributions and providing an outlook on future work.

Part V.

Postlude

11. Related Work

This chapter discusses approaches related to model weaving, domain-specific model transformations, and building specification integration.

11.1. Model Weaving Approaches

Since AOM and MDE emerged as software engineering principles the research community investigated various ways to incorporate cross-cutting concerns directly into models. Numerous approaches in this area focus on specific modelling languages and design processes. In this section we present a selection of approaches that we consider relevant for this thesis because they had a great influence on the community or are similar to our work. In contrast to our approach, all approaches presented in this section provide no or only very basic possibilities for domain-specific extensions. Thus, we decided not to mention this difference for each and every approach.

11.1.1. Aspect-Oriented Analysis and Design: The Theme Approach

The *Theme* approach is one of the first methodologies for aspect-oriented design and analysis that provides concepts to handle cross-cutting concerns as first-class entities and defines composition operators. It consists of two parts. The first part, Theme/Doc, was developed in order to identify cross-cutting requirements and for mapping them to an appropriate design using the second part, Theme/UML. This second part is realised as an extension to the UML metamodel and supports symmetric and asymmetric composition. Being a pure design approach, Theme/UML provides no tool support for model composition and has only lately been integrated into a transformative tool by Carton et al. [CDJC09]. This lack of automated composition and the restriction on UML models are the two main differences compared to the approach presented in this thesis.

11.1.2. WEAVR: Model Weaving in a Large Industrial Context

To our knowledge, the Motorola *WEAVR* approach by Cottenier et al. [CvdBE06] is the only model weaving approach that is used in a productive industrial setting. It is realised as a profile for the UML and provides weaving support for executable statecharts with action semantics. Aspects are defined using stereotyped pointcut and advice models and they apply at two types of join points: actions and transitions. In

contrast to our approach, the Motorola WEAVR is specialised for a specific modelling language and provides elaborate tool support at a high level of maturity. An academic license can be obtained free of charge.

11.1.3. AMW: Atlas Model Weaver

Despite its name, the Atlas Model Weaver (AMW) by Didonet et al. [FBV06] was originally developed to establish links between models. These links are called *weaving models* and have to be created in a semi-automatic manner using an interactive tool. They can be used for many tasks such as comparing, tracing, or matching different models with related elements. It would also be possible to use these links for transformations that weave elements into a model instance. Published applications of the AMW, e.g. by Didonet and Valduriez [DDFV09], mainly use the links for heterogeneous model transformations and model comparison. A use case example that can be found online[1] describes how the AMW can be used to add attributes to metaclasses of a metamodel using the links of a weaving model. In this use the weaving model needs to define the join points as no pointcut definition and join-point detection mechanism is provided. This lack of automation and the design for a different purpose are the two main distinguishing features between the AMW and our approach.

11.1.4. Kompose: Directives for Composing AOD Class Models

The *Kompose* approach by Reddy et al. [RGF+06] supports name-based an signature-based merging of class diagrams. Aspect models describe cross-cutting features and have to be instantiated manually before they can be symmetrically composed with a primary model. To identify elements to be merged their names or signatures are compared. In this approach, a signature is a collection containing the values for a subset of the properties defined in the metamodel for the element's metaclass. An outstanding feature of *Kompose* is the use of composition directives to customise parts of the weaving. This customisation is done by defining the order in which classes are merged or explicit actions that have to be carried out before, during, or after the merge. *Kompose* provides conflict detection mechanisms but lacks full tool support, for example, regarding the extension possibilities. This deficit, the missing join point detection and the restriction to class diagrams are the main differences to our approach.

11.1.5. SmartAdapters: Towards a Generic AOM Framework

SmartAdapters is one of the model weaving techniques that shares many concepts and solutions with the approach presented in this thesis. In the initial version of SmartAdapters, join points had to be specified manually and the approach had to be tailored to specific metamodels. After an application to Java Programs by Lahire and Quintian [LQ06] and a further application to Class Diagrams by Lahire et al. [LMV+07], the approach has been generalised by Morin et al. [MBJR07] to support arbitrary metamodels. Later efforts, however, focused on the use of SmartAdapters for adapting component models at runtime. As a result initially generic weaving functionality cannot always be separated from advanced concepts for component based systems. Nevertheless, some of the features of SmartAdapters were reused in the approach presented in this thesis. The join point detection mechanism, for example, is built on top of the same platform. Furthermore, the different strategies for advice instantiation described by Morin et al.[MKKJ10] were applied to SmartAdapters before we used them in our approach. Why we realised these two features separately and differently is explained in Section 5.1 and 5.3.2.

[1] eclipse.org/gmt/amw/usecases/AOM

A major difference between SmartAdapters and our approach lies in the representations that are used to express weaving instructions. In addition to a declarative pointcut and advice model, SmartAdapters needs an imperative description of how an aspect should be woven. This *composition protocol* makes it possible to define sophisticated weaving operations that cannot be expressed with the pointcut-to-advice mapping of our approach. But it also requires that even basic weaving tasks have to be defined in explicit detail, which may represent a barrier to the adoption of SmartAdapters. Apart from the extension possibilities, which all approaches lack, these imperative weaving instructions are the main difference between SmartAdapters and the approach presented in this thesis.

11.1.6. XWeave: Models and Aspects in Concert

XWeave is a generic approach by Groher and Voelter [GV07] that can be used to weave Ecore metamodels and their model instances. It was developed in the context of software product line engineering and supports weaving according to product line configurations i.e. selected features. Aspect models for this asymmetric approach are ordinary models that may contain wildcard names as well as pointcut expressions. These pointcut expressions are defined using a language based on the Object Constraint Language (OCL) and may, for example, contain path expressions and collection filters. XWeave only supports additive weaving because existing model elements cannot be changed or removed. This deficiency can be partially corrected using the related tool *XVar* by Groher and Voelter [GV09]. XVar focuses support for Software Product Lines (SPLs) and code generation. It can be used for negative variability as it is able to delete code-level elements that correspond to unselected features. Because this is not the same as removing or changing existing model elements the restriction on additive weaving is the main difference to our approach.

11.1.7. MATA: Modeling Aspects Using a Transformation Approach

MATA is an approach by Whittle and Jayaraman [WJ08] that supports generic model weaving based on graph transformations. It converts a base model into an attributed type graph, applies graph rules obtained from composition rules, and converts the resulting graph back to the original model type. The rules for the asymmetric composition are defined as left-hand-side (pointcut) and right-hand side (advice) but can also be expressed in a single diagram. Although the approach is conceptually generic we are only aware of an application to UML models in which composition rules are defined using the concrete syntax of the UML. In this application an aspect is defined using a UML profile and stereotypes are used to mark elements that have to be created, matched, or deleted. Because model instances of Ecore metamodels can be seen as a special case of attributed type graphs, MATA's weaving operator is similar to the one used in our approach. The explicit transformation to graphs, however, results in two major differences: In contrast to our approach, MATA does not work with aspects defined using the language of base models and it does not directly operate on the input models.

11.1.8. Almazara: AOM Weaving Beyond Model Composition

Almazara is a model weaver presented by Sánchez et al. [SFS+08] that generates model transformations from Join Point Designation Diagrams (JPDDs). These JPDDs can be defined using an UML Profile and support various selection criteria, such as indirect relationships or regular expressions for element names. Model transformations that realise the weaving are generated using graph-based model transformation templates. This is very different from classic AOM approaches that compare pointcuts to base

models and replace matched snippets. Furthermore, the obtained transformations are not only used to find join points. They are also responsible for collecting runtime information and evaluating dynamic selection constraints. The authors of Almazara state that JPDDs can be used with any modelling languages, but we are not aware of any applications to non-UML notations. This language restriction and the use of JPDDs are the main differences between Almazara and the approach of this thesis.

11.1.9. Aspect KerTheme: Composing Multi-View Aspect Models

Barais et al. [BKB+08] presented a weaving approach for an extension to the Theme/UML approach named *KerTheme*. Models of this approach contain two views for structural and behavioural modelling: Executable Class Diagrams (ECDs), which are based on UML class diagrams, and scenario models, which are analogue to UML sequence diagrams. For these modelling languages *Aspect KerTheme* supports two types of composition: The symmetric composition of two base models, called merge, and the asymmetric composition of a base and an aspect model, called weaving. Regarding structural models, Aspect KerTheme shares some ideas and features with Kompose (see Section 11.1.4). In both approaches elements are matched by name or signature and in both approaches asymmetric structural composition is reduced to symmetric compositions through conversions. Behavioural models, however, are woven using a semantic weaver by Klein et al. [KHJ06], which is specialised on scenario models. The restriction on specific modelling languages and the lack of tool support are, as so often, the main difference to our approach.

11.1.10. GeKo: A Generic Weaver for Supporting Product Lines

The weaving approach described by Morin et al.[MKBJ08] is the predecessor to the approach described in this thesis. Various concepts that are central to the original approach still play an important role in the successor. Others parts have been replaced, improved, simplified, or added.

We replaced the Prolog-based join point detection mechanism of the initial version with the approach of SmartAdapters, which uses the Drools business logic platform. This switch in technology made it easier for us to render the process of join-point detection more independent from individual modelling language: In the previous version of GeKo each specific metamodel had to be taken into account during the creation of a knowledge base for join point detection. This is not the case for the rules and the knowledge base that are generated for join-point detection in the approach presented in this thesis. Both, the rules and the knowledge base, are independent of metamodels because they are generated in a generic way and directly reuse the available implementation of a metamodel.

Relaxed metamodel variants as described by Ramos et al. [RBJ07] are used in the old and new version of GeKo. These relaxed metamodels are used for the definition of pointcut and model snippets, which do not need to fulfil all the constraints of metamodels used for base models. While metamodels had to be manually relaxed in the initial version of GeKo, they are automatically derived in our enhanced weaver.

In the previous version of GeKo the declarative mapping from pointcut to advice elements always had to be defined manually. The enhanced approach, however, provides an algorithm that can automatically infer this mapping in case of unambiguous correspondences. In order to compose models based on this pointcut-to-advice mapping, the initial version of GeKo used an elaborate set-theoretic formalisation. For the enhanced version we slightly simplified this formalisation and added support for mapping situations that relate multiple elements to single elements and vice-versa.

The last difference between the two versions of GeKo is the support for different advice instantiation strategies, which we added to the enhanced approach following the ideas of Morin et al.[MKKJ10]. Altogether, the comparison of the initial and enhanced approach boils down to various detail improvements, better tool support, increased genericity, and the support for domain-specific extensions.

11.1.11. Tree Based Domain-Specific Mapping Languages

Kalnina et al. [KKS+12a] presented an approach for domain-specific model transformations based on mappings between model trees. Although this approach does not explicitly address cross-cutting concerns in the sense of Aspect-Orientation, it could be used to weave models. In the approach graphically defined mappings specify correspondences between source and target models. These mapped models are based on tree types, which can be seen as a lightweight variant of metamodelling that is based on the observation that model structure is usually determined by containment trees. Two of these model trees are combined in a mapping diagram with named relations. One tree acts as source tree specifying model elements to be mapped, and the other tree serves as target tree specifying the assignment of values and the assembly of newly created nodes. The mapping relations are executed with create-if-not-exists semantics in top-down order of their occurrence in the source tree. This ensures that elements that contain other elements are processed before their children are processed.

Apart from the different design purpose the mapping approach by Kalnina et al. differs from our approach with respect to the notation in which changes are defined. In contrast to our approach, it is not possible to reuse exactly the same notation that has been used for ordinary models. As a result, the approach has to be adapted for new metamodels no matter whether customised behaviour is desired or not.

11.1.12. General-Purpose Model Transformation Languages

The Epsilon Merging Language (EML) [KPP06] can be used to merge heterogeneous models based on their type using a textual syntax that is similar to declarative model transformation languages like QVT-R. These languages support various transformation scenarios and are not specialized for in-place asymmetric homogeneous weaving according to property-based conditions. As a result basic weaving operations, such as the merge of two instances of the same metaclass, have to be redefined for every domain-specific application. Additionally, general-purpose transformation languages can be disadvantageous for detecting join points: Although affected elements are automatically identified, it is usually more complicated to express the corresponding conditions compared to AOM approaches. The same applies for advanced concepts such as different strategies for advice instantiation. Another disadvantage of general-purpose transformation languages is that users are forced to master a new, yet powerful, language although they might only need small parts of it. This is not true for those weaving approaches that let users express aspects using the same notation as for base models.

11.1.13. Higher-Order Transformations

Some of the downsides of general-purpose model transformation languages can be mitigated using advanced transformation approaches such as Higher-Order Transformations (HOTs). These transformations can be used to adapt generic transformation patterns to domain models and to specific points of application. Kápova and Reussner [KR10], for example, used this technique to complete component models with information for performance predictions. In their approach, the obtained customised model completions refined annotated elements according to the configuration specified in these annotations.

Weaving Approach	Join Point Detection	Model-independent	Metamodel-independent	Notation Reused	Declarative Instructions	Instantiation Scope	Extensible	Tool Support	Available	Mature
Theme/UML – Clarke and Baniassad [CB05]	✓	x	x	∼	✓	x	x	∼	✓	✓
WEAVR – Cottenier et al. [CvdBE06]	✓	x	x	∼	✓	x	x	✓	∼	✓
AMW – Didonet et al. [FBV06]	x	✓	x	∼	✓	x	∼	✓	✓	✓
Kompose – Reddy et al. [RGF+06]	x	x	x	✓	✓	x	∼	∼	x	∼
SmartAdapters – Morin et al. [MBJR07]	✓	✓	∼	✓	x	✓	x	✓	✓	∼
XWeave – Groher and Voelter [GV07]	✓	✓	∼	✓	✓	x	x	✓	x	∼
MATA – Whittle and Jayaraman [WJ08]	✓	✓	∼	x	∼	x	x	✓	x	∼
Almazara – Sánchez et al. [SFS+08]	✓	∼	∼	∼	✓	x	x	✓	x	∼
Aspect KerTheme – Barais et al. [BKB+08]	✓	x	x	∼	✓	x	x	∼	x	∼
GeKo – Morin et al. [MKBJ08]	✓	✓	∼	✓	✓	x	x	∼	∼	∼
Tree Mappings – Kalnina et al. [KKS+12a]	✓	✓	∼	x	✓	x	∼	✓	x	∼
Transformation Languages: QVT-R / EML	∼	✓	x	x	✓	x	x	✓	✓	✓
HOTs – e.g. Kápova and Reussner [KR10]	∼	✓	x	x	✓	x	x	✓	x	∼
GeKo – as presented in this thesis	✓	✓	∼	✓	✓	✓	✓	✓	✓	x

Table 11.1.: Comparison of features for model weaving approaches. GeKo's unique extensibility feature and highly distinguishing features are marked in grey.

This is different from the automated join point detection process of our approach as the base models have to be prepared in advance. It facilitates, however, the development of refinement product lines that reuse transformations or parts of it.

11.1.14. Comparison

We summarise the results of the comparison between our approach and other model weaving and transformation approaches in Table 11.1. It contains a row for every approach that we presented above and has columns for features that we decided to compare. When an approach exhibits a certain feature the cell corresponding to the row of the approach and to the column of the feature contains a checkmark (✓). A tilde (∼) is used when a feature is not fully supported or when according information was not provided or questionable. Features that are not supported are marked with a cross (x).

The approach presented in this thesis is the only one to support various extensions at all stages of weaving. Therefore, we highlighted this unique extensibility feature by colouring the background of the complete column in grey. Three other features that distinguish our approach from most of the presented approaches are the reuse of existing notation for aspects, the support for different instantiation scopes, and the free availability of a tool. Where these features are provided we marked the corresponding cell in grey for faster comparison.

Note that our selection of existing approaches and comparison criteria is not suited for a complete comparison of AOM approaches but reflects the goal to demonstrate differences between our and other approaches. We ignored various features that lie outside the scope of our work, such as sophisticated specialisation for certain notations, run-time weaving, or composition according to product-line features. Surveys on approaches for Aspect-Oriented Software Development (AOSD) and model weaving were conducted by Filman et al. [FECA04], Chitchyan et al. [CRS+05], Schauerhuber et al. [SSK+07], and Wimmer et al. [SSK+07].

11.2. Domain-Specific Approaches

In Chapter 9 we proposed a Domain-Specific Model Transformation Language (DSMTL) for weaving building specification information into building models. In the following we briefly discuss some related approaches to domain-specific model weaving and DSMTLs.

Gray et al. [GBN+03], for example, presented *An Approach for Supporting Aspect-Oriented Domain Modeling* using a Constraint-Specification Aspect Weaver (C-SAW). Although the C-SAW is based on the Generic Modeling Environment (GME), a language and tooling framework similar to EMF, it does not compose models. In this approach, domain-specific weavers are generated by integrating domain-specific descriptions and strategies into a meta-weaver framework. The obtained weavers traverse models in order to select elements and apply changes directly on these elements.

Reiter et al. [RKR+06] provided *A Generator Framework for Domain-Specific Model Transformation Languages*. Their framework can be used for realising text-based transformation languages using EBNF grammars. A DSMTL that uses this approach is implemented using source-to-source transformations that target a general-purpose model transformation language in order to provide users the possibility to analyse and customise the obtained code. To this end, the framework assembles templates that were defined by the language designer using the general-purpose transformation language.

Reinhartz-Berger [RB09] presented an AOSD approach for *Domain Aspects*. It separates systems into three layers in order to foster domain-specific adaptations: application, domain, and language. The intermediate domain layer is specified in terms of common features and allowed variability as a SPL and the lowest language layer contains metamodels of modelling languages. So far, the tool for this approach only supports checks for completeness and correctness and thus cannot be used for automated weaving.

11.3. Approaches to Enriching Building Models

In Chapter 10 we proposed to enrich building models with building specification information using a customisation to our generic weaver. Let us now discuss other approaches to integrating such information directly into building models.

11.3.1. A DSMTL for Quantity Take-Off

Steel and Drogemuller [SD11] presented a technique for increasing the efficiency of the construction industry's cost estimation process. This estimation of the overall costs of a building is an important step during the design and it usually consists of two parts. First a quantity surveyor estimates the quantities of building elements and the amount of required work by analysing the building model and the building specification text. This information is stored in a so called bill of quantities. In a second step, the quantity surveyor applies unit rates to these quantities in order to estimate the overall cost of

construction. Steel and Drogemuller presented a tool that helps quantity surveyors during the first step of cost estimation, which is also called *quantity take-off*.

In this approach, an IFC building model is converted to a simplified version that considers less elements and has a flattened type hierarchy. Based on this simplified model, users define take-off rules that specify which elements and quantities should be added to the bill of quantities in which situations. Because these rules are defined using a mixture of textual syntax and tabular representation the users do not need to know that they are using an DSMTL.

This DSMTL for building models inspired the solution proposed in this thesis but also exhibits several differences. We proposed to define a language that modifies building models that were defined in the IFC format. The approach of Steel and Drogemuller, however, used a non-IFC building model obtained after a conversion from IFC to modify a bill of quantities model. This means that, in contrast to our work, the source metamodel and target metamodel of the transformation were different from each other and different from the metamodel we use. Furthermore, the approach by Steel and Drogemuller used a custom transformation engine, whereas we used a customisation to the generic model weaver presented in this thesis.

11.3.2. Industrial CAD and Specification Tools

The problem that textual building specification information is not added to building models is also addressed by commercial CAD tools. The main difference between the efforts conducted in the construction industry and our approach is that we tackle these cross-cutting concerns using concepts of AOSD.

According to [ETSL11], most of the widely used CAD tools, only support cross-links between building specification texts and building models. They allow users to see which elements of the model are affected by which parts of the specification and vice-versa. But, in contrast to our approach, these tools do not actually transform the model. An example of such a tool, which even shows potential conflicts between specifications and models, is BSD LinkMan-E[2].

Some tools for building specifications, such as NBS Create[3], provide possibilities to export specification information into IFC format. Unlike our solution, this approach cannot be used to introduce specification information at multiple regions of an IFC model based on the properties of these regions.

The building SMART alliance®, a council of the National Institute of Building Sciences, runs a project on Specifier's Properties Information Exchange (SPIE)[4]. To our knowledge, the definition of a new format or language for building specifications is not a goal of this project. Thus, our proposal for a constrained natural language for building specifications remains untangled. It seems more likely that the project will result in a set of product templates based on the IFC format. These templates could allow specifiers to define that all objects of the same type have a certain property. But so far no information is available that shows that these templates will be aware of any sort of context or object-individual properties.

11.4. Summary

In this chapter we discussed related work on generic model weaving, approaches to domain-specific transformations, and possibilities to integrate building specification information into building models. The comparison of our model weaving approach with

[2]BSD LinkMan-E - links models from Autodesk® Revit to a specification: bsdsoftlink.com/linkman
[3]NBS Create - A specification tool offering IFC export: thenbs.com/products/nbsCreate
[4]SPIE project: buildingsmartalliance.org/index.php/projects/activeprojects/32

previous work showed that the biggest differences can be observed in terms of extensibility and genericity. It also demonstrated that the approach presented in this thesis was not designed from scratch but combines features of various previously presented model weavers. The discussion of other concepts, tools, and languages for domain-specific transformations showed that this field gets comparatively little attention. This might also be true for the problem of enriching building models with specification data, which is only partially solved and addressed by the tools used in construction industry.

12. Conclusions & Future Work

In this last chapter we summarise the contributions of this thesis and present an outlook on some of the resulting possibilities for future work.

12.1. Conclusions

This thesis presented the concepts and implementation of a generic approach to model weaving as well as a possible use of this approach for building models. Based on previous work on model weaving, we described a weaver that combines metamodel-independent core operations with extension possibilities in order to achieve practical genericity.

In the first part of this thesis we presented our approach conceptually. First, we showed how aspects can be expressed using familiar syntax with pointcut and advice snippets that are defined as instances of automatically derived, relaxed metamodel variants. Afterwards, we presented how genericity and extensibility are achieved through operations that are strictly formulated according to the features of metamodelling languages and through numerous extension points for all phases of weaving. Then, we discussed how affected elements can be automatically detected using the metamodel-defined properties of pointcut elements. Next, we provided insights into the declarative mapping from pointcut to advice elements, which induces all weaving operations and renders imperative weaving instructions unnecessary. Thereafter, we presented the central composition phase of our approach, its set-theoretic formalisation, different possibilities of advice instantiation and challenging weaving scenarios requiring merge and duplication operations. Last, we explained how metamodel conformance is ensured during the removal of elements.

We presented the prototype implementation of our generic model weaver in the second part of this thesis. First, we explained the major design decisions, provided according rationale, and outlined the overall structure and environment of our implementation. Then, we discussed the internal plug-in structure of our prototype and selected solutions to encountered problems. Last, we presented algorithms used at critical points of weaving and explained the design of underlying data structures.

In the third part we investigated a possible application of our weaver for integrating building specifications information into building models. Following up a discussion on domain-specific aspects, we showed that some of the concerns expressed in building specifications can be classified as such domain aspects. In order to provide a complete

outlook for an integration of these concerns we proposed to create a controlled natural language that can be defined by domain-experts. To illustrate how such a language based on interpretation patterns can close the semantic gap between building specifications and building models we described a possible parsing and interpretation process for a concrete example. Then, we explained how the aspects created by this specification language can be woven into building models using a customisation of our weaver. We presented the individual extensions that complete or change the generic weaving behaviour and illustrated their individual effects using an example.

Altogether, we showed in this thesis that generic model weaving can be practically realised in a declarative, metamodel-independent way if extension facilities are provided. Because our building models customisation required less than 10% of the lines of code of the generic weaver, we are confident that this strategy of customisation is even suited for non-software domains with outstanding specifics. But this assumption has to be confirmed in future work, which we describe in the next section.

12.2. Future Work

Because most of the contributions of this thesis have an initial and essential character they open up avenues for future research. In this section we present selected issues that could serve as starting points for future work and discuss them according to the three parts of our thesis.

12.2.1. Conceptual Extensions and Improvements

Our weaving approach was inspired by previous work on model weaving but we were not able to incorporate all functionality of these approaches into it. This is one of the reasons why various improvements could be investigated in the future:

Semantic Weaving
Because our approach is based on the abstract syntax of modelling languages, it cannot directly be used in situations in which model semantics have to be taken into account. Exemplary for such a situation are sequence diagrams involving loops that would result in different weaving when unrolled (see Section 4.2.1 and [KHJ06]). Future work could investigate this and other examples for semantic weaving and verify whether our approach can be customised or enhanced to support it.

Metamodel Restrictions
It is unclear whether certain ways to define metamodels are incompatible with our generic model weaving approach. Ordered collections, for example, are a problem for which it may be hard to develop a solution that yields correct results for every possible modelling language. Prospective research could analyse this and other metamodel properties that render generic weaving more difficult or even impossible and formulate rules for weavable metamodels.

Advice Instantiation Scope
So far, our approach supports only some of the advice instantiation strategies presented by Morin et al. [MKKJ10]. Integration of the remaining strategies and development of new or improved strategies could be an area for future work.

Weaving Order
If an aspect applies to multiple join points, the order in which weaving is performed at these join points may influence the result in case of overlapping join points or reused instantiations. Future research could investigate how a detection of conflicting join points and support for conflict resolution could be integrated into our approach.

12.2.2. Implementation Improvements

The prototype model weaver provides several possibilities for future work:

Alternative Join-Point Detection
We encountered technical problems with the library used for join-point detection and discovered possible performance bottlenecks that may become critical for scaling. Therefore, we suggest investigating alternative ways to detect join points. Candidates for alternative realisations of the join-point detection phase are, for example, EMFQuery[1] or in-memory databases.

Usability and Dependability
As a research prototype our implementation exhibits a huge potential for usability and dependability improvements. A more convenient and reliable tool could make our approach more attractive for other researchers and for users outside academia.

Code Quality
Because of the redundancy enforced by the mix of declarative and imperative statements for extensions, the use of another programming language could improve the code quality of our implementation. One possibility would be to use Xtend[2], which can be compiled to Java code and therefore sustains extension and debugging compatibility. This language could even be extended with a custom DSL for extension points using the DSL-framework Xtext[3] upon which Xtend is built.

12.2.3. Realisation of a Building Specifications Language

Because we presented a plan for the development of a building specification language the realisation of this plan is the main avenue for future work in this context. A possibility could be to start with a preliminary grammar and a rapid prototype in order to quickly obtain feedback for improvement. First experiments may be conducted with a fixed parser and hard-coded interpretations. But for the planned use by domain-experts it will be crucial to define appropriate languages and mechanisms for user-defined parsing (content word rules) and dynamic semantics (interpretation patterns).

[1] EMFQuery - Search and retrieval for EMF models: eclipse.org/modeling/emf/?project=query2
[2] Xtend - An extension to the Java programming language: eclipse.org/Xtext/xtend
[3] Xtext - A framework for creating extensible DSLs: eclipse.org/Xtext

Bibliography

[AAC+11] M. Alférez, N. Amálio, S. Ciraci, F. Fleurey, J. Kienzle, J. Klein, M. Kramer, S. Mosser, G. Mussbacher, E. Roubtsova, and G. Zhang, "Aspect-oriented model development at different levels of abstraction," in *Proceedings of the 7th European conference on Modelling foundations and applications*, ser. ECMFA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 361–376.

[ABH11] S. Ali, L. Briand, and H. Hemmati, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems," *Software and Systems Modeling*, pp. 1–38, 2011.

[AGGR07] O. Avila-García, A. E. García, and E. V. S. Rebull, "Using software product lines to manage model families in model-driven engineering," in *Proceedings of the 2007 ACM symposium on Applied computing*, ser. SAC '07. New York, NY, USA: ACM, 2007, pp. 1006–1011.

[BCC+10] H. Brunelière, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin, "Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, Eds. Springer Berlin / Heidelberg, 2010, vol. 6138, pp. 32–47.

[BDD+06] J. Bézivin, V. Devedzic, D. Djuric, J.-M. Favreau, D. Gasevic, and F. Jouault, "An m3-neutral infrastructure for bridging model engineering and ontology engineering," in *Interoperability of Enterprise Software and Applications*, D. Konstantas, J.-P. Bourrières, M. Léonard, and N. Boudjlida, Eds. Springer London, 2006, pp. 159–171.

[Béz05] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, pp. 171–188, 2005.

[BKB+08] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing multi-view aspect models," in *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*, ser. ICCBSS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 43–52.

[CB05] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, 2005.

[CDJC09] A. Carton, C. Driver, A. Jackson, and S. Clarke, "Model-driven theme/uml," in *Transactions on Aspect-Oriented Software Development VI*, ser. Lecture Notes in Computer Science, S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5560, pp. 238–266.

[CRS+05] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdoğan, S. Clarke, and A. Jackson, "Survey of aspect-oriented analysis and design approaches," AOSD-Europe, Tech. Rep., 2005.

[CSN05] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 35–43.

[CvdBE06] T. Cottenier, A. van den Berg, and T. Elrad, "The motorola weavr: Model weaving in a large industrial context," in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), Industry Track*. Bonn, Germany: ACM, Mar. 2006.

[DDFV09] M. Didonet Del Fabro and P. Valduriez, "Towards the efficient development of model transformations using model weaving and matching transformations," *Software and Systems Modeling*, vol. 8, pp. 305–324, 2009.

[ETSL11] C. Eastman, P. Teicholz, R. Sacks, and K. Liston, *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*, 2nd ed., ser. s. a: Wiley, 4 2011.

[FBFG08] F. Fleurey, B. Baudry, R. France, and S. Ghosh, "A generic approach for automatic model composition," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin / Heidelberg, 2008, vol. 5002, pp. 7–15.

[FBV06] M. D. D. Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the eclipse amw plugin," in *In Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[FECA04] R. Filman, T. Elrad, S. Clarke, and M. Akşit, *Aspect-oriented software development*, 1st ed. Addison-Wesley Professional, 2004.

[FF00] R. E. Filman and D. P. Friedman, "Aspect-Oriented programming is quantification and obliviousness," in *Workshop on Advanced Separation of Concerns, OOPSLA 2000*. Minneapolis: Addison-Wesley, 2000, pp. 21–35.

[For82] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17 – 37, 1982.

[FS96] N. E. Fuchs and R. Schwitter, "Attempto controlled english (ace)," in *Proceedings of the First International Workshop on Controlled Language Applications (CLAW'96)*, 1996.

[GBN+03] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan, "An approach for supporting aspect-oriented domain modeling," in *Proceedings of the 2nd international conference on Generative programming and component engineering*, ser. GPCE '03. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 151–168.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[GV07] I. Groher and M. Voelter, "Xweave: models and aspects in concert," in *Proceedings of the 10th international workshop on Aspect-oriented modeling*, ser. AOM '07. New York, NY, USA: ACM, 2007, pp. 35–40.

[GV09] I. Groher and M. Völter, "Aspect-oriented model-driven software product line engineering," in *Transactions on Aspect-Oriented Software Development VI*, ser. Lecture Notes in Computer Science, S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5560, pp. 111–152.

[HB08] R. Howard and B.-C. Björk, "Building information modelling - experts' views on standardisation and industry deployment," *Advanced Engineering Informatics*, vol. 22, no. 2, pp. 271 – 280, 2008.

[HE08] K. Hoffman and P. Eugster, "Towards reusable components with aspects: an empirical study on modularity and obliviousness," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 91–100.

[HK02] J. Hannemann and G. Kiczales, "Design pattern implementation in java and aspectj," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 161–173.

[HR04] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64 – 72, oct. 2004.

[JAB+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 719–720.

[JB06] F. Jouault and J. Bézivin, "Km3: A dsl for metamodel specification," in *Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science, R. Gorrieri and H. Wehrheim, Eds. Springer Berlin / Heidelberg, 2006, vol. 4037, pp. 171–185.

[Jéz08] J.-M. Jézéquel, "Model driven design and aspect weaving," *Software and Systems Modeling*, vol. 7, pp. 209–218, 2008.

[KBA02] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," in *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of aspectj," in *ECOOP 2001 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Knudsen, Ed. Springer Berlin / Heidelberg, 2001, vol. 2072, pp. 327–354.

[KHJ06] J. Klein, L. Hélouet, and J.-M. Jézéquel, "Semantic-based weaving of scenarios," in *Proceedings of the 5th International Conference on*

Aspect-Oriented Software Development (AOSD'06). Bonn, Germany: ACM, Mar. 2006.

[KKS+12a] E. Kalnina, A. Kalnins, A. Sostaks, E. Celms, and J. Iraids, "Tree based domain-specific mapping languages," in *SOFSEM 2012: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, Eds. Springer Berlin / Heidelberg, 2012, vol. 7147, pp. 492–504.

[KKS+12b] J. Klein, M. E. Kramer, J. R. H. Steel, B. Morin, J. Kienzle, O. Barais, and J.-M. Jézéquel, "On the formalisation of geko: a generic aspect models weaver," University of Luxembourg, Tech. Rep., 2012.

[KKS12c] M. E. Kramer, J. Klein, and J. R. Steel, "Building specifications as a domain-specific aspect language," in *Proceedings of the seventh workshop on Domain-Specific Aspect Languages*, ser. DSAL '12. New York, NY, USA: ACM, 2012, pp. 29–32.

[KPP06] D. Kolovos, R. Paige, and F. Polack, "Merging models with the epsilon merging language (eml)," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin / Heidelberg, 2006, vol. 4199, pp. 215–229.

[KR10] L. Kápova and R. Reussner, "Application of advanced model-driven techniques in performance engineering," in *Computer Performance Engineering*, ser. Lecture Notes in Computer Science, A. Aldini, M. Bernardo, L. Bononi, and V. Cortellessa, Eds. Springer Berlin / Heidelberg, 2010, vol. 6342, pp. 17–36.

[KT08] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, 1st ed. Wiley-IEEE Computer Society Pr, 3 2008.

[LMT+04] D. Lugato, F. Maraux, Y. L. Traon, V. Normand, H. Dubois, J.-Y. Pierron, J.-P. Gallois, and C. Nebut, "Automated functional test case synthesis from thales industrial requirements," in *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2004, pp. 104–111.

[LMV+07] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J.-M. Jézéquel, "Introducing variability into aspect-oriented modeling approaches," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer Berlin / Heidelberg, 2007, vol. 4735, pp. 498–513.

[LQ06] P. Lahire and L. Quintian, "New Perspective To Improve Reusability in Object-Oriented Languages," *Journal of Object Technology (ETH Zurich)*, vol. 5, no. 1, pp. 117–138, 2006.

[MBJR07] B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos, "Towards a generic aspect-oriented modeling framework," in *Models and Aspects workshop, at ECOOP 2007*, July 2007.

[MBNJ09] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel, "Taming Dynamically Adaptive Systems with Models and Aspects," in *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, 2009.

[MCMTFBM09] C. Martínez-Costa, M. Menárguez-Tortosa, J. T. Fernández-Breis, and J. A. Maldonado, "A model-driven approach for representing clinical archetypes for semantic web environments," *Journal of Biomedical Informatics*, vol. 42, no. 1, pp. 150 – 164, 2009.

[MG06] T. Mens and P. V. Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, no. 0, pp. 125 – 142, 2006.

[MGvFGCB10] A. Molesini, A. Garcia, C. von Flach Garcia Chavez, and T. V. Batista, "Stability assessment of aspect-oriented software architectures: A quantitative study," *Journal of Systems and Software*, vol. 83, no. 5, pp. 711 – 722, 2010.

[MKBJ08] B. Morin, J. Klein, O. Barais, and J.-M. Jézéquel, "A generic weaver for supporting product lines," in *EA '08: Proc. of the 13th international workshop on Early Aspects at ICSE'08*. New York, NY, USA: ACM, 2008, pp. 11–18.

[MKKJ10] B. Morin, J. Klein, J. Kienzle, and J.-M. Jézéquel, "Flexible model element introduction policies for aspect-oriented modeling," in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 63–77.

[RB09] I. Reinhartz-Berger, "Domain aspects: Weaving aspect families to domain-specific applications," in *Domain Engineering Workshop at CAiSE'09*, 2009.

[RBJ07] R. Ramos, O. Barais, and J.-M. Jézéquel, "Matching model-snippets," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer Berlin / Heidelberg, 2007, vol. 4735, pp. 121–135.

[RCG+10] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, and W. Joosen, "Aspect-oriented software development in practice: Tales from aosd-europe," *Computer*, vol. 43, no. 2, pp. 19 –26, feb. 2010.

[RGF+06] Y. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg, "Directives for composing aspect-oriented design class models," in *Transactions on Aspect-Oriented Software Development I*, ser. Lecture Notes in Computer Science, A. Rashid and M. Aksit, Eds. Springer Berlin / Heidelberg, 2006, vol. 3880, pp. 75–105.

[RKR+06] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger, and M. Stumptner, "A generator framework for domainspecific model transformation languages," in *In Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS), Paphos*, 2006.

[RM06] A. Rashid and A. Moreira, "Domain models are not aspect free," in *Proc. of MoDELS 2006*, ser. LNCS. Berlin / Heidelberg: Springer-Verlag, 2006, vol. 4199, pp. 155–169.

[SD11] J. Steel and R. Drogemuller, "Domain-specific model transformation in building quantity take-off," in *Proc. of MoDELS 2011*, ser. LNCS. Berlin / Heidelberg: Springer-Verlag, 2011, pp. 198–212.

[SDD11] J. Steel, K. Duddy, and R. Drogemuller, "A transformation workbench for building information models," in *Theory and Practice of Model Transformations*, ser. LNCS. Berlin / Heidelberg: Springer-Verlag, 2011, vol. 6707, pp. 93–107.

[SFS+08] P. Sánchez, L. Fuentes, D. Stein, S. Hanenberg, and R. Unland, "Aspect-oriented model weaving beyond model composition and model transformation," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds. Springer Berlin / Heidelberg, 2008, vol. 5301, pp. 766–781.

[SHM+10] W. Shen, Q. Hao, H. Mak, J. Neelamkavil, H. Xie, J. Dickinson, R. Thomas, A. Pardasani, and H. Xue, "Systems integration and collaboration in architecture, engineering, construction, and facilities management: A review," *Advanced Engineering Informatics*, vol. 24, no. 2, pp. 196 – 207, 2010.

[SSK+07] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel, "A survey on aspect-oriented modeling approaches," Vienna University of Technology, Tech. Rep., 2007.

[Sta73] H. Stachowiak, *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

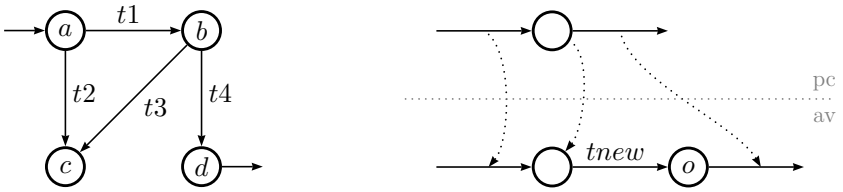
[Ste05] F. Steimann, "Domain models are aspect free," in *Proc. of MoDELS 2005*, ser. LNCS. Berlin / Heidelberg: Springer-Verlag, 2005, vol. 3713, pp. 171–185.

[SVBvS06] T. Stahl, M. Völter, J. Bettin, and B. von Stockfleth, *Model-driven software development: technology, engineering, management*, ser. Wiley Software Patterns Series. John Wiley, 2006.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 107–119.

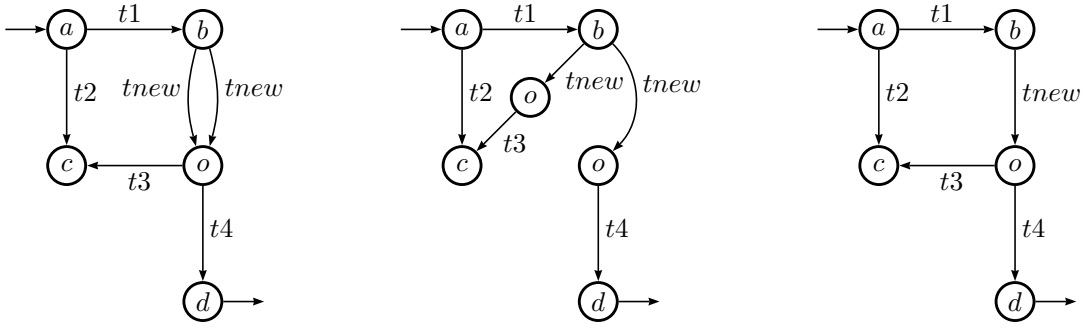
[WJ08] J. Whittle and P. Jayaraman, "Mata: A tool for aspect-oriented modeling based on graph transformation," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin / Heidelberg, 2008, vol. 5002, pp. 16–27.

Appendix



(a) base model (b) pointcut and advice model

Fig. A.1.: A base model used for weaving the LTS aspect of Fig. 5.7 as shown in Fig. A.2.



(a) global scope (b) per-join-point scope (c) custom scope

Fig. A.2.: Woven models obtained for the base and aspect model of Fig. A.1 using (a) global advice instantiation scope for o, (b) per-join-point scope for *tnew* and o, and (c) custom per-matched-state scope for *tnew* and o.

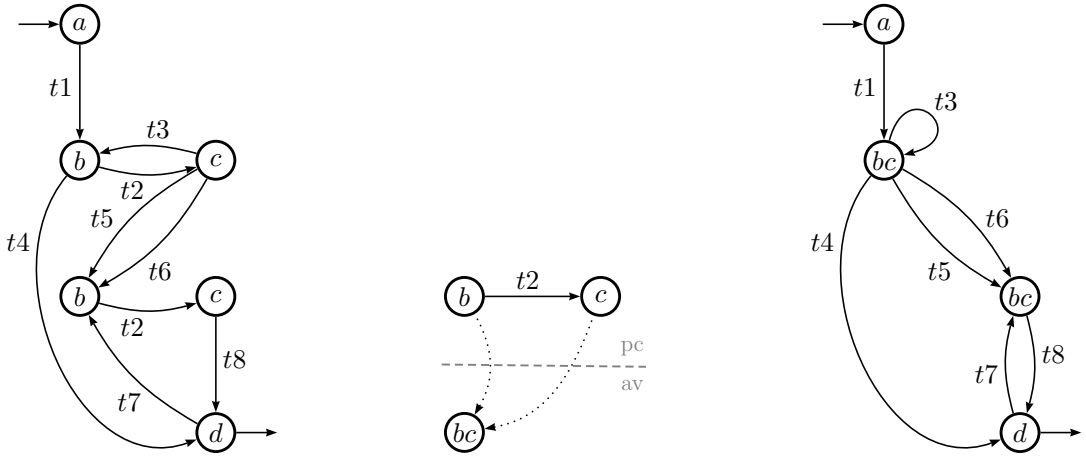


Fig. A.3.: Weaving an aspect into an LTS while merging the base elements b and c.

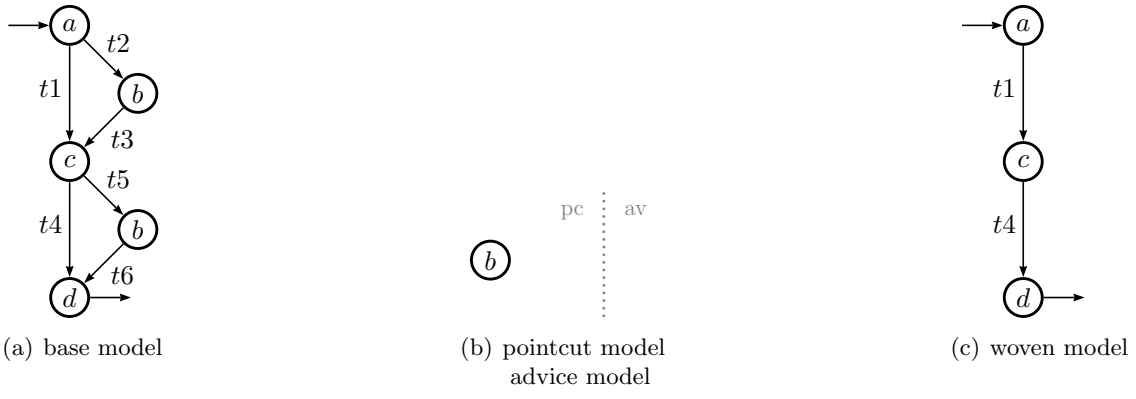


Fig. A.4.: Weaving an aspect for a small LTS while removing the base elements b.

List of Figures