Karlsruhe Institute of Technology

Configurable Model Refinements for Palladio based on a Generic Model Weaver

Diploma Thesis of

Flavie Roussy

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

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	Dipl.-Inform. Max Kramer
Second advisor:	Dipl.-Inform. Erik Burger

Duration:: 21. January 2013 – 19. July 2013

Flavie Roussy: *Configurable Model Refinements for Palladio based on a Generic Model Weaver*, Diploma Thesis, Version 1.0, July 19, 2013

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 institution for any award. Where information was derived from the published or unpublished work of others, this has been acknowledged.

Karlsruhe, July 2013

Flavie Roussy

Acknowledgements

Special thanks to my advisor, Max Kramer, for his useful practical comments as well as remarks on the structure of the thesis and his engagement throughout the entire process of this master thesis. I am also deeply grateful to all people of the chair of Prof. Dr. Ralf H. Reussner who provided me with their work and helped me to reuse them.

I would like to express my gratitude to the serval team of Yves Le Traon for three months I spent at the University of Luxembourg. My sincere thanks goes to Jacques Klein and Phu Hong Nguyen for their feedback. I would also like to express my sincere gratitude to Elöd Egyed-Zsigmond and Christine Solnon that accepted to support this thesis at my home university, INSA.

I want to sincerely thank all people that allowed me to work on this thesis supported in three countries. It was a nice challenge to face and perfectly ends my bi-national university studies.

Abstract

In Component-Based Software Engineering (CBSE), abstract views of systems are realized. These abstract design models ignore low-level details that are not relevant at early stages of the conception. In software performance engineering, design models are used to predict performance characteristics of systems prior to their implementation. The Palladio approach uses CBSE concepts to predict the performance and other qualities of software architectures. To perform accurate performance predictions, design models need to be refined with low-level details. Including these details directly into design models would increase the modeling effort and the complexity of models in an undesired manner. Performance model completions are model refinements which close the gap between abstract models and low-level details needed to perform accurate performance predictions. They need to be configurable so that inserted low-level details fit specific considered cases. In the context of PCM models, two model completions are implemented using two different model transformation languages. In this thesis these two completions are realized using model weaving. The applied completion concepts are compared with those used in model weaving based completions.

One completion adds information on networks and middleware composing a connector. It is implemented in Java. The second completion specifies details on thread pools used in parallel processing. It is realized using Higher Order Transformations (HOTs) implemented in the general-purpose model transformation language QVT-R. Model weaving approaches lack configurability. To counter this problem, configuration-based variability is introduced in a model weaving approach. Similarly to HOTs, aspects are applied on aspects to specialize their behavior. The chosen model weaving approach is then adjusted to fit PCM models. The three introduced realizations of performance model completions using model transformation languages and model weaving are finally compared in terms of length, complexity and execution time using the two model completions as case studies.

Zusammenfassung

In der Component-Based Software Engineering (CBSE) werden abstrakte Sichten von Systemen realisiert. Diese abstrakte Entwurfsmodelle berücksichtigen nicht Details der Implementierung, die in einer frühen Entwicklungsphase nicht relevant sind. In der Softwareentwicklung können Entwurfsmodelle dazu verwendet werden, die Leistungsmerkmalle des Systems vor seines Implementierung vorherzusagen. Der Palladio Ansatz verwendet CBSE Konzepten, um Leistungsvorhersage und andere Eigenschaften einer Softwarearchitektur vorherzusagen. Um präsize Leistungsvorhersagen durchzuführen müssen Entwurfsmodelle mit Details der Implementierung verfeinert werden. Das Hinzufügen dieser Details auf der Entwurfsmodelle selbst würde den Modellierungsaufwand und die Komplexität in einer unerwünschten Weise erhöhen. Leistungsmodellvervollständigungen sind Modelle-Verfeinerungen, die den Abstand zwischen abstrakte Modelle und Details der Implementierung schließen. Leistungsmodellvervollständigungen müssen konfigurierbar sein, sodass eingeführte Details der Implementierung sich an bestimmte Fälle anpassen. Im Rahmen von PCM-Modelle sind zwei Modellvervollständigungen mittels zwei verschiedenen Modell-Transformation Sprachen implementiert. In dieser Diplomarbeit realisieren wir diese zwei Vervollständigungen mittels Modellweberei. Die angewendete Konzepte wurden mit denen verglichen, die in Modellweberei basierte Modellvervollständigungen angewendet wurden.

Die erste Modellvervollständigung fügt Informationen über das Netz und die Middleware hinzu, die einen Konnektor bilden. Sie wurde in Java implementiert. Die zweite betrachte Modellvervollständigung gibt Details über eine Thread Pool Komponent an. Sie anwendet Higher Order Transformations (HOTs) implementiert in einer Universalsmodell-Transformation Sprache, QVT-R. Modellweberei Ansätze fehlen Variabilität. Um es zu entgegnen haben wir konfigurierbare Variabilität in einen Ansatz zur Modellweberei integriert. Ähnlich wie bei HOTs werden Aspekte auf Aspekte angewendet, um ihr Verhalten zu spezialisieren. Die ausgewählte Modellweberei Ansatz wurde angepasst, um PCM Modelle zu verarbeiten. Die drei eingeführte Realisierungen von Leistungsmodellvervollständigungen mittels Modell-Transformation Sprachen und Modellweberei wurden schließlich im Bezug auf Größe, Komplexität und Ausführungslaufzeit anhand der beiden Modellverfeinerungen verglichen.

Résumé

En utilisant la programmation orientée composant, les systèmes modélisés sont représentés par des vues abstraites. Ces design models abstraits font abstraction des détails de bas niveau qui ne sont pas pertinents aux premiers stades de la conception. En software performance engineering, les design models sont utilisés pour prédire les performances d'un système avant son implémentation. La technique de prédiction de performances nomée Palladio est basée sur les concepts de la programmation orientée composant. Pour effectuer des prédictions réalistes, les design models doivent être complétés avec des détails de bas niveau. Modéliser ces détails directement sur les design models augmenterait de manière indésirable l'effort de modélisation ainsi que la complexité des modèles. Les performance model completions sont des model refinements qui complètent un modèle en y incluant les détails de bas niveau nécessaires pour prédire des performances réalistes. Pour que les détails ajoutés aux modèles puissent s'adapter à la diversité des situations considérés, les performance model completions doivent être configurables. Afin de prédire des performances réalistes à partir de modèles PCM, deux model completions sont implémentées à, chacune utilisant un langage de transformation de modèles. Dans le cadre de ce projet de fin d'études, ces deux completions sont réalisées à l'aide du model weaving. Les concepts appliqués sont comparés avec ceux utilisés dans les précédantes implémentations.

La première complétion ajoute des informations sur le réseau et les middlewares utilisés par un connecteur. Elle est implémentée en Java. La seconde détaille une thread pool utilisée pour du traitement en parallèle. Elle est réalisée à l'aide de Higher Order Transformations (HOTs) implémentés en QVT-R, un langage de transformation de modèles. Les techniques de model weaving manquent de configurabilité. Pour surmonter cela, de la variabilité est introduite dans une technique de model weaving afin de la rendre configurable. Dans la même logique que les HOTs, des aspects sont appliqués sur d'autres aspects pour spécialiser le comportement de ces derniers. La technique de model weaving choisie est ensuite adaptée pour correspondre aux modèles PCM. Les trois réalisations de performance model completions utilisant des langages de transformation de modèles et le model weaving sont finalement comparées en termes de longueur, complexité et temps d'exécution en utilisant les deux modèles complétions comme études de cas.

Contents

Nomenclature

AMW	Atlas Model Weaver
AOM	Aspect Oriented Modeling
ATL	ATLAS Transformation Language
BSD	Basic Sequence Diagram
CBSE	Component-Based Software Engineering
CMOF	Complete MOF
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
HOT	Higher Order Transformation
IFC	Industry Foundation Classes
LTS	Labeled Transition System
M2M	Model-to-Model
M2T	Model-to-Text
MDSD	Model Driven Software Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PCM	Palladio Component Model
QoS	quality-of-service
QVT	Query View Transformation
QVT-O	QVT Operational
QVT-R	QVT Relational
RAM	Reusable Aspect Models
SEFF	Service Effect Specifications
SPL	Software Product Line

1. Introduction & Motivation

The complexity of developed software grows constantly. To manage this complexity, abstract views of systems are realized using Model Driven Software Development (MDSD). MDSD increases the level of abstraction in program specification and the automation in program development. A global view of systems is realized, ignoring details that are not relevant at early stages of the conception. In software performance engineering design models are used to predict performance characteristics of systems prior to their implementation. However, to perform accurate performance predictions, details need to be introduced into design models. All low-level details are not added to design models. Design models have to be configurable to take into account low-level details that are relevant to perform accurate performance predictions. Drawing these details directly on models would increase the modeling effort and the complexity of the models in an undesired manner. Model refinement techniques are used to include low-level details in design models without explicitly draw these details on the models.

Components form the central building blocks in Component-Based Software Engineering (CBSE). The Palladio Component Model (PCM) approach is a performance prediction approach based on CBSE concepts. To perform accurate performance predictions, design models are refined to form performance models. Two realizations of model completions are used to refine models. The first considered model completion is the thread pool completion realized by Higher Order Transformation (HOT). This approach takes as input a base model with annotated model elements, apply to this model a completion and its configuration and produces as output a refined model. This method, applied to software predictions, is described by Kapova and Reussner [KR10]. The second model completion considered, the connector completion, is realized in Java and described by Becker [Bec08]. These two realizations use transformation languages to implement model completions and thus a textual representation.

As explained previously, while refining models, low-level details are introduced into abstract design models. These low-level details are applied to a specific type of model elements. As models contain several elements of the same type, same model refinements are applied at different locations in the model. From this viewpoint, model refinements can be considered as cross-cutting concerns, as they are aspects involved in different parts of the classical model of our application. Model weaving approaches deal with these cross-cutting concerns, that is why we would like to use a model weaving approach to perform model refinements. Model refinements realized using model weaving also differ from the

two first realizations introduced as they use a graphical representation. As Kramer described it [Kra12], a model weaver takes as input a base, pointcut and advices models and produces as output a woven model. Various model weaving approaches exist and a first decision was to choose which model weaver to use, which one best fits our needs.

Model weaving and model transformations are based on different observations, use very different techniques and have been developed separately. The main concern of this thesis is to compare a model weaver and model transformations languages in the context of model completions. This comparison is based on two case studies: the thread pool completion and the connector completion. Both completions are implemented and refine PCM models. For each case studies, equivalent implementations are realized using model weaving. The thread pool completion needs to be configurable. Whereas model completions realized by HOTs are configurable, model weaving approaches lack configuration-based variability. A first step to realize this completion is to introduce configuration-based variability into the chosen model weaver.

The structure of the document reflects different aspects of the thesis. Chapter 2 and Chapter 3 introduce domains covered by the thesis. Chapter 2 provides the fundamental background for our work. Principles and key notions of MDSD, model weaving and CBSE are defined in this chapter. The chosen model weaver is described in this chapter. Chapter 3 discusses related work on model completions, HOTs, model weaving and variability management in model weaving.

Chapter 4 describes the approach chosen to introduce configuration-based variability into the chosen model weaver. Comparably to HOTs, aspects are applied on other aspects to specialize them. This approach includes a Software Product Line (SPL) to handle the variability.

Chapter 5 introduces the central item of the thesis, the comparison between model weaving and model transformation to realize performance model completions. On a conceptual level the representation and verbosity of the two approaches are considered. Two case studies, the thread pool and connector completion, illustrate how these conceptual differences influence the realization of a precise model completion. Finally, limitations of each approach due to tool support are studied.

The chosen model weaver has been expanded to weave PCM models. Additional functionnalities are added to the weaver, meta-model code generation phase are adjusted and some weaving operations are improved. Chapter 6 closes the thesis with practice oriented observations about the weaving of PCM models with the chosen model weaver.

2. Foundations

In this chapter we introduce the concepts that are fundamental to the work presented in this thesis. In Section 2.1, the basic concepts of MDSD and model transformations are explained. Section 2.2 describes the principles of model weaving and shows why it can be used to perform model completions. Finally, we introduce in Section 2.3 CBSE, our definition of model completions, and then present the PCM, a modeling language for CBSE.

2.1. Model-Driven Software Development

As Stahl and Voelter describe it [SV06], the goal of MDSD is to improve the quality, the reusability and the development performance of software. These goals are achieved by raising the level of abstraction in program specification and increasing the automation in program development. During the program specification, models at different levels of abstraction are used to specify global concepts as well as low level details. Executable model transformations are used during the program development to navigate from higher-level models to lower-level models, or from lower-level models to higher-level models. Based on the navigation direction, the transformations add details to the models, or omit details useless at a higher level of abstraction. In this software development method, models represent the software during its whole life-cycle and thus are considered as first class entities. A model is an abstraction of the reality and is characterized by three main properties:

- Abstraction: A model does not contain all characteristics of the represented system. Details that do not serve our purpose are removed from the model.
- Homomorphism: Statements on model elements hold for real world entities.
- Pragmatics: Each model represents a vision of the system that serves our purpose. Models have to be designed in a goal-driven way.

Each model is described on a higher level of abstraction by its meta-model. A definition of a meta-model can be formulated this way:

Definition 1 (Meta-Model) *A Meta-Model is a precise definition of the model elements and rules needed to create valid models.*

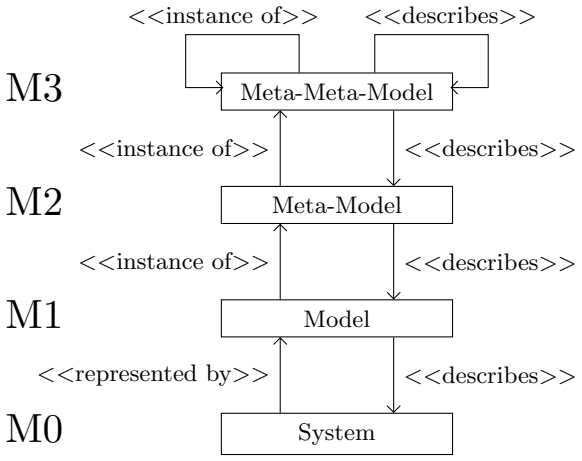


Fig. 2.1.: Layers of modeling, adapted from [Kra12]

A meta-model itself is described on a higher level by a meta-meta-model. The Object Management Group (OMG) standard defines four levels of abstractions numbered from M0 to M3. Fig. 2.1 shows these four levels. In MDSD, the objects we handle is software, thus the M0-level, in our case, represents these softwares. The M1-level depicts the models describing the softwares, the M2-level the meta-models describing our models and the M3-level the meta-modeling languages describing our meta-models. In the OMG standard the meta-meta-model is self-describing, that means that the meta-meta-model is an instance of itself.

On the meta-meta-model level the OMG defines a standard called the Meta Object Facility (MOF). This standard evolved to Essential MOF (EMOF) and Complete MOF (CMOF). The Eclipse Modeling Framework (EMF) [SBMP08] is a framework that facilitates the modeling and code generation for building tools and applications based on a structured data model. This is an implementation based on the EMOF standard. The meta-model of EMF is Ecore, which closely matches EMOF 2.0. Fig. 2.2 shows the principal components of Ecore. It shows that all meta-models that are instances of Ecore are organized in packages, and that each of these packages contains meta-classes and data types. The meta-classes contain attributes, references and operations, which contain parameters. The figure also shows a data type, the enumerations, and that the attributes, references, operations and parameters possess a type.

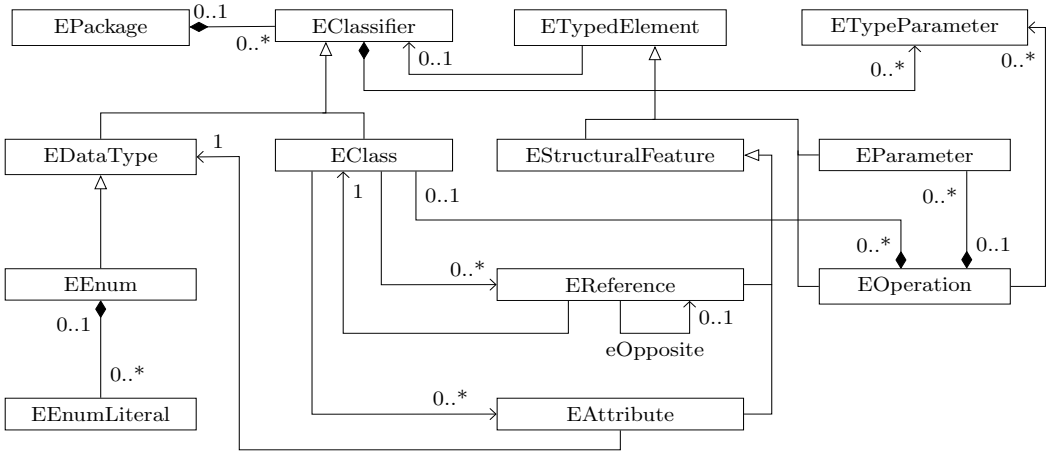


Fig. 2.2.: A simplified representation of central concepts of EMF's meta-modeling language Ecore from [FBFG08]

Section 2.1.1 introduces model transformations and their utility in MDSD. Some language used to implement are defined in Section 2.1.2. Finally, HOTs are described in Section 2.1.3.

2.1.1. Model Transformation

As models are designed in a goal-driven way, various different models can be designed from a single system. The many models mainly vary in terms of representations, abstraction levels or views of the system. However all these models are related to each other and we want to navigate between them. From a certain model of our system, we want to be able to generate a model for a higher abstraction level for example. We use model transformations to achieve this goal. As shown in Fig. 2.3, a model transformation *Tab* modifies a model *Ma* , instance of a meta-model *MMa* , in a model *Mb* , instance of a meta-model *MMb* . The transformation itself is an instance of a meta-model *MMt* and refers to the meta-models *MMa* and *MMb* . The meta-models *MMt* , *MMa* and *MMb* are instances of a common meta-meta-model, *MMM* .

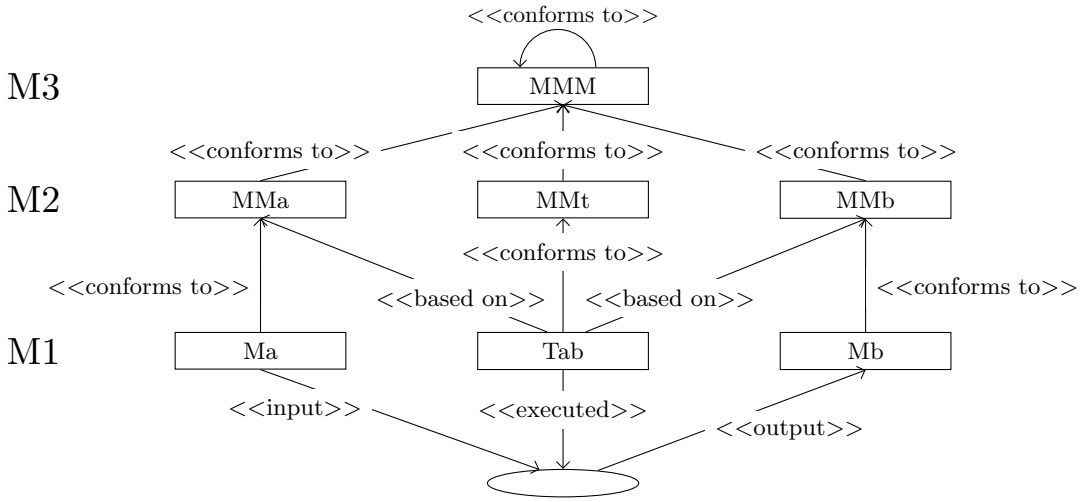


Fig. 2.3.: Principle of a model transformation, from [JAB+06]

Stahl et al. [SVC06] give the following definition for a model transformation:

Definition 2 (Model Transformation) *A Model Transformation is an automated modification of models that ensures conformance to the structural constraints for these models.*

The model transformations can also be characterized by their properties. Mens and Van Gorp [MG06] classify model transformations according to these properties. They suggest a taxonomy for model transformations. Based on this classification, we introduce some types of transformations, symbolized by Fig. 2.4. Following properties have been considered to establish the classification:

- Type of the output: We distinguish Model-to-Model (M2M) and Model-to-Text (M2T) transformations.
- Run direction: Bidirectional transformations can be run in both directions, and thereby the input and output models can be interchanged.
- Modelisation language: Models can be expressed in different modelisation languages and model transformations can be used to move from a representation to another.

Endogenous transformations are transformations where the input and output models are expressed in the same modelisation language. Exogenous transformations are transformations where the input and output models are expressed in different modelisation languages.

- One model used as input and output: The endogenous transformations that use the same model as input and output model are called in-place transformations. On the contrary, transformations that take different models as input and output are called out-place transformations.
- Abstract or refine: Horizontal transformations are transformations where the source and target levels are at the same level of abstraction. In vertical transformations the source and target model have different levels of abstraction.
- Input meta-model: Y-transformations take instances of different meta-models as input.
- Configuration model: As Kapova [Kap11] described it, in some Y-transformations, one input model configures the transformation itself. These configuration models are called mark models and the transformations mark transformations.

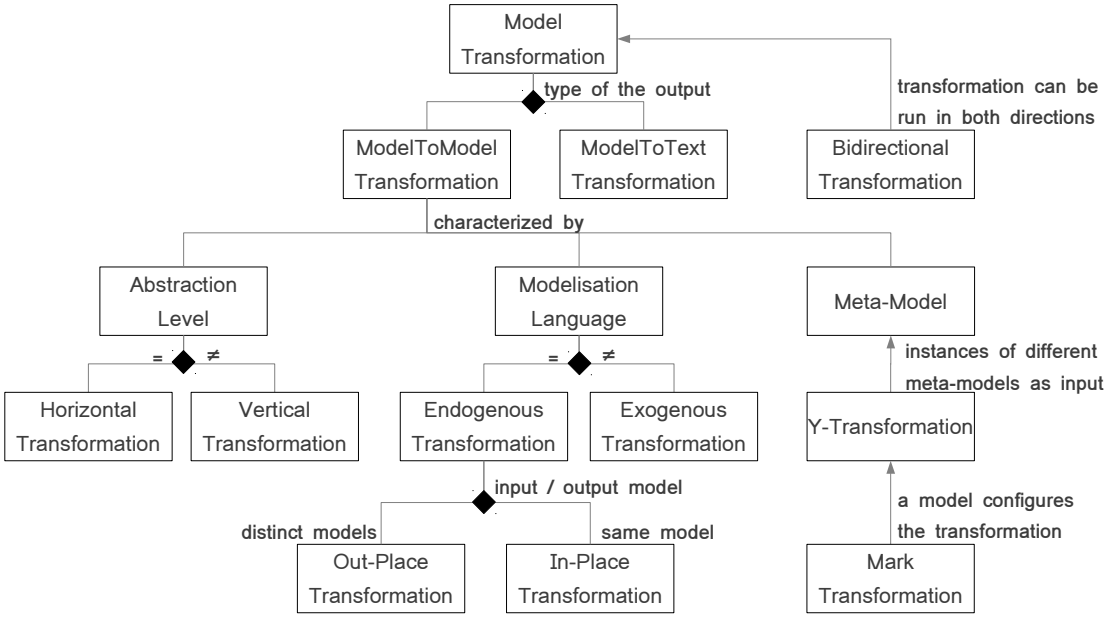


Fig. 2.4.: Classification of some model transformations

Biehl [Bie10] highlights the variety of model transformations and the diversity of tasks they are used for. "Modifying, creating, adapting, merging, weaving or filtering models" are tasks for which model transformations are typically used. In his study, he also classifies the changes applied to a model by transformations in six categories: change of abstraction, change of meta-models, supported technical spaces, supported number of models, supported target type and preservation of properties.

Knowing all the characteristics of the transformation we want to realize is necessary to choose the right tool to do it. Section 2.1.2 introduces some transformation languages and their characteristics. Being aware of the variety of model transformations will help us to understand the differences between the distinct transformation languages.

2.1.2. Model Transformation Language

In this part we focus on M2M transformations. The concepts used in M2T transformations differ from the one used in M2M transformations and thus the languages used to do it also differ. A model transformation takes as input the meta-models of the input and output models. This way, we are able to specify all kinds of model transformations and abstract of the fact that a transformation is endogenous or exogenous for example. The transformation takes a meta-model for each input (and output) model, which allows us to perform Y-transformations. Knowing the meta-models of the input and output models allows to navigate in these models according to their structure. It enables us to know the relations between the model elements and to access them by the name they have in the model.

However, it is possible to implement model transformations without the meta-models of the input and output models. Java is for example used to perform model transformations. On one hand, we do not benefit from language concepts that allow us to directly use the features defined in the meta-model and thus can not statically check them. On the other hand, as this language is well known, there is no need for the developer to learn a new specific language. Besides Java works with instructions, in an imperative programming style. This is what programmers are used to but this is not necessarily the most efficient way to specify model transformations, compared to declarative languages.

Two model transformations specific languages are introduced. QVT Relational (QVT-R) is part of a standard defined by the OMG, Query View Transformation (QVT). It works with rules and uses a declarative paradigm. The second language we present, ATLAS Transformation Language (ATL), combines operational and relational approaches.

2.1.2.1. QVT

The OMG defines a standard set of transformation languages [Nol09] consisting of three languages: QVT Operational (QVT-O) on the operational side, QVT-R and QVT Core Languages on the relational side. All the models handled by these transformation languages have to conform to the OMG standard for meta-models, MOF, that we presented in Section 2.1. In this thesis we use QVT-R, therefore we give more informations about it in the following sections. The detailed specifications for all three languages can be found on the OMG website [Obj13].

Unlike generic languages such as Java, QVT-R is a declarative language. Many programmers are not used to this way of thinking and therefore they need to adapt their logic to this declarative paradigm the first time they use it. A QVT-R transformation is specified in form of a set of rules that have to be satisfied. At design time, the input and output models are not distinguished. The distinction is only made at the execution time. As Kurtev [Kur08] describes it, a QVT-R transformation can be executed according to four execution scenarios.

- In *check-only* mode the transformation checks if the given models are consistent according to the transformation. A boolean returns the result of the comparison.
- An unidirectional transformation will be executed in a given direction. The output models will be created so that, after the transformation execution, they fit the rules defined in the transformation.
- Executing model synchronization, a set of models is compared to the relations defined in the transformation definition. If a relation in not satisfied, the models are updated in order to satisfy the relations.
- In-place updates are used to perform changes on a unique model.

A QVT-R transformation consists of a set of rules, described with relations. Each relation contains a set of object patterns, that are *check-only* or *enforce* object patterns. Elements contained in *check-only* object patterns will be matched against existing model elements, whereas *enforce* object patterns will be created or updated to match the relation if necessary. The structure of the relations allows QVT-R to support complex object pattern matching and object template creation. QVT-R creates traces to record the model elements involved in the transformation. These traces are created implicitly, so that the whole process is transparent for the developer.

2.1.2.2. ATLAS Transformation Language

The ATL is a model transformation language that combines relational and operational approaches. It supports only unidirectional transformations and produces a set of target models from a set of source models. The source models are read-only models: they can be navigated but not altered. The target models are write-only models: they can be altered but not navigated. Fig. 2.5 shows that the meta-model of the ATL transformation is MOF. Therefore, the meta-models of the handled models have to conform to MOF, as shown in Fig. 2.3.

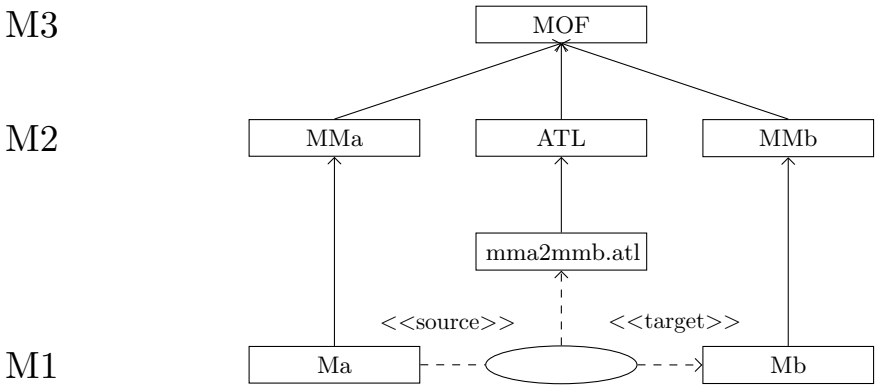


Fig. 2.5.: Overview of ATL transformational approach, from [JK06]

Jouault and Kurtev [JK06] described the structure of an ATL transformation. It is defined in a module, that contains a header section, an import section and helper and transformation rules. The header section specifies the name of the transformation and the source and target models used, as well as their respective meta-models. The term helper comes from the Object Constraint Language (OCL) [WK03], a formal language used to describe expressions on models, defined by the OMG. Helpers are composed of operation and attributes helpers, have a name, a context and a type, and can be defined recursively. Besides, operation helpers can have input parameters. Operation helpers contexts refer to a model element or a module, while attribute helpers are used to associate read-only named values to source model elements. The logic of the ATL transformation is specified using transformation rules. They can be specified using a declarative style and match rules, or using an imperative style and called rules and action block.

As input models are read-only models and output models write-only models, ATL can not perform in-place transformations. If the purpose of a transformation is to modify only a few elements of the input model, the use of ATL, that only performs out-of-place transformations, implies the creation of copy rules for all model elements, which is a time-consuming and error-prone task. To avoid it, ATL provides a *refining* mode. Using this mode, all source model elements are copied into the target models. Only the elements concerned by the matched rules are modified.

2.1.3. Higher Order Transformations

As defined in Section 2.1, models are considered as first class entities in MDSD. Like Kapova [Kap11] described it, in early uses of MDSD technologies, models were used to communicate. These models were manipulated using fixed model transformations. They served as program documentation and code visualization and did not need a high level of abstraction. Then forward engineering was automatized. Code generation tools became powerful and allowed generated code to be similar to hand-written code. Nowadays the transformations themselves become subject of manipulation. Like Bézivin [Béz05] said, "in MDSD everything is a model". Thus we manipulate transformations in the same way we would do it with models.

Biehl [Bie10] suggests a definition for HOTs:

Definition 3 (Higher Order Transformation) *A Higher Order Transformation is a model transformation description with a transformation model as source or target model.*

The HOTs we handle in this diploma thesis realize performance model completions, as we define them in Section 2.3.2. They take as input a base model of the application and a mark model used to configure the transformation. According to the classification we introduced in Section 2.1.1, the HOTs handled in this work are mark transformations. They use a provided library of model transformation fragments to produce a refined model of the application. The input mark model contains the completion to apply, the model elements to refine and the configuration options for each element. These configuration options are specified using feature models. At first we select all feature models that correspond to annotated model elements using this mark model. As shown in Fig. 2.6, a feature model contains configuration options and the corresponding transformation fragments. From these feature models, the global refinement transformation is then built. Finally, the refinement transformation is applied on the base model and produces the refined model. The output model contains the detailed specifications for all marked elements.

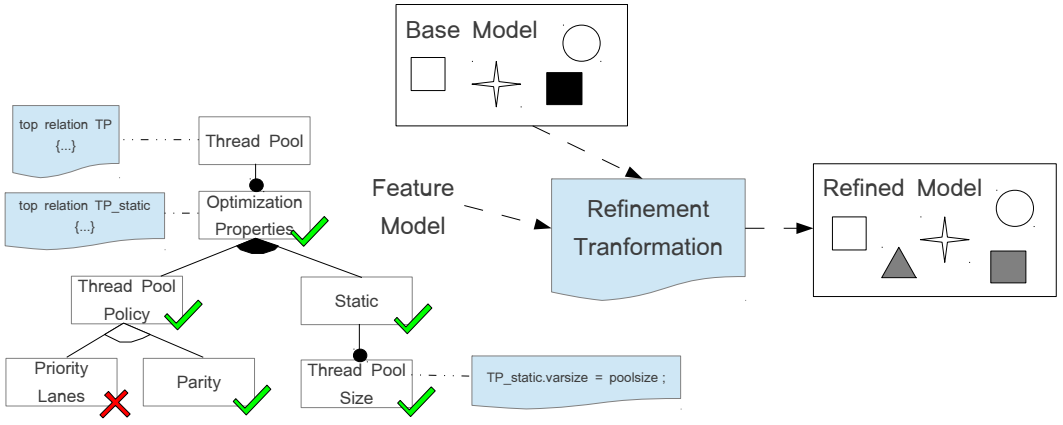


Fig. 2.6.: An example of a model completion realized with HOTs, from [Kap11]

2.2. Model Weaving

In computer science, a coherent set of information having an effect on the software is referred to as a concern. When modeling a software, we try to identify these concerns in order to deal with each individually. Some concerns, like authentication, data validation or memory management, appear at different locations in the software and are called cross-cutting concerns. In other words cross-cutting concerns are aspects that are involved in

different parts of the classical model of our application. Using model weaving, a software architect is able to identify and specify these cross-cutting concerns. These concerns can then be managed at one place, thus eliminating the need to manage the concerns in each part of model they are involved in. Four types of models are defined in many model weaving approaches:

- The base model is the model describing our application.
- The pointcut model defines all model artifacts where we would like to apply the aspect. In a search-and-replace scenario this would be the *search* term. The parts of the base model that match the pointcut model are called join points.
- The advice model describes the changes we would like to apply to each join point. In a search-and-replace scenario this would be the *replace* term.
- The woven model is the produced model.

Kramer [Kra12] describes a transformation process which is composed of two distinct phases. The first phase takes as input the base and pointcut models and identifies the join points. The second phase takes the join points and the advice model as input. This phase applies the changes described in the advice model to the join points and produces the woven model. This second phase is called model weaving.

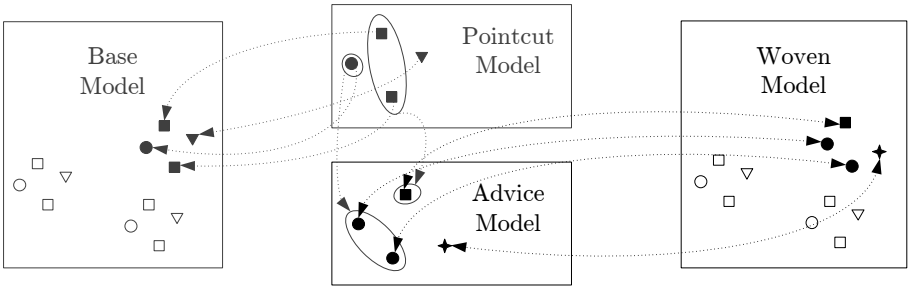


Fig. 2.7.: Symbolic representation of model weaving actions from [Kra12] p.39

Fig. 2.7 illustrates the principle of model weaving using a symbolic representation. It shows the three input models and the output model, and highlights four types of operations that can be executed using model weaving. We remark the following changes between the base and the woven models: the triangle has been removed, a star has been added, the two triangles have been merged and the circle has been duplicated. All these modifications are performed due to the designed advice and pointcut models. This figure illustrates also the mappings performed between the various models. The mapping between the base and the pointcut models depicts the join points, the parts of the base model that match the pointcut model. Elements of the pointcut model are linked with their representative elements in the advice model. The mapping between the advice and woven models links corresponding elements.

Model weaving uses the same concepts as the one used by Aspect Oriented Modeling (AOM) but the principles of these two approaches differ. In AOM aspects are persistent, they live as long as the classes do. The aspect influence on the classes is based on a parametrization and can appear at any time during the life-cycle of the classes. In model weaving, aspects are applied once and for all to the models. However, if model weavers are used in an AOM context, aspects are kept after the weaving operation and are thus persistent aspects. Keeping the aspects after the weaving is the fact of AOM and not model weaving. Considering model weaving aside from AOM, once an aspect has been applied to a model, we consider the woven model and set aside the aspect.

2.2.1. GeKo

Model weavers are the tools used to realize model weaving. Each model weaver conforms to specific requirements. Some only support UML models, while others support instances of any given meta-model ; some only allow to create links between model elements while others are able to modify the models, and thus perform model transformation. The following section introduces GeKo, the model weaver we used in this thesis, and describes its main characteristics. Section 3.3 lists other existing implementations and highlights the features that distinguish them from GeKo.

The model weaver we use, named GeKo, is described by Kramer et al. [KKS+12b]. The two main characteristics of this model weaver are its genericity and extensibility.

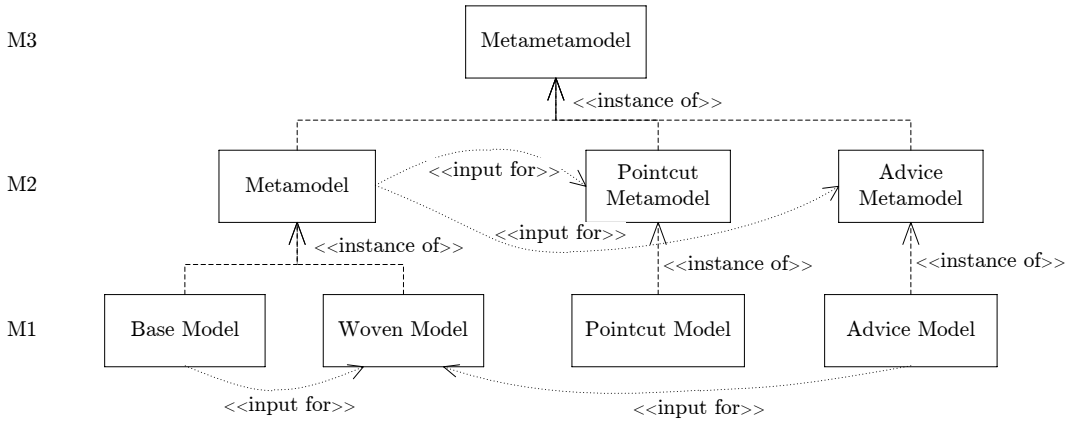


Fig. 2.8.: Models, Meta-Models and Meta-Meta-Models involved in the generic weaving process, from [Kra12]

GeKo is generic because it operates only on the meta-model level and thus enables the transformation of models that are instances of various meta-models. To support instances of any given meta-model, GeKo derives pointcut and advice editors from the original meta-model. First, it derives from the original meta-model the pointcut and advice meta-models, that are relaxed meta-models. As Kramer et al. [KKS+12b] describes it, these variants of the original meta-models are obtained by removing invariants and preconditions, specifying all features of all meta-classes as optional and rendering all abstract classes concrete. Fig. 2.8 shows the models involved in a generic weaving process, their meta-models and meta-meta-models, aligned to the level of model abstraction defined by the OMG. These relaxed meta-models are super-types of the original meta-model, as they modify constraints but do not add or remove model elements. Thus, the original languages and tools can be used to define the pointcut and advice models. The derivation of the pointcut and advice meta-models needs to be executed only once per meta-model. It is a preparatory step and is not part of the weaving process. Using the produced relaxed meta-models, the user specifies the pointcut and advice models that compose the aspect for the given model.

The weaving process is then divided in five phases, represented in Fig. 2.9:

0. Loading: the input models are loaded and made available for all other weaving phases.

1. Join Point Detection: identification of all regions that match the model snippet defined in the pointcut model. In GeKo, this step is performed using the business logic integration platform Drools [JBo13].

2. Inferring a Pointcut to Advice Mapping: inferring a mapping from pointcut model elements to advice model elements for all unambiguous cases. Unambiguous cases

are defined by Kramer et al. [KKS+12b] as cases where "every pointcut element matches at most one advice element of the same type that has the same primitive attributes".

3. Model Composition: composing the base and advice models. Based on the inferred mapping or on the mapping defined by the user, we complete or replace base elements, matched with a pointcut model element, with the corresponding advice element. GeKo does not remove any elements, the removal operations are performed in the last weaving phase.
4. Removal & Clean-up: in this last weaving phase, all base elements that correspond to a pointcut element but that do not correspond to any advice element are removed from the woven model. This removal phase is followed by a clean-up phase that removes all references to removed elements. To ensure that woven models conform to their meta-models, the weaver also removes elements that violate lower bounds constraints. Indeed it is possible that the referenced features have been removed during the first step of the clean-up phase.

At the end of the weaving process, the woven models contain all modifications described in the aspect, are consistent and conform to their meta-models.

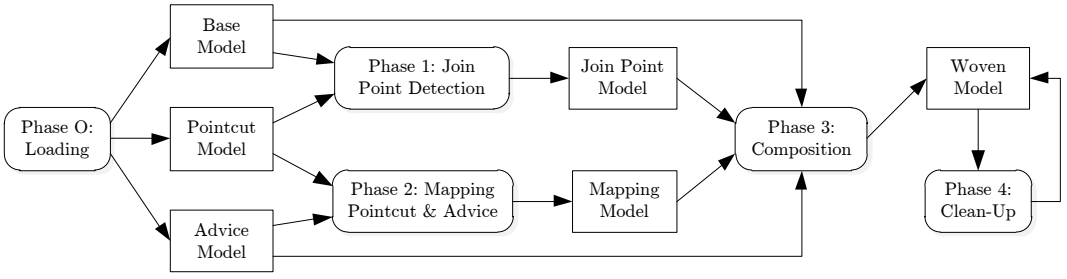


Fig. 2.9.: Weaving Phases in GeKo, from [Kra12]

In order to fit the largest possible number of cases, seven extension points are provided in GeKo. These extension points allow the user to customize the phases of model weaving to specific needs. At least one extension point is provided for each one of the five phases introduced before.

In this section we highlighted the two principal features that influenced our choice. The first one is the fact that GeKo is a generic model weaver and enables the transformation of models that are instances of various meta-models. As a result of this genericity, the usual modeling languages and tools are used to design the pointcut and advice models. The second feature is the complete modification of models offered by GeKo: it supports addition, removal and modification of model elements.

2.3. Component-Based Software Engineering

CBSE is a branch of software engineering that builds systems by defining, implementing and composing independent components. The same way models were considered as first class entities in MDSD, components constitute the key notion in CBSE. The main characteristics of a component have been defined by Reussner et al. [RBH+07]: components form "black-box entities with contractually specified interfaces". They are compared to building blocks for software, as a user does not need to understand their internal structure to reuse them, and are assembled with other components via their interfaces.

Using CBSE, the way components are defined implies to split the task of designing software between a component developer and a software architect. The component developer is in

charge of the internal structure of the components, and develops individual components. The software architect assembles the produced components to build the application but only considers the interfaces provided by the components, not their internal behavior. The benefits of CBSE, achieved, among others, by splitting the design tasks are:

- Increased Reusability: components are designed as independent software blocks. They provide, via their interfaces, a coherent set of functionalities, and therefore can be reused unchanged in a different context.
- Preparation for Evolution: if the behavior of a component has to be updated, its internal structure is changed but not its interfaces. All other components, that use the interfaces of the modified component, will not be impacted by this update. This way, the number of the elements impacted by the change is minimized.
- Short Time-to-Market: as CBSE offers an increased reusability, it enables to promptly build a software that matches the customer needs, reusing existing components.
- Predictability for the QoS Properties of the Architecture: the quality-of-service (QoS) properties of a software are its performance and reliability. In CBSE, each designed component should be provided with detailed QoS specifications. Then the properties of the final architecture can be predicted using the specifications of the separate components assembled to build this architecture.

In this thesis the whole CBSE process is not considered. The focus is on completion techniques that produce models that can be used to perform predictions. Thus we are interested in the prediction of performance properties using CBSE. First, we introduce the PCM, a software component model specifically tuned to enable model-driven QoS predictions. Model completions, a specific type of model transformation used to predict performance properties, are defined in a second step.

2.3.1. Palladio Component Model

Palladio [BKR07] is a software architecture simulation approach based on CBSE. According to Reussner et al. [RBH+07], the goal of PCM is "to assess the expected response times, throughput, and resource utilization of component-based software architectures during early development stages". Designing all information needed to perform performance predictions, four types of developer roles are defined. Fig. 2.10 shows these four types of developer roles and the artifacts they produce in the Palladio Process Model.

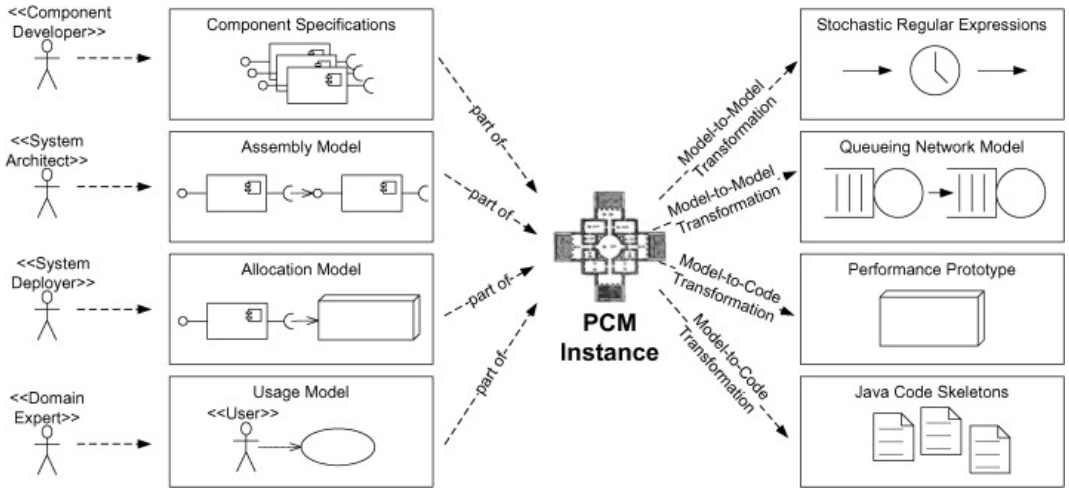


Fig. 2.10.: Developer roles in the Palladio Process Model, from [BKR09]

- Component developers are in charge of the components specification and implementation. They produce repository models that describe components and interfaces. Each component is linked to the interfaces it requires and provides. Performance properties of the components are described via Service Effect Specifications (SEFF), that abstractly illustrate the relations between the required and provided services.
- Software architects compose the components to build applications in an architecture model. As multiple instances of the same component can be used to build an architecture, components are embedded in assembly contexts. These assembly contexts, used in the next step of the Palladio Process Model, are described in assembly models.
- System deployers model the resource environment in a resource environment model. Then they allocate each assembly context to a resource of the resource environment in an allocation model.
- Domain experts are business experts that are familiar with the end-users of the application. They design usage models that describe the system's usage in terms of workload, user behavior and parameters.

The models created by the four types of developer roles are parts of a PCM instance. The information contained in these models is then transformed into analysis models to predict the QoS properties of the build architecture. Many different QoS predictions can be realized depending on the needs. A M2M transformation can be executed to produce stochastic regular expressions or a performance prototype can be generated using a model-to-code transformation. The produced stochastic regular expressions will be analyzed by a QoS analyst, while the performance prototype will be executed and its performance will be measured.

2.3.2. Performance Model Completion

In PCM, performance predictions are performed during early development stages, based on the several design models we described in Section 2.3.1. A PCM model has two purposes: represent the coarse-grained architecture and account for performance-relevant facts. However, these two purposes cannot be satisfied using a single model. Indeed, to perform correct performance predictions, the models have to contain performance-relevant facts that are low-level details. Include these low-level details in the original models breaks their abstraction level and makes it difficult to concentrate on important elements. If one of the component is a thread pool, for example, if we consider the design of the architecture, the detailed characteristics of the thread pool (static or dynamic size, size of the waiting queue etc.) are low-level details. But these low-level details are needed to perform realistic performance predictions. Model completions are model transformations that offer a solution to this problem. Performance model completions are model completions that specifically addresses quality attributes of a software.

The first definition of model completions was given in 2002 by Woodside et al. [WPS02]. They identified model completions as :

Definition 4 (Model Completion) *A Model Completion is a process that closes the gap between abstract architectural models and required low-level details.*

Kapova [Kap11] developed an approach, named CHILIES, that handles performance model completions in the PCM, using HOTs. To allow variability in the HOTs, a SPL for the transformations has been created. This SPL takes as input the configuration of the wanted

transformation and produces as output the required transformation. Doing this, the generation of transformations can be automated. At this stage, the term model completion was redefined.

Definition 5 (Model Completion) *A Model Completion is a configurable purpose-specific transformation increasing model completeness while maintaining the language of the abstract level.*

The result of this approach is a set of Eclipse plug-ins that adds performance low-level details to models developed in PCM [Cha13]. A completion library is provided too. This library allows to reuse expert knowledge and improve the accuracy of performance predictions.

In contrast to other approaches reviewed in Section 3.1, the main advantages of the approach provided by Kapova are the provided completion library and its configurability.

The approach realized by Kapova allows to perform automatically all performance model completions we want, if the corresponding transformation fragments are available in the library.
In Section 5.2.2 we deal with another realization of model completions that addresses only one specific configuration: two components, connected to each other, are deployed on two different resource containers. This completion, realized in Java, simulates the network the components will use to exchange information.

3. Related Work

In this thesis we chose to compare several realizations of model completions. One is realized by HOTs implemented in QVT-R and ATL. One is implemented in Java. Another uses a model weaving approach, in which we introduced variability. In this section we review other existing model completions, uses of HOTs, model weaving approaches and the introduction of variability in model weaving approaches.

3.1. Model Completions

Wu and Woodside [WW04] describe a first approach, where performance completions are added manually to models. These completions have the form of performance specific annotations and can be used only in the domain of performance predictions. They identify as future work the specification of a library for elementary components like databases, the definition of rules to add these completions into the models and the automatization of this process.

Verdickt et al. [VDGD05] and Grassi et al. [GMS06] define approaches that realize model completions using model driven technologies. The approach developed by Verdickt et al. proposes a framework that injects a construct similar to completion into the models and maps high-level platform-independent UML models to other platform-specific UML models. Doing this decreases the level of abstraction of the output models. Grassi et al. use the principle of classical vertical model refinements. During this refinement, aspects of performance and reliability are added to the models. The biggest drawback of these two approaches is their lack of configurability. The first approach does not let the user configure the construct injected into the models and the second one does not allow the configuration of the model transformations used. As a result, all elements of a given type in the input model will be replaced by the same completion with the same configuration in the output model.

Happe et al. [HFBR08] suggest another approach focusing on the parametrization of completions. Parametrization differs from configuration as parameterizing a completion means completing the added elements with parameters. However elements used to perform the completion are the same. Configuring a completion, the elements used to perform the completion are selected among available elements. To parametrize a completions, Happe et al. analyze a completion for Message-Oriented Middleware and implement the resulting completion using Java, mark transformations and then model driven techniques. The

major disadvantage of this approach is that the completions always operate on the same subsystem. Thus it is not possible to design a model where given services are not supported by a particular platform. This approach focuses on the parametrization of completions on one hand but does not consider the variability of the model skeleton on the other hand.

Happe et al. [HBR+10] implemented in 2010 performance completions into PCM. Earlier, these mechanisms of performance completions were specific to the middleware vendor. This means that for each middleware vendor a specific completion had to be implemented. These completions were neither configurable and it was not possible to specify different configurations of an execution environment. The approach developed by Happe et al. introduces configuration options and abstracts from platform specific details using a test driver. This driver collects on a specific platform all measurement data needed to instantiate a completion for this platform. Once the completion's behavior and the configuration options have been fixed, a M2M transformation changes the architectural model into a performance model. The most significant limitation of this approach is that completions have to be calculated for each platform.

3.2. Higher Order Transformation

Tisi et al. [TJF+09] propose a classification for HOTs. They detect that ATL was, in 2009, the most used language to write HOTs and thus gathered a set of 44 transformations written in ATL. The process of a typical HOT is given: first, an ATL injector converts the input of the transformation into an ATL input model. Then the HOT is executed on this ATL model and produces an ATL output model. This model is finally converted, with an ATL extractor, into the output model of the transformation.
Tisi et al. decompose the HOTs into four base patterns: transformation synthesis, analysis, (de)composition and modification. Fig. 3.1 illustrates this decomposition. Whereas a transformation synthesis takes inputs that are not transformations and produces a transformation, a transformation analysis takes a transformation as input and produces something else than a transformation. Transformation compositions take as input several transformations and produce a single transformation. On the contrary, a transformation decomposition takes as input a single transformation and produces several transformations. Finally, a transformation modification has exactly one transformation as input and one transformation as output. The HOT we implemented in this work corresponds to the composition base pattern, as the input for the transformation consist of several transformation fragments and the output is made of a single transformation, resulting from the composition of the input transformation fragments.
Tisi et al. identify four application areas for HOTs that have not been explored at the time of the paper: transformation metrics (analyze a transformation to derive metric values), transformation refactoring, transformation optimization (produce an equivalent version of the transformation that improves the speed or memory management) and partial evaluation (detect input models that are not modified and hard-code them into the transformation).

After distinguishing between external and internal compositions, Wagelaar [Wag08] focuses on internal composition for ATL: multiple transformation definitions are combined into a single transformation definition. This implies that all the transformation definitions are written in the same transformation language. In ATL, each transformation is characterized by a module and thus, these modules are combined during the composition. To merge the modules, superimposition is used. Let us consider the case where a module A is superimposed on a module B to obtain a module C. As a result, C contains the intersection of the sets of the language elements of A and B. For all comparable language elements, C also contains the language elements of the module A. In ATL, the term language elements

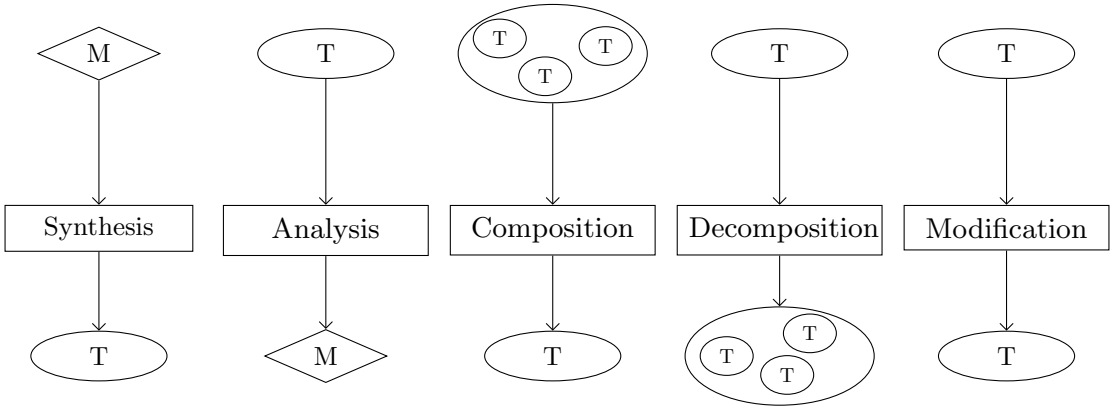


Fig. 3.1.: HOTs base patterns identified by Tisi et al. [TJF+09]

designates matched, lazy and called rules, helper attributes and methods. Comparable language elements refer to the same type of language element (a matched rule can not be compared to a called rule), with the same name and the same context for helpers. This superimposition is written in the form of a HOT that takes two ATL modules as input and produces one ATL module.
The principle of the superimposition can be adapted to any rule-based transformation language. For example, for QVT-R, the concept of "transformation" fits the one of ATL "module" and "relation" is equivalent to "rule".

A major drawback of QVT-R for refinement transformations is the fact that QVT-R does not support in-place transformation, for example similar to ATLs refinement mode. Thus, to perform refinement transformations in QVT-R, the user has to define manually copy rules for all model elements. This task is time-consuming and error-prone. Goldschmidt and Wachsmuth [GW08] define a generic pattern for copy rules in QVT-R and define a HOT, based on this pattern, that generates copy rules from a given meta-model. The pattern defines a top relation for every class of the meta-model. A specific behavior is defined to copy correctly sub-classes. Similar relations are defined for each attribute and reference, with an additional condition: the copy of the referenced classes has to be performed before we copy the attribute or reference. Goldschmidt and Wachsmuth implement this pattern in QVT-R, and thus the produced HOT can be applied to any Ecore meta-model. The HOT runs through the input model and produces suitable copy rules for every model element.
Goldschmidt and Wachsmuth define two possibilities to override the copy transformations in order to specify refinement transformations: extend manually the refinement transformations or use an automatic weaving. If the transformations are extended manually, the major difficulty is that, for each modified copy rule, the calls to the other non modified copy rules must not be broken and have to be manually specified. The second possibility offers to mark all rules to refine and, using a HOT, to translate the refinement rules into rules that conform to the expected calls to the other copy rules.
However, these techniques stay more complicated than a direct support for in-place transformations, where no copy rules need to be generated or modified.

3.3. Model Weaving

In this section, we discuss existing model weavers and list their respective advantages and disadvantages compared to GeKo, the model weaver we used, described in Section 2.2.1. We observed that various approaches are designed to fit only models designed in UML, or that the realizations of these model weaving approaches use only UML, while other model

weavers are able to support models designed in various modeling languages. We classify the existing model weavers according to this criterion, because we need to choose a model weaver supporting models designed in various modeling languages.

3.3.1. Weaving Approaches Restricted to UML

Many of the existing weaving approaches use artifacts of UML. Thus their use is restricted to models designed in this particular modeling language.

The Theme approach, described by Baniassad and Clarke [BC04], is one of these approaches that only supports UML models. The Theme approach is divided into two segments: the first, Theme/Doc, is used during the analysis phase of a system and the second, Theme/UML during the design phase. Theme/Doc allows the user to identify cross-cutting behaviors. Theme/UML models the system and integrates cross-cutting behaviors into the system model. As Theme/UML was created as an extension for UML, the Theme approach supports only UML models.

Cottenier et al. [CvdBE06] designed the Motorola WEAVR approach to match the production environment existing at Motorola. This production environment already used UML behavioral models, that's why the model weaver has been realized as a profile for UML. More specifically, this model weaver can be applied exclusively on UML state machine diagrams.

The Kompose approach, defined by Reddy et al. [RGF+06] is based on UML class diagrams. This approach uses a primary model and various aspect models. All models are designed in the form of UML class diagrams and each aspect model defines a cross-cutting concern. The elements of the primary model are then merged with elements of the aspect models, comparing the names or signatures of elements.

In the approach named KerTheme, the merge between models is based on the comparison of elements name or signature. It is an extension to the Theme/UML approach and has been described by Morin et al. [BKB+08]. This approach uses Executable Class Diagrams, based on UML class diagrams, as structural view on the model, and scenario models based on UML sequence diagrams, as behavioral view.

Mouheb et al. [MAN+12] assess that many weaving approaches are presented from a practical perspective. They handle the theoretical foundations in order to offer complete and rigorous definitions for model weaving. They propose formal specifications for model weaving in UML activity diagrams and define a syntax for an activity diagram and an aspect, as well as the semantic and algorithms used during the matching and weaving phases. An outstanding property of this approach is the possibility to consider control nodes in the pointcut model and thus to specify constructs like alternative, loops etc. On the other side, this approach only enables to add or remove elements and has been created specifically for UML activity diagrams.

3.3.2. Weaver Implementations Restricted to UML

So far all approaches we have considered have been designed to match only UML models. We add in this category of UML restricted approaches three additional approaches, whose design isn't bound to UML but for which we are only aware of realizations using UML. These three approaches are based on graph transformations. This means that the input model is converted into a graph and the aspect is defined using graph rules. These graph rules define the transformation applied to the input graph that generates a target graph. The target graph is finally converted back into the desired format, a model instance of the meta-model defining the source model.

The first approach is the Almazara approach, described by Sánchez et al. [SFS+08], which is based on the use of Join Point Designation Diagrams. In theory these Join Point Designation Diagrams can use any modeling languages but we are aware of a single application of this Join Point Designation Diagrams using UML Profile.

The second approach, called MATA, converts the two input models into an attributed type graph and defines two types of graph rules, corresponding to the advice and pointcut models used in many model weaving approaches. The only realization we are aware of uses UML profiles and stereotypes to define the aspects.

The third approach we consider focuses on UML activity diagrams. Khennouf et al. [KHCB12] define an extended meta-model for the base model, that adds to the original meta-model an integer attribute, to identify each control node, and a boolean attribute for each node, to distinguish between aspect and base actions. The aspect is composed of pointcut and advice diagrams. The pointcut diagram contains a set of join points and the advice models use the extended base meta-model. The AToM3 tool is then used to generate automatically visual modeling tools for editing aspect and base models and to transform them. In this case, the transformation consists in applying a grammar rule, composed of 139 rules, to the base and aspects models and in generating automatically the composed model. To reuse this approach for another meta-model, the user needs to extend manually the meta-model and redefine the rules.

3.3.3. Language-Independent Weaving Approaches

The Atlas Model Weaver (AMW) approach, described by Del Fabro et al. [DFBV06] is used to create links between model elements and associations between links. Yet this approach does not perform any modification on the models. Therefore it can be used for model mergings, compositions and comparisons as well as for data translation or tool interoperability. However, we can't use this approach for model transformations. AMW is implemented as an Eclipse plug-in.
The AMW model weaver has been used by Besova et al. [BWWB12] to propose an approach that links models of a multi-layer system in order to translate the design models into analysis models. They exemplified their approaches by the generation of analysis models for PCM.

The XWeave approach described by Groher and Voelter [GV07] is based on EMF and therefore can weave models that are instances of Ecore and instances of these models. According to the layers of modeling shown in Fig. 2.1 these models and their instances correspond to meta-models of level M2 and models of level M1. In its initial implementation XWeave only supports additive weaving. It can't remove, modify or override elements of the base model. The XVar tool offers a partial solution that offers the possibility to delete code for unselected features. This solution is still incomplete because deleting code is not the same as modifying existing model elements.

The next model weaving approach can be adapted to various meta-models but these adaptations need to be done manually for each new meta-model considered. Kalnina et al. [KKS+12a] invented an approach to ease the use of model transformations in the industry. They use mapping between model trees to execute the transformations so that the transformation user does not need to have any knowledges about meta-modeling. The transformation expert specifies the tree types for a specific meta-model and then the user describes graphically the mapping between the source and target models. The mapping is then executed with "creates if not exists" semantic. For language-independent uses of this approach, the biggest disadvantage of this approach is that the tree specification has to be done manually for each new meta-model.

Another possibility would be the use of general-purpose model transformation languages, such as the Epsilon Merging Language, to merge models. This language uses a textual syntax similar to declarative model transformation languages like QVT-R and supports various transformation scenarios. Therefore we can define all weaving operations using such a language. As these languages are general-purpose languages, they are not specialized for in-place model refinements. Therefore they do not provide specific constructs for such transformations.

The SmartAdapters approach [MBJ+07] and GeKo were initially developed by the same team. They share many concepts and characteristics: SmartAdapters also supports arbitrary meta-models, aspects are defined by pointcut and advice models and th join point detection is built on top of the same platform. In addition to these models, the SmartAdapters approach defines a composition protocol. This composition protocol describes in detail which weaving operations will be executed and allows the specification of complex and specialized weaving operations. On the other side this composition protocol makes the description of easy weaving operations more complex, requiring a detailed description for these easy operations as well.

3.4. Variability Management in Model Weaving

This section reviews existing approaches that introduce variability in model weaving. The first approach presented inserts variability into the domain meta-models, whereas the following two support variability at the model level. One approach is applied on UML models and the other uses SmartAdapters, the model weaver we described in Section 3.3.3. Finally one paper introduces how to manage complexity introduced by variability in model weaving.

Morin et al. [MPL+09] use a reusable variability aspect, defined at the meta-model level, to weave variability into domain meta-models. First, a pattern is modeled to describe variability concepts and their relationships, independently from any domain meta-model. To combine this pattern with several meta-models, an aspect, based on this pattern, is modeled. In this case, the aspects have been designed using SmartAdapters, defining the graft and target models and the adaptations. Designing the pattern using an aspect first allows to keep the pattern reusable, as it stays independent from any domain meta-model. It also introduces variability in a semi-automatic way. Finally, this allows to design the meta-model and variability concepts separately and thus ease their evolution. This aspect is then woven into the domain meta-model and produces a new domain meta-model that contains both concepts from SPL and the domain specific concepts of the original meta-model. Using this new meta-model, a modeler is able to design models that contain variability.

Reusable Aspect Models (RAM), described by Klein and Kienzle [KK07], define aspect models with a high degree of reusability. In RAM, any functionality that is reusable is modeled as an aspect. The aspects define as well the structure as the behavior of an application, using UML class and sequence diagrams. To define aspects with a high degree of reusability, the application to model is split into basic functionalities. Each functionality is modeled in a different aspect and each aspect may depend on other aspects. Kienzle et al. [KAAF+10] describe the gains having aspects that describe only one precise functionality. The resulting aspects contain only a few elements and thus a modeler can visualize the whole aspect at once and understand fast the internal working of the aspect. Before weaving an aspect with a base model, all aspects that it depends on must be instantiated and woven into it. As a complex aspect can depend on many lower-level aspects, the lower-level aspects are first woven so that the final woven aspect does not depend on any other aspect. RAM offers a way to detect possible conflicts while weaving

all aspects into each other. It detects all aspects pairs that search for similar elements, all aspects pairs that possibly modify the same elements in different ways.
Reusable aspect models can be designed using TouchRAM, "a multitouch-enabled tool for agile software design modeling aimed at developing scalable and reusable software design models", described by Wisam et al. [AABS+13]. This tool provides the user with a library of reusable design models.

The aim of approach proposed by Lahire et al. [LMV+07] is to weave in different ways a same aspect. It introduces matching and adaptation variability. The approach is based on SmartAdapters and the composition protocol has been extended to support variability. An adapter can be specified with the key-word "derivable", meaning that it may present several alternatives. Inside this derivable adapter, an "alternative" clause can specify a choice between two or more variants, each variants containing adaptation target declarations and adaptations. Optional and constraint clauses are also introduced into the composition protocol to manage more variability constructs. Mutual exclusion and dependency constraints can also be specified to insure the consistency of the composition protocol. When deriving from a derivable adapter, an adapter has to specify all the options and variants it uses. So the derivable adapters can be reused in different contexts and adapted to the specificities of each context.

Morin et al. [MVL+08] define some rules to ensure that an aspect model with variability can be safely integrated into an existing model. These checks are both static and dynamic, to ensure that the composition can be performed and that the resulting model is correct. These checks are performed on SmartAdapters, and specifically on the extended version introduced in the previous paragraph. Variants can be incompatible for several reasons. A variant may depend on elements introduced by another, non-selected, variant either introduce elements that prevent other selected variants to be applied or the selected combination of variants may break the conformance with the meta-model or invalidate user-level constraints. Until now, the whole family had to be derived manually, the incompatibilities had to be detected by the modeler, that specified the corresponding mutual exclusion and dependency constraints. This process is time-consuming and may be error-prone.
The static and dynamic checks are performed in three steps:

1. an aspect model without variability is produced for each variant of the product family
2. a static check is performed on all elements of the produced aspect: pointcut and advice models and composition protocol. If inconstancy is detected in the aspect, the variant is declared inconsistent.
3. the produced aspect is applied to a base model that is generated using elements of the pointcut model, so that all pointcut model elements (and the corresponding advice model elements) will be matched during the composition. If an error is thrown during the composition, the variant is declared inconsistent.

At the end of the process a report is delivered which contains, for all inconsistent variants, the weaving step that failed and a detailed diagnostic. This information helps the modeler to modify the problematic aspects, or to specify additional mutual exclusion and dependency constraints, in order to make the whole family consistent.

4. Variability and Model Weaving

Section 2.3.2 describes how HOTs can be used to perform performance model completions in Palladio. An outstanding property of this approach is its configurability. For each completion, the configuration properties are specified using feature diagrams. This configurability solves a large number of problems with a solution that originally considers only a few cases. The model weaving approach we chose, described in Section 2.2, lacks configurability. The advice and pointcut models have to be manually defined for each specific case. The approach we define in Section 4.1 is inspired by the RAM approach described by Klein and Kienzle [KK07, KAAF+10] and introduces variability into model weaving. Section 4.2 describes the key concepts that have to be considered to ensure coherence between the configurable aspects. Section 4.3 explains how the variability is handled via a SPL. Finally, Section 4.4 discusses possible extensions of the introduced approach.

4.1. Introduce Variability in Aspects

The variability we introduce into the aspects is designed to correspond to the variability needed in performance model completions. However, the concepts have been conceived to be easily extended to meet other needs. The general idea is to design global advice and pointcut models that can be customized for a specific use case with configuration options.

In RAM reusable aspects are designed. To define aspects with a high degree of reusability, the application to model is split into basic functionalities. Each functionality is modeled in an aspect that may depend on other aspects. This way, a hierarchy of inter-dependent aspects is created. The distinct aspects are then woven pairwise to form a global aspect, wherein the hierarchy is flattened.

The formation of a global aspect weaving pairwise aspects is reused in this thesis. However, in our approach, the aspects do not represent basic functionalities but the configuration options. As a result, configurable aspect models are designed. Before applying a configurable aspect on a model, the configuration of this aspect has to be determined. Once the aspect configuration is chosen, the sub-aspects corresponding to this configuration are selected. A global aspect representing this configuration is realized, weaving pairwise the selected aspects.

Fig. 4.1 illustrates this principle with two aspects, A and B. All aspects are composed of pointcut and advice models. The pointcut models are symbolized with rhombuses and the advice models with pentagons. Aspect B configures the pointcut and advice models of

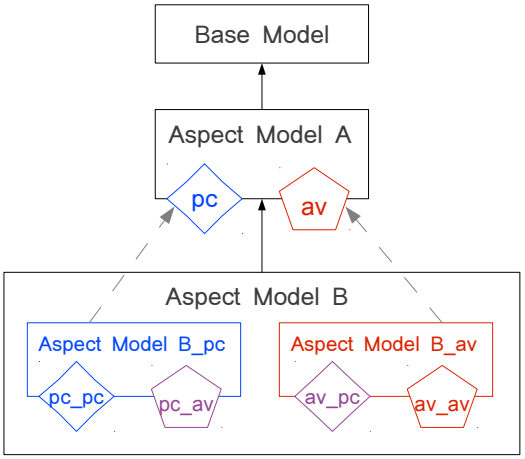


Fig. 4.1.: Weaving aspects into aspects

aspect A. Thus aspect B is composed of two sub-aspects named B_pc and B_av targeting respectively the pointcut and advice models of A. This way, aspect B is composed of four models:

- $B.\_pc\_pc$: pointcut model of the aspect that targets the pointcut model of A.
- $B.\_pc\_av$: advice model of the aspect that targets the pointcut model of A.
- $B.\_av\_pc$: pointcut model of the aspect that targets the advice model of A.
- $B.\_av\_av$: advice model of the aspect that targets the advice model of A.

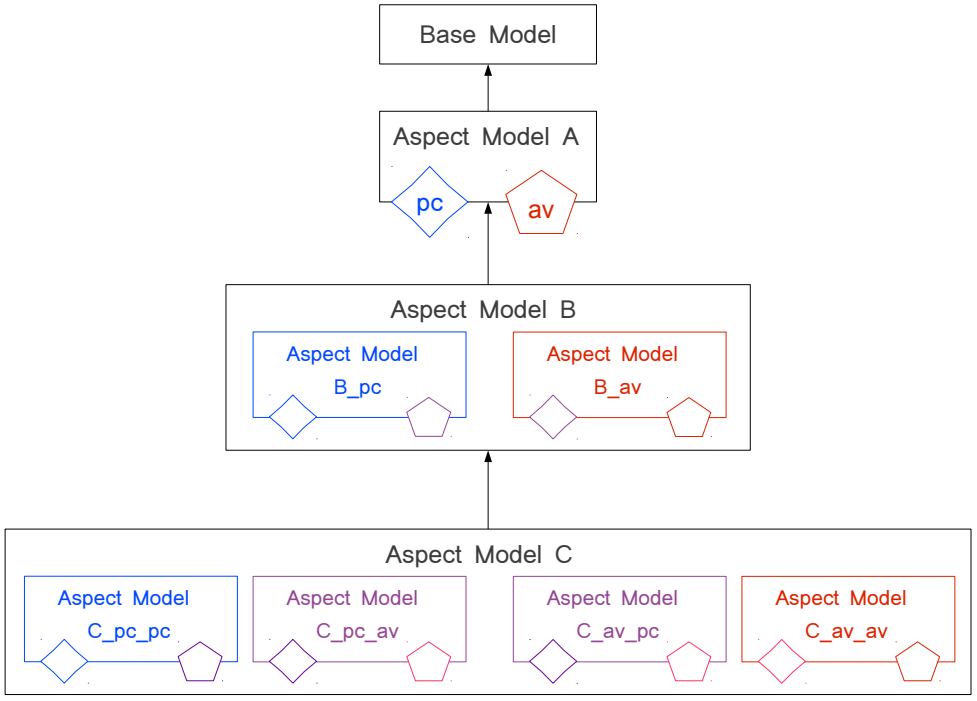


Fig. 4.2.: Aspects Hierarchy

Fig. 4.2 shows that this principle can be applied recursively. In this example, aspect C may specialize aspect B. C is composed of four aspects, each aspect targeting one of the four models of B. As each aspect is composed of two models, C contains eight models. Several

aspect models may correspond to an aspect level. Aspect B may have twins aspect models corresponding to the same aspect level. Aspect models stand for configuration option(s).

4.2. Coherence between the aspects

Configurable aspect models introduced in Section 4.1 depends on several sub-aspects. Section 4.1 introduces that several aspect models may correspond to a same level. The configuration determines which aspect(s) is(are) applied. When two different aspects are applied on a same aspect, we have to ensure that the produced aspect is correct and meet the requirements. If the produced aspect is correct, the two aspects are declared coherent relative to each other.

Section 3.4 introduces a method developed by Morin et al. [MVL+08]. Their approach aims to ensure that an aspect model with variability can be safely integrated into an existing model. To achieve this goal, all global aspects variants are created by weaving pairwise the contained sub-aspects. This way, the hierarchy of the global aspect is flattened. Static and dynamic checks are then performed on each global aspect. These checks ensure that the composition can be performed and that the resulting model is correct.

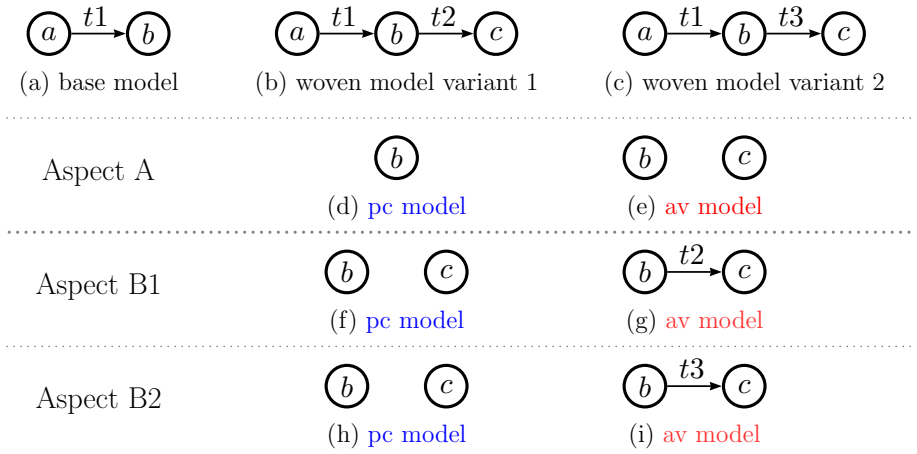


Fig. 4.3.: An example of model weaving with LTS models, where an aspect depends on two aspects

The approach developed by Morin et al. is complete as it detects all inconsistent variants of an aspect. However, this approach does not consider correctness regarding user knowledge. Fig. 4.3 depicts an example where the user defines a base model and two woven model variants, using Labeled Transition System (LTS). To obtain these two woven models, the user models three aspect models. The aim of aspect A is to add a c state for each b state found. Aspects *B1* and *B2* specify the name of the transition that links the two states. The approach developed by Morin et al. introduces a third variant by applying both aspects *B1* and *B2* to aspect A. The resulting woven model contains two transitions triggered by b and targeting c, *t2* and *t3*. This variant is coherent but incorrect regarding user knowledge. In this example, a third variant is declared coherent, even if it does not match the user requirements.

RAM [KAAF+10] permits to detect automatically that two aspects are potentially conflicting. The search for potentially conflicting aspects is limited to aspects on a same hierarchical level. Two potentially conflicting aspects are defined as aspects looking for the same element in the base model. A warning is thrown to the user for each pair of potentially conflicting aspects detected. It is then up to the user to decide if the aspects

are conflicting or not. In case they are conflicting, conflict resolution aspect models are used to fix the conflicts. The resolution aspects, designed in RAM, contain modification views and conflict criteria conditions. During the weaving, if a conflict criteria condition is met, the corresponding modification is applied on the aspect to raise the conflict.

The warnings thrown by RAM reports to the user all potentially conflicting aspects. When the combination of two sub-aspects introduces an unexpected behavior in the woven aspect, the woven aspect is declared incorrect regarding user knowledge. We assume that the combination of two sub-aspects introduces an unexpected behavior mainly if the sub-aspects modify the same elements. Thus we assume that incorrect aspects are detected in RAM as potentially conflicting aspects. This is the case in the example illustrated in Fig. 4.3. In this case, to make the designed aspects conform to the user requirements, aspects *B1* and *B2* must not be applied at the same time. This means that aspect *B1* excludes *B2* and the other way around.

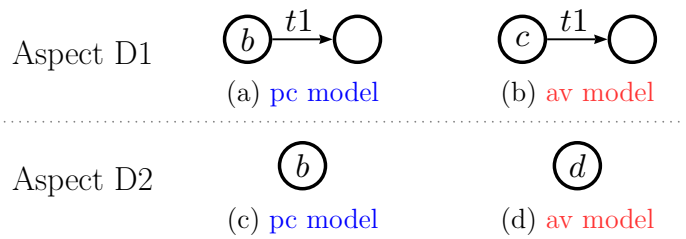


Fig. 4.4.: An example of model weaving with LTS models, illustrating the importance of weaving order

Another way to improve the coherence of inter-dependent aspects is to specify a weaving order. Fig. 4.4 defines two aspects *D1* and *D2*. These two aspects are applied on a arbitrarily aspect *A*. When weaving the global aspect, aspects *D1* and *D2* are woven pairwise into aspect *A*. No assumption is made about the weaving order. The weaving operations $A + D1$ and $A + D2$ are executed in a arbitrarily order. In this case, however, the order in which the weaving is performed has an influence on the woven global aspect. If aspect *D1* is applied first on *A*, a *b* state is renamed *c*, if it triggers a *t1* transition, and *d* otherwise. If *D2* is applied first, a *b* state is renamed *d* and the *D1* aspect performs no modifications. Cases where the weaving order is important are detected as potentially conflicting aspects. Specifying the order in which the weaving should be performed, the user makes the aspects conform to the user requirements.

None of these method to detect incoherence between the aspects is implemented in our approach. However, it is important to know the possible incoherences that can appear between aspects. During the design of the configurable aspect models, a special attention should be paid to these possible incoherences to avoid them. Section 4.3 introduces the way we handled the configuration of the configurable aspect models. Constructs are provided that allow to solve some conflicting aspect cases by declaring two aspects self excluding.

4.3. Variability Support

The considered approach is extended with SPL concepts to support variability. A SPL [PBvdL05] represents a family of related programs. Each program contained in a SPL can be derived from a feature model. A feature model conventionally consists of a feature diagram and cross-tree constraints. The feature diagram is a visual representation of the feature model. It is generally represented with an and-or tree. The sub-features of the feature diagram can be mandatory, optional or alternative. The cross-tree constraints are dependency and

mutual exclusion constraints. They are specified with the keywords *implies* and *excludes*. The constraints can be combined with *and*, *or* and *not* constructs to create more complex constraints. A feature configuration contains features selected in the feature model and describes a member of a SPL. The selected set of features have to satisfy all constraints specified in the feature model.

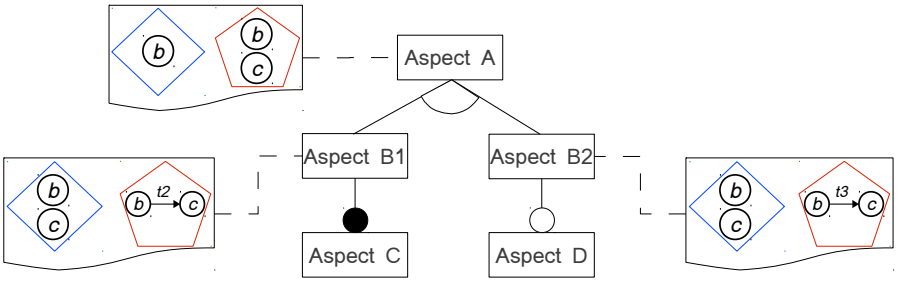


Fig. 4.5.: A Software Product Line for our LTS example

Fig. 4.5 shows the feature diagram created for the LTS example described in Fig. 4.3. The example is extended with two aspects, C and D. Each feature of the feature model stands for a configurable aspect. In this case, aspect C is a mandatory sub-feature of aspect *B1* and aspect D an optional sub-feature of aspect *B2*. The aspects *B1* and *B2* are designed as alternative sub-features for the aspect A. This mutual exclusion constraint makes the global aspect correct regarding user knowledge, as detected in Section 4.2.

The dependency and mutual exclusion constraints provide the user with constructs to solve incoherence and incorrectness problems highlighted in Section 4.2. The feature models represent the globality of the configurable aspect models. A feature configuration stands for a variant of a configurable aspect model. The aspects corresponding to the features contained in the feature configuration are then woven pairwise to build the global aspect.

4.4. Extensions

The aspects we consider in this thesis are composed of three models: pointcut, advice and mapping models. The pointcut and advice meta-models are similar as they are defined from refined meta-models of the application model. Our approach applies modifies the pointcut and advice models of an aspect. Mapping models conform to another meta-model. They are composed of mapping entries linking pointcut element(s) to advice element(s). Fig. 4.6 depicts the mapping of aspect *B1* from the example illustrated in Fig. 4.5. This mapping is composed of two mapping entries represented with black rectangles. Each mapping entry references pointcut and advice elements. The references to pointcut and advice elements are symbolized by colored rhombuses and pentagons. The pointcut and advice models containing the targeted elements are also represented in the figure. Thus, pointcut and advice models designed for a mapping model also contain pointcut and advice elements: the elements linked in the mapping entries. However, as GeKo is a generic tool, an aspect for mapping models can be easily designed.

Considering the example depicted in Fig. 4.5, aspect C may modify aspect *B1* in order to duplicate the state b. To achieve this goal, the first step is to apply an aspect on the advice model of *B1*, so that two states b are present in the advice model of *B1*. As two states can not have the same name in LTS, the added state is renamed *b1*. Applying at this time aspect *B1* to aspect A, a new state *b1* would be created in the advice model of A. However, the added state would not be a copy of state b but a new element added. Thus the new state does not copy all properties of state b. Fig. 4.7(a) shows the advice

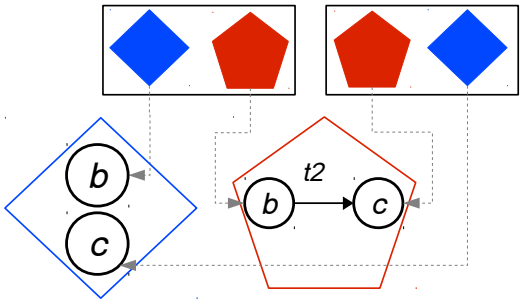


Fig. 4.6.: Mapping of aspect B1 from Fig. 4.5

model of aspect A that would be obtained applying aspect $B1$ on aspect A, in case aspect C modifies only the advice model of $B1$.

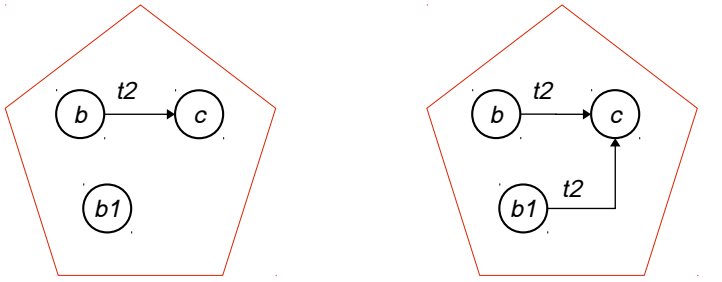


Fig. 4.7.: Advice model of aspect A, with (a) a new state added, (b) the state b duplicated

To duplicate state b, the mapping file of aspect $B1$ has to be modified. Fig. 4.6 depicts the original mapping file of aspect $B1$. The pointcut and advice elements are matched using XPath expressions relating their relative position in the models. This mapping contains two mapping entries. The first entry links the two states b, the second the two states c. Each entry links exactly one pointcut element to exactly one advice element and thus contains one source and one target. To duplicate the state b, the entry linking the two states b is modified. A second target is added to this mapping entry. Fig. 4.8 represents models of aspect C that modify the mapping file of aspect $B1$. To represent visually the mapping models, each entry is modeled with a rectangle. Each mapping entry is composed of rhombus(s) representing the pointcut element(s) referenced as source(s) and of pentagon(s) for the advice element(s) referenced as target(s). Pointcut and advice models are modeled with rhombuses and pentagons, according to Section 4.1. For readability reasons, the mapping between pointcut and advice models of aspect C is not represented in the figure.

The aim of the aspect depicted in Fig. 4.8, applied on the mapping file of aspect $B1$, is to duplicate the state b. Its pointcut model contains a mapping entry with one source element and one target element. The pointcut and advice elements targeted by this mapping entry correspond to the b states of aspect $B1$. The linked pointcut and advice elements are contained in the pointcut model too. The pointcut model also includes the advice element that corresponds to state $b1$ added in the advice model of aspect $B1$. The advice model introduces a new target in the mapping entry. This new target references the state $b1$. Applying the resulting $B1$ aspect to aspect A, the advice model depicted in Fig. 4.7(b) is obtained.

If aspect C has to perform changes on all three models of aspect $B1$, some precautionary rules must be followed. The three aspects composing aspect C are related to each other.

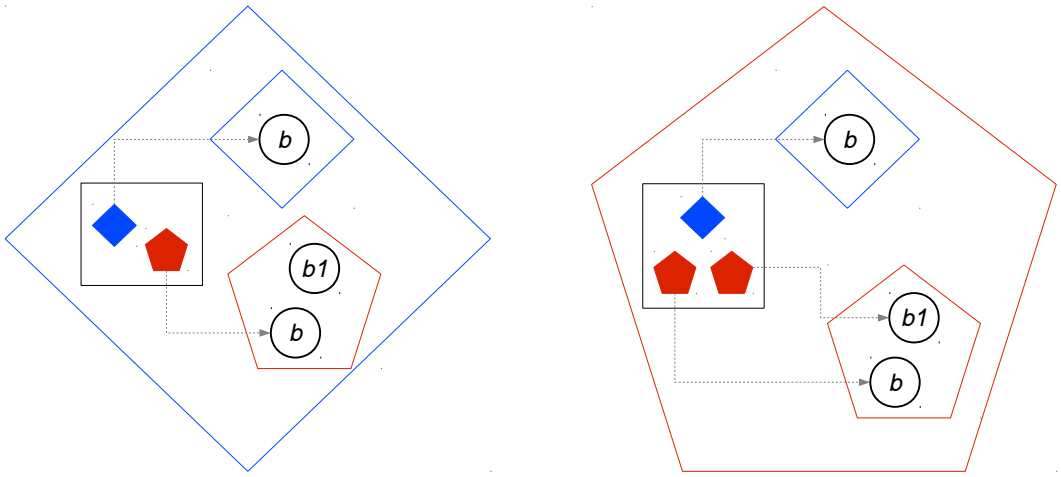


Fig. 4.8.: A part of aspect C: pointcut and advice models modifying the mapping file

Except adding or deleting an element, changes affect more than one aspect. To delete an element, only the pointcut model is updated. The advice model is the only one to update if a model element is added, and if this added model element does not reference or is referenced by any other model element. All other modifications are likely to modify the three models of aspect *B1* and thus the three aspects composing aspect C. Changes must be thinking globally.
In addition to this, the order in which the three aspects of aspect C are applied to the three models of aspect *B1* matters. As the mapping model references both other models, pointcut and advice models of aspect *B1* have to be modified before modifying its mapping file.

In this chapter configurable aspect models have been introduced and two future work directions have been highlighted. A first improvement of this method would be to automatize the coherence checks between sub-aspects. The approach could also be extended to weave the mapping part of an aspect. Weaving all three models composing an aspect allows to satisfy more specific requirements.

5. Model Weaving Compared with Model Transformation in the Context of Performance Model Completions

This chapter compares the use of model weaving and model transformation to realize performance model completions. These two approaches are first compared on a conceptual level in Section 5.1. Section 5.2 pursues the comparison with two case studies. The last section of the chapter, Section 5.3 introduces the limitations of each approach due to tool support.

5.1. Conceptual Comparison

In the following part we compare on a conceptual level model weaving and model transformations. More specifically we compare the HOTs we introduced in Section 2.1.3 and the model weaving approach including variability we introduced in Chapter 4.

As we consider a model weaving approach including variability, the input for both model completion realizations are the input models of the application, the configuration files and a library of transformation or aspect fragments. In both approaches the variability is realized using SPLs. Thus the configuration files specify which features are selected, as well as a value for the attributes. The transformation or aspect fragments used to build the global transformation or aspect are selected according to the chosen features in the SPL.

Except for this configuration step, the concepts used in the two realizations are different. After showing how both approaches differ in their representations, we compare the length of a same operation implemented using the two different approaches and discuss the expressiveness allowed by the constructs of each approach.

5.1.1. Representation

The first major difference between model transformation and weaving concern their representation. Model weaving has a graphical representation whereas model transformations have a textual representation. Model transformations are described using model transformation languages, defined in Section 2.1.2. The meta-model of the language is fixed for each transformation language and does not depend on the application meta-model.

This means that the constructs used to implement the transformation depend only on the transformation language. Two transformations implemented in the same language but that specify model completions for two different meta-models will look similar. If they are implemented in QVT-R, both will contain relations containing “checkonly” and “enforce” domain patterns for example. To be able to read a transformation, the first things the user needs to understand are the transformation constructs and thus the transformation language meta-model. Once the global structure of the transformation is clear, the user will focus on the application meta-model to identify the changes performed by the different steps of the transformation.

On the contrary, two aspects designed for two different applications do not look similar. Indeed, as described in Section 2.2, the original modeling languages and tools are used to design the aspects. A drawback of model weaving is that the relaxed meta-models that form the meta-models of the pointcut and aspect models have to be redefined for each new application considered. On the other side, a user familiar with the modeling language is able to understand quickly the mean of the aspect, as the latter is specified using the modeling language he is used to.

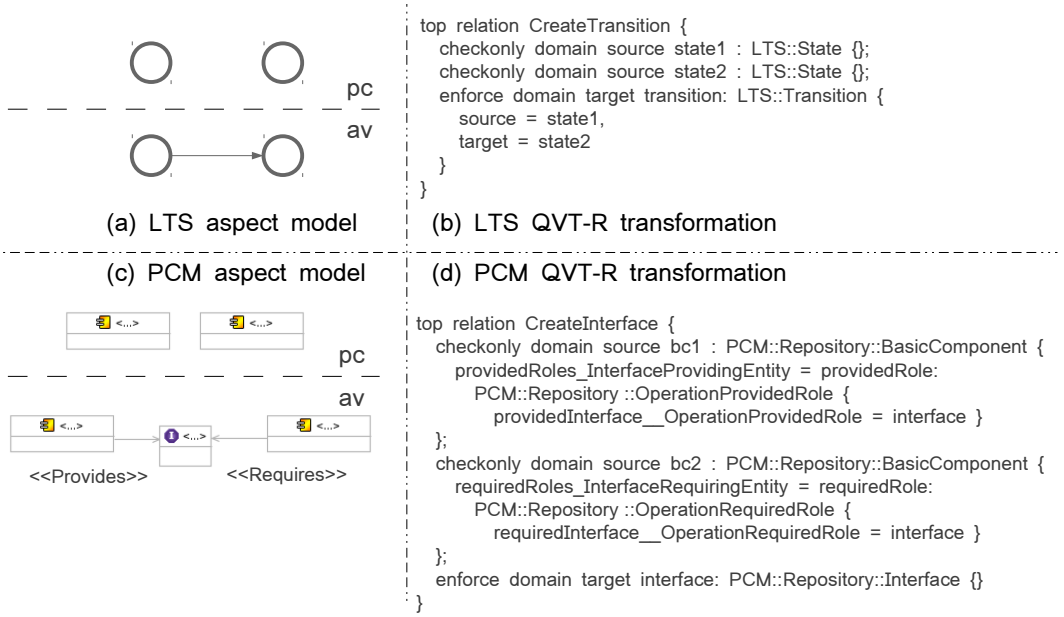


Fig. 5.1.: Element addition for LTS ((a) and (b)) and PCM ((c) and (d)) using model weaving ((a) and (c)) and model transformation ((b) and (d))

Fig. 5.1 shows an example of a simple model modification, the addition of a basic element. This addition is realized for two different meta-models, LTS and PCM, and using two different concepts, model weaving and a model transformation language, QVT-R. For space reasons we chose not to represent the copy rules implemented in the QVT-R transformation. With a basic example like this one, both realizations are understandable at a glance. However it illustrates that model transformations can be globally understood without knowing in detail the application meta-model, whereas people familiar with the modeling language will understand the overall meaning faster if model weaving is used.

5.1.2. Verbosity

The previous part shows that model transformations and weaving produce two distinct representations of a same operation. In the following part the expressiveness of model transformations and weaving are discussed.

Only three operations are realizable in model weaving: add, remove or update an element. The adding operations correspond to the elements included in the advice model but not in the pointcut model. On the contrary, if an element can be found in the pointcut model but not in the pointcut, it will be deleted. Three configurations are possible for an update operation: if a pointcut element corresponds to exactly one advice element (one-to-one mapping), its properties will be updated so that the properties of the woven element correspond to the advice element. Or, if one pointcut element corresponds to two or more advice elements (one-to-many mapping), the pointcut element will be duplicated. The last possibility is that two or more pointcut elements correspond to one advice elements (many-to-one mapping). In this case, the pointcut elements will be merged. The modeler creates the wanted behavior by combining configurations of the three operations described above.

Using model transformations, the global structure provided by the transformation languages has to be respected. All constructs offered by the language are combined to match the wanted behavior.
Using QVT-R, the developer describes the transformation by defining relations that contain "check-only" and "enforce" object patterns. Each object pattern specifies elementary changes that have to be performed on a given model element.
Using Java, the developer captures the transformation behavior by splitting it into several methods. Each method performs coherent changes on a set of model elements, using Java keywords and classes and methods offered by the API.

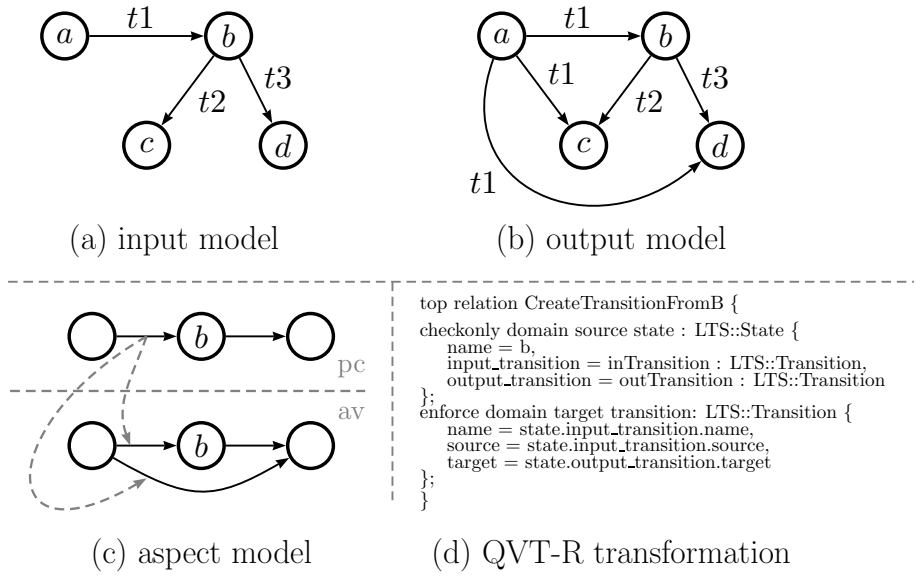


Fig. 5.2.: An example of a model transformation ((a) and (b)) realized using model weaving (c) and QVT-R, a model transformation language (d)

Fig. 5.2 illustrates how model weaving, Fig. 5.2(c), and model transformation languages, Fig. 5.2(d), handle in different ways the same problem. The input and output models used in this example are described in Fig. 5.2(a) and Fig. 5.2(b). The input model contains four states, a, b, c and d, linked with three transitions, $t1$, $t2$ and $t3$. Two occurrences of the $t1$ transition are introduced in the output model. The behavior showed with these modifications is: for each pair of transition (tX, tY), where tX targets the state b and tY is triggered by b, add an occurrence of tX, triggered by the state that triggers tX, and that targets the state targeted by tY.

Fig. 5.2(c) shows this transformation realized using model weaving. The realized aspect is composed of a pointcut and an advice models. For readability reasons, the mapping between pointcut and advice models is incomplete, all links between non-modified elements are missing. Only the duplicate mapping, that implies the adding of a new component, is represented. When applying this aspect on the base model represented in Fig. 5.2(a), a join point detection will first be executed: in the base model search for the pattern designed in the pointcut model: a state named b and two transitions. The first transition, called T, has to be triggered by b and targets an other state, denoted by S. The second transition targets b. In this case, two join points are detected. The first join point is composed of the transitions $t1$, corresponding to T, and $t2$. The state denoted by S corresponds to the state targeted by $t2$, c. The transitions $t1$, corresponding to T, and $t3$ form the second join point. The state S corresponds to the state d.
Then, for each join point detected, the T transition is duplicated. In our example, this corresponds to two duplications of the $t1$ transition. These two transitions are then updated to target the S states, which are c for the first duplication and d for the second one. The woven model is the model depicted in Fig. 5.2(b).

Fig. 5.2(d) contains the lines of code needed to perform the same model transformation using a declarative transformation language, QVT-R. For space reasons we chose not to represent all copy rules needed to perform this model transformation. The transformation is composed of one unique top relation that defines, in its first object pattern, a triplet composed of a state named b and two transitions, one triggered by b, the other targeting b, called T. In this example, the object pattern matches twice the base model. The elements contained in these two matches correspond to the elements that form the join points detected model weaving. Every time the triplet is matched in the base model, the second part of the relation is called: a new transition is created, named after the T transition matched. Its source state is the same as the source state of T and its output state is the output state of the second transition matched.

In this precise case, the mechanisms used to detect the points where the modifications should be done are the same. However the constructs used to perform these modifications differ. Using model weaving, we duplicated a transformation contained in the base model and then modified the duplicated occurrences of this transformation so that they conform the required output model. Only the target state of the matched transitions had to be updated. Using QVT-R, a new model element is created and all its properties are set according to the required output model. Model weaving and model transformation languages are able to perform the same model transformations but use different logics to achieve it.

5.2. Case Studies

This section analyzes several realizations of two different model completions. The two model completions considered are designed for PCM models and thus add low-level details, needed to perform correct performance predictions, to PCM models representing the coarse-grained architecture. The first model completion, the thread pool completion, adds, to a thread pool element, the low-level details needed to perform performance predictions: management of threads and characteristics of the waiting queue. Section 5.2.1 describes this completion. The second completion considered, the connector completion, is introduced in Section 5.2.2. In case two assembly contexts are linked with an assembly connector and deployed on two different resource containers, they communicate through a network. Some characteristics of the network are needed to perform correct performance predictions. The connector completion simulates the network. Finally Section 5.2.3 compares the various implementations of both considered model completions and concludes the case studies.

5.2.1. Thread Pool Completion

The thread pool completion adds to a thread pool element low-level details needed to perform accurate performance predictions. This completion is described in Section 5.2.1.1. Three realizations of this model completion are compared, using QVT-R, ATL and GeKo. These three implementations are introduced respectively in Section 5.2.1.2, Section 5.2.1.3 and Section 5.2.1.4.

5.2.1.1. Description

The thread pool completion is based on the thread pool pattern, used by many multi-threaded applications. This thread pool pattern is described by Kapova [Kap11]. A thread pool is implemented to manage several instances of the same resource. Running short tasks, a thread pool component allows to avoid the overhead induced if each task creates and destroys its own thread. Fig. 5.3 depicts the behavior of a thread pool with a petri net graph. Several threads are created in advance to perform a number of tasks. An incoming task is linked with a thread, if any thread is available, or stored in a waiting queue. Once a thread completes the task, it requires the next waiting task. When all waiting tasks are completed, threads can terminate or sleep until new tasks arrive.

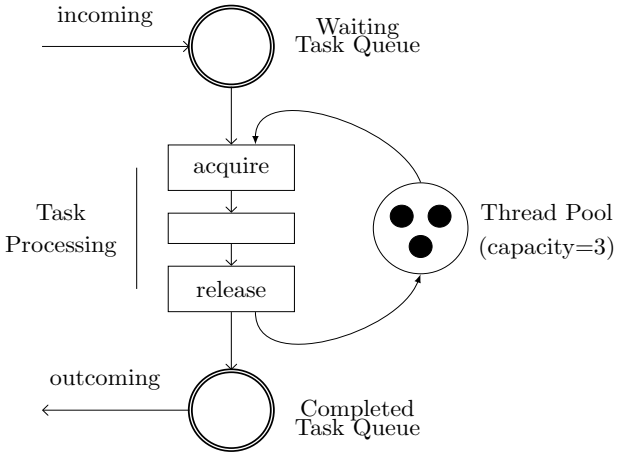


Fig. 5.3.: A thread pool with a capacity of three worker threads, from [Kap11]

Two factors influence the performance characteristics of a thread pool: how the threads are managed and the characteristics of the waiting queue. Fig. 5.4 illustrates options

considered in this thesis to configure a thread pool. These options and others are defined by Kapova and Reussner [KR10]. The threads may be managed statically or dynamically. If they are managed statically, the number of threads is set at the creation of the thread pool. The optional property KeepIdleTime defines the time a thread stays idle before being returned to the pool. The core pool size of a dynamically managed thread pool indicates the number of threads generated at the creation of the thread pool. At runtime, the number of threads is dynamically adjusted to the number of incoming tasks. Creating too many threads leads to resource wasting. To avoid it, a maximum pool size may be specified, at the creation of the thread pool. Time and resources are needed at the creation and destruction of threads. To ensure that threads are not destroyed and created again shortly after, unused threads may be kept alive for a short period. When the keepAliveTime is up, if the thread has not been needed to handle an incoming task, it is destroyed.

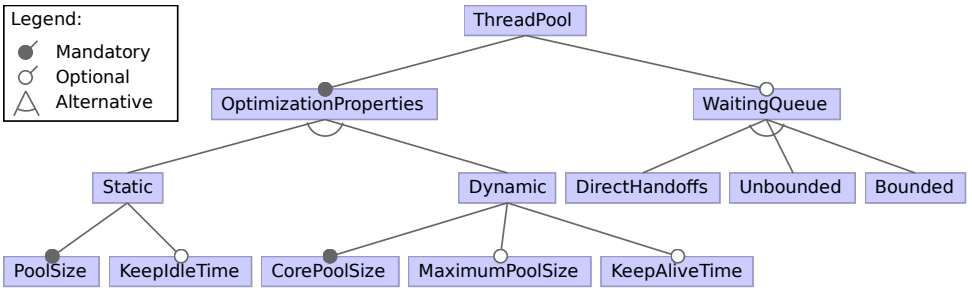
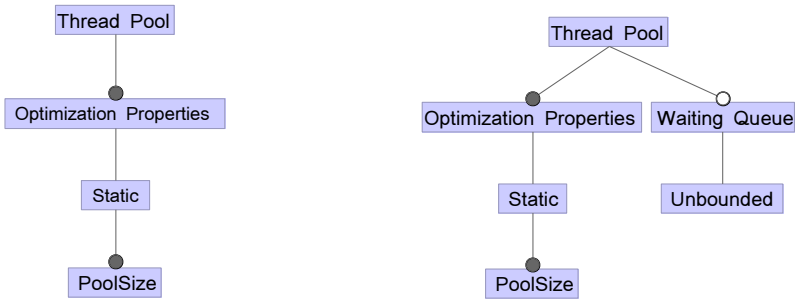


Fig. 5.4.: Part of a thread pool feature model of the thread pool completion, from [KR10]

The way incoming tasks are managed also influence the characteristics of the thread pool. Three different strategies may be chosen to handle a waiting queue. Using the direct handoff strategy, tasks are transfered to available threads without otherwise holding them. When a new task comes in, it is linked to an available thread. If no thread is available a new one is created, if it is possible considering the management of the threads. If no thread can be created, the new submitted task is rejected. Two situations prevent from creating a new thread: the thread pool is statically managed, or the maximum number of threads is already reached. If the thread pool possesses a bounded queue, the behavior of the thread pool is similar. Each incoming task is linked to an existing thread or to a new thread, whenever possible. However, if no thread can be created, the incoming task is added to the waiting queue, if its size is not exceeded. If the waiting queue is full, the incoming task is rejected. Using an unbounded queue and a dynamically managed thread pool, only the threads of the core pool are used. Indeed, if all core pool threads are busy, the incoming task is added to the waiting queue. If the thread pool has a static size, incoming tasks are added to the waiting queue if all threads are busy.

Thread management and characteristics of the waiting queue influence the performance characteristics of a thread pool element. These details are low-level details. They are not included in the models representing the coarse-grained architecture of a PCM instance. They are added to these models in order to perform correct performance predictions using performance model completions, described in Section 2.3.2. In the design models, elements to refine are selected. For each element selected, a configuration is specified: a set of valid features is selected from the feature model depicted in Fig. 5.4. Executing the model completion, components needed to simulate the behavior of a thread pool are added to each element selected, conforming to the chosen configuration. In this thesis, we discuss two configurations of the thread pool completion. The first configuration considers a thread pool characterized by a static size but without KeepIdleTime. The second configuration corresponds to a fixed-size thread pool with an unbounded waiting queue. Fig. 5.5 indicates



(a) fixed-size thread pool (b) fixed-size thread pool with an unbounded waiting queue

Fig. 5.5.: Feature models of the two considered configurations of the thread pool completion

feature models of each configuration and illustrates which features are selected in each configuration. Four features are selected to build a fixed-size thread pool: ThreadPool, OptimizationProperties, Static and PoolSize. If this fixed-size thread pool possesses an unbounded waiting queue, two more features are selected: WaitingQueue and Unbounded.

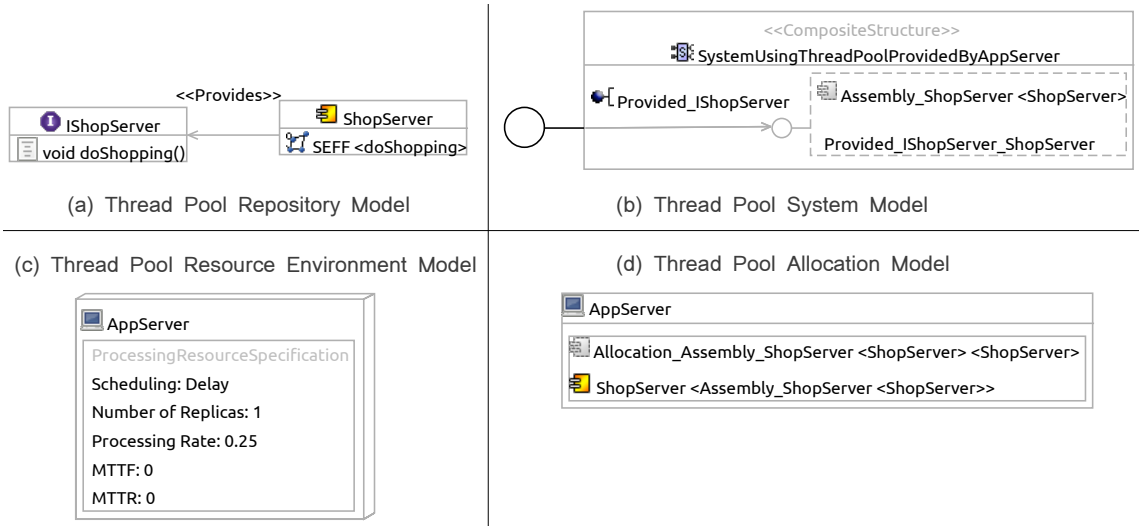


Fig. 5.6.: PCM instance on which the thread pool completion is performed

In the context of PCM models, three views of a PCM instance are refined by the thread pool completion: the repository, system and allocation models. Fig. 5.6 depicts the PCM instance used to perform the thread pool completion. Fig. 5.6(a) illustrates the used repository model. It contains one basic component ShopServer that provides the interface IShopServer. The interface IShopServer contains a signature named doShopping. The behavior of this signature is described in a SEFF for the basic component ShopServer. Elements are added to this element ShopServer to simulate a thread pool. The original system model, illustrated in Fig. 5.6(b), is composed of a system that contains an allocation context Assembly_ShopServer<ShopServer> encapsulating the ShopServer basic component. The system offers to the outside world a provided role corresponding to the provided role offered by the basic component ShopServer. A provided delegation connector is used to specify this correspondence. Fig. 5.6(c) shows the basic resource environment used in this example. It is composed of only one resource container, named AppServer. Finally, Fig. 5.6(d) depicts the allocation of the assembly context Assembly_ShopServer<ShopServer> on the resource container AppServer. This model also

highlights that the assembly context Assembly_ShopServer<ShopServer> encapsulates the basic component ShopServer.

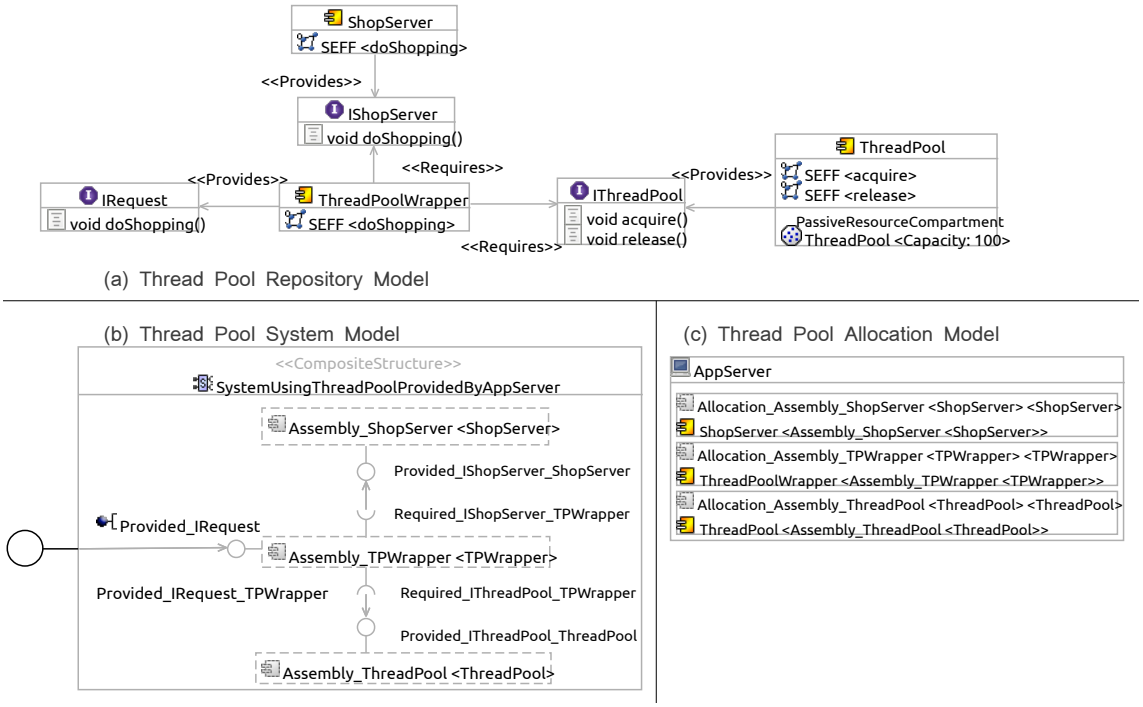


Fig. 5.7.: PCM instance depicted in Fig. 5.6 refined with the fixed-size thread pool completion

Fig. 5.7 depicts the PCM models obtained after applying the first configuration of the thread pool completion on the PCM instance introduced by Fig. 5.6. As the thread pool completion does not modify the resource environment model, it is not depicted in the figure. To simulate the thread pool, two basic components, ThreadPool and ThreadPool-Wrapper are added in the repository model illustrated by Fig. 5.7(a). The basic component ThreadPool contains a passive resource named ThreadPool that represents the pool size of a static thread pool. They communicate via an interface IThreadPool that contains two signatures, acquire and release. The basic component ThreadPoolWrapper provides an interface named IRequest containing an interface similar to the one contained in IShopServer. Both interfaces have the same name and thus provide the same service but they are described by different SEFFs. The SEFF describing the doShopping signature of the IRequest interface considers the influence of the thread pool on the performance properties of this signature. Fig. 5.7(b) shows the system model created. Each component of the repository model is encapsulated in an assembly context. The provided, required roles and delegation connectors replicate the dependencies defined in the repository model. All assembly contexts defined in the system models are allocated to the same resource in the allocation model, represented by Fig. 5.7(c).

Fig. 5.8 show the PCM models obtained after applying the second considered configuration of the thread pool completion on the PCM instance introduced by Fig. 5.6. Compared to the repository model obtained with the first configuration, depicted in Fig. 5.7(a), the repository model described in Fig. 5.8(a) contains one more basic component, representing the waiting queue. The IWaitingQueue interface links the added waiting queue component to the existing ThreadPool component. This interface contains two signatures. The first signature, addTask, is called by the ThreadPool component to store a task in the waiting

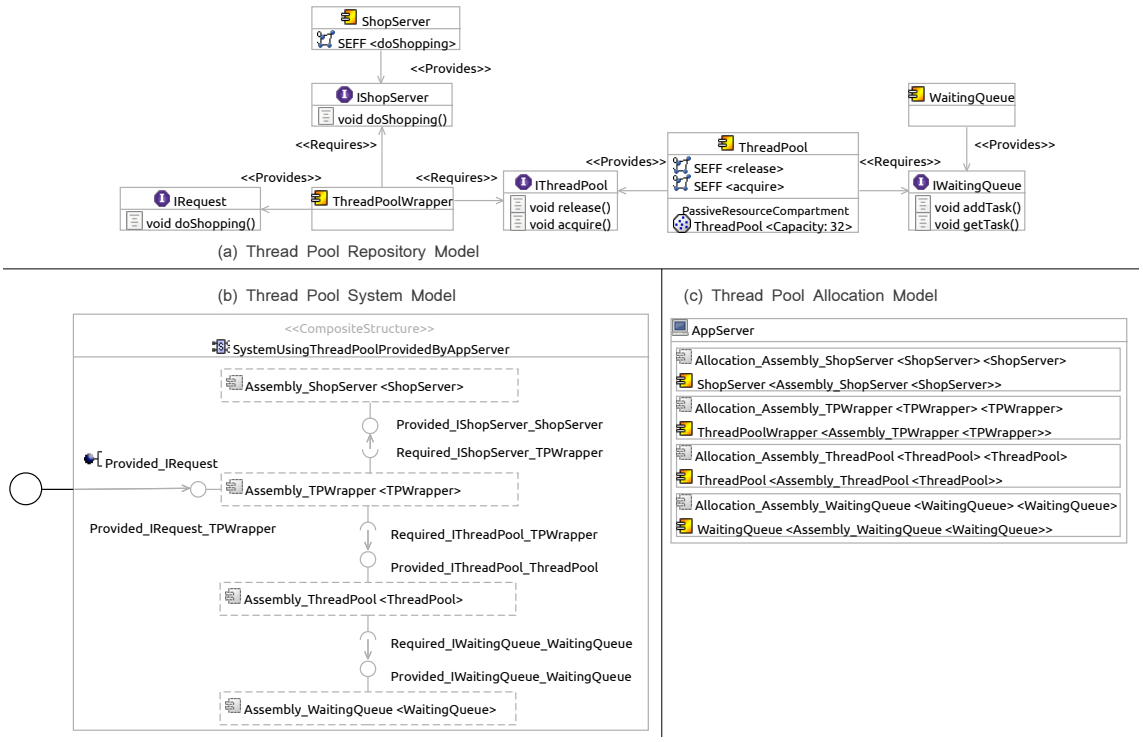


Fig. 5.8.: PCM instance depicted in Fig. 5.6 refined with the fixed-size unbounded-waiting-queue thread pool completion

queue. When a thread is released, the ThreadPool component calls getTask to get the next stored task to execute. System and allocation models, depicted in Fig. 5.8(b) and Fig. 5.8(c) illustrate the encapsulation and allocation of these new elements.

5.2.1.2. Implementation in QVT-R

The first implementation of the thread pool completion we consider is realized using HOTs implemented in QVT-R. This implementation has been produced in the context of the Chilies approach, described in Section 2.3.2. Using this implementation, only a fixed-size thread pool is introduced. This corresponds to the first configuration introduced in Section 5.2.1.1. Two transformation fragments are needed to perform this configuration of the thread pool completion. The first fragment corresponds to the root element of the feature model defined in Fig. 5.4, the ThreadPool feature. This first fragment contains rules that copy all elements and adds the two basic components ThreadPool and ThreadPoolWrapper and the two interfaces IThreadPool and IRequest. It also defines all provided roles, SEFFs and signatures of the final repository model. The second fragment corresponds to the feature PoolSize of the feature model and sets the size of a fixed-size thread pool. As the feature PoolSize is a mandatory child of feature Static, all changes due to these two features are combined in the transformation fragment corresponding to the Static feature. Lst. 5.1 shows this transformation fragment composed of a unique relation. This relation looks for every basic component named ThreadPool and adds a resource to store the size of the thread pool. This resource is composed of a random variable representing the value of the thread pool size.

A first QVT-R transformation builds the final transformation from these two transformation fragments. The first transformation takes as input the configuration of the wanted transformation. A feature model represents this configuration. The transformation combines all fragments corresponding to chosen features to build the final transformation.

```
1  transformation ThreadPoolRoot (source: pcm, target: pcm
       ) {
2    top relation CreateThreadPoolComponent {
3      checkonly domain source sourceBasicComponent:pcm::
           repository::BasicComponent{
4        entityName = 'ThreadPool'
5      };
6
7      enforce domain target targetPassiveResource:pcm::
           repository::PassiveResource{
8        capacity\_PassiveResource = ThreadPoolSize : pcm
             ::core::PCMRandomVariable {
9          specification = 'x'
10       }
11     };
12   }
```

Listing 5.1: QVT-R transformation fragment corresponding to the feature PoolSize

This final transformation is then applied on the base models. In our example the final transformation is applied on models represented in Fig. 5.6.

5.2.1.3. Implementation in ATL

The ATL realization of the thread pool completion we implement in this work is a HOT, like the QVT-R realization considered in Section 5.2.1.2. Unlike the QVT-R realization, the the first transformation of the ATL HOT is implemented using Java. This Java transformation takes as input a global ATL transformation and a feature model. The global transformation contains transformation fragments corresponding to all features. Transformation fragments corresponding to a specific feature are identified with annotations. The Java transformation removes fragments corresponding to non-selected fragments.

Lst. 5.2 shows the fragment of the ATL transformation that correspond to the feature PoolSize. The implementation of the same operation implemented in QVT-R is illustrated in Lst. 5.1. As a reminder, the feature PoolSize is a mandatory child of feature Static, and thus all changes are combined in the transformation fragment corresponding to the Static feature. In QVT-R the operation corresponding to the PoolSize feature is performed in a relation. This relation looks for a basic component called thread pool, created previously by an other fragment of the transformation. For each thread pool component found, an attribute specifying the size of the static thread pool is added. In ATL, the output model is a write-only model. Read operations can not be performed on the output model. Thus, perform the same operation is not possible.

In the ATL transformation, if the feature PoolSize is selected, the attribute representing the size of the thread pool is added at the creation of the thread pool component. This addition of an attribute at the creation of the thread pool component is represented by line 15 of Lst. 5.2. The characteristics of this attribute are then set by the code contained in lines 19 to 27. If the PoolSize feature is not selected, a first Java transformation removes these lines of code from the global transformation. Thus, the final ATL transformation does not contain lines of code corresponding to this modification.

The fact that ATL output models are write-only models leads to the major difference between QVT-R and ATL HOTs. The QVT-R HOT takes as input transformations fragments whereas the input of the ATL HOT is composed of a unique global transformation. This global transformation contains transformation fragments corresponding to all features. The ATL refining mode, introduced in Section 2.1.2.2, avoids to the user the

```
1  rule CreateThreadPoolComponent
2  {
3    from inputSystem : PCMsys!System,
4     [...]
5    to outputSystem : PCMsys!System
6     [...]
7       -- new ThreadPool Component
8       newTP : PCMrepo!BasicComponent
9       (
10        entityName <- 'ThreadPool',
11        repository__RepositoryComponent <-
              inputRepository,
12        providedRoles_InterfaceProvidingEntity <-
              providedRole
13
14        -- Static Begin
15        ,passiveResource_BasicComponent <-
              passiveResource
16        -- Static End
17      ),
18       -- Static Begin
19       passiveResource : PCMrepo!PassiveResource
20       (
21         entityName <- 'ThreadPool',
22         capacity_PassiveResource <- pcmRandomVariable
23       ),
24       pcmRandomVariable : PCMrepo!PCMRandomVariable
25       (
26         specification <- thisModule.getSize()
27       ),
28       -- Static End
29        [...]
30  }
```

Listing 5.2: ATL transformation fragment corresponding to the feature PoolSize

creation of rules that copy all elements. This is a significant difference in the implementation of ATL global transformation. The last difference between the two realizations is the language of the first transformation. In QVT-R HOT, the first transformation is realized using QVT-R, whereas in the ATL HOT, this first transformation is implemented in Java.

5.2.1.4. Realized using GeKo

The two realizations presented in Section 5.2.1.2 and Section 5.2.1.3 use model transformation languages to implement the thread pool completion. The approach introduced in this section use the model weaving approach described in Chapter 4 to realize this thread pool completion. Model weaving and model transformations are based on different observations and use different techniques. The realization presented in this section differs from the two implementations described previously.

Fig. 5.6 depicts the base model used for the thread pool completion. Fig. 5.7 and Fig. 5.8 illustrate the models generated by the two configurations of the thread pool completion introduced in Section 5.2.1.1. Section 5.2.1.4.a introduces steps performed to realize a fixed-size thread pool, corresponding to the first configuration. Section 5.2.1.4.b describes steps performed to realize a fixed-size thread pool with an unbounded waiting queue, corresponding to the second configuration.

5.2.1.4.a. Fixed-Size Thread Pool

These base and woven models show that the thread pool completion modifies the repository, system and allocation models of a PCM instance. However, using GeKo, the modifications of the three models are performed separately. As the repository model does not depend on any other models, we modify it first. Then, modifications are performed on the system model, that references elements of the repository model. Finally, the allocation model is modified, as it references elements of the system model.

First step consists in weaving the repository model depicted in Fig. 5.6(a). Fig. 5.9 illustrates the aspect fragments applied to this base model to update it with a fixed-size thread pool. To design this configuration, four features are selected: ThreadPool, Optimization-Properties, Static and PoolSize. The pointcut model of the ThreadPool aspect fragment is composed of a basic component, an interface and a signature. Only condition is that the signature is contained in the interface. Names of elements are not specified in pointcut and advice models, however, for readability reasons, elements depicted in figures need to be named. By convention names of unnamed elements start with the prefix *X_* followed by the type of the element. The advice model of the ThreadPool aspect fragment contains the three elements contained in the pointcut model unchanged. This advice model adds two basic components ThreadPool and ThreadPoolWrapper and two interfaces IThreaPool and IRequest. The interface IRequest contains a signature that is a duplication of the signature matched in pointcut model. These four added elements simulate the basic behavior of a thread pool.

As we design a fixed-size thread pool, the Static aspect fragment is applied on the advice model of the ThreadPool aspect fragment to configure it. As the feature PoolSize is a mandatory sub-feature of the feature Static, it is selected as soon as the feature Static is selected. The changes caused by these two features are gathered in the aspect fragment corresponding to the super-feature Static. The Static aspect fragment adds acquire and release signatures to the IThreadPool interface and their corresponding SEFFs in the ThreadPool basic component. This addition is due to the selection of the Static feature. But this aspect fragment also specifies the thread pool size attribute corresponding to the Static feature.

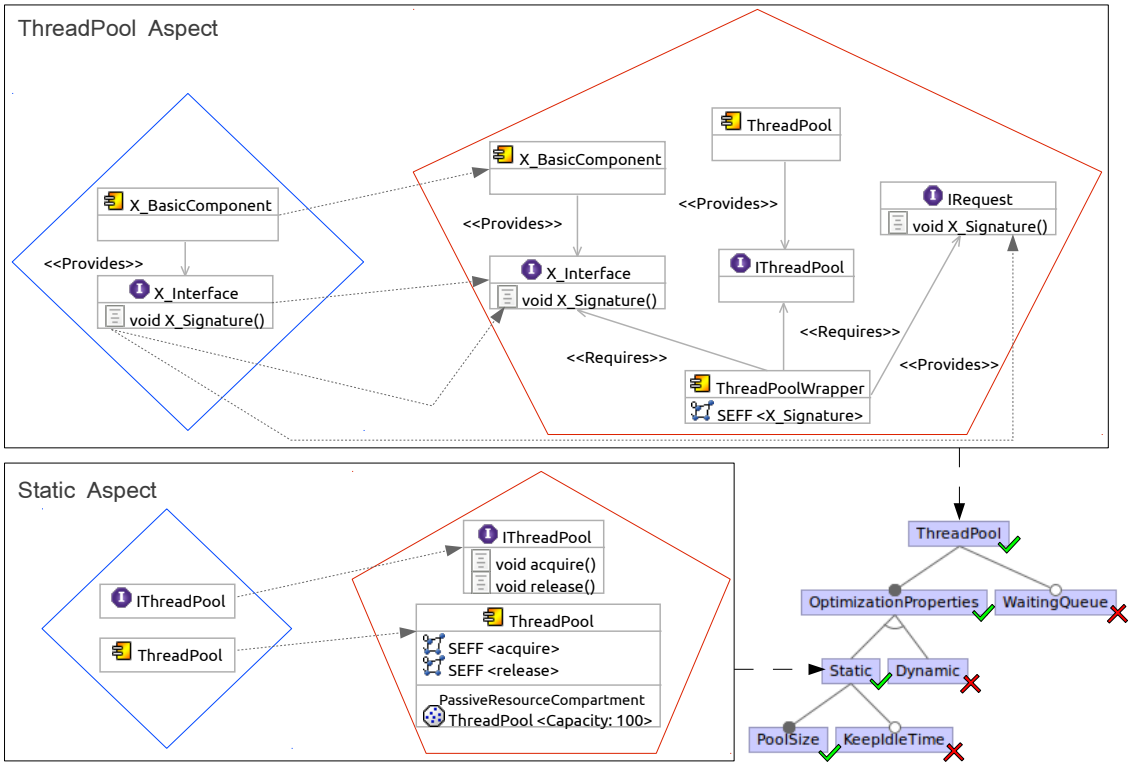


Fig. 5.9.: Fixed-size thread pool feature repository model and its corresponding aspects

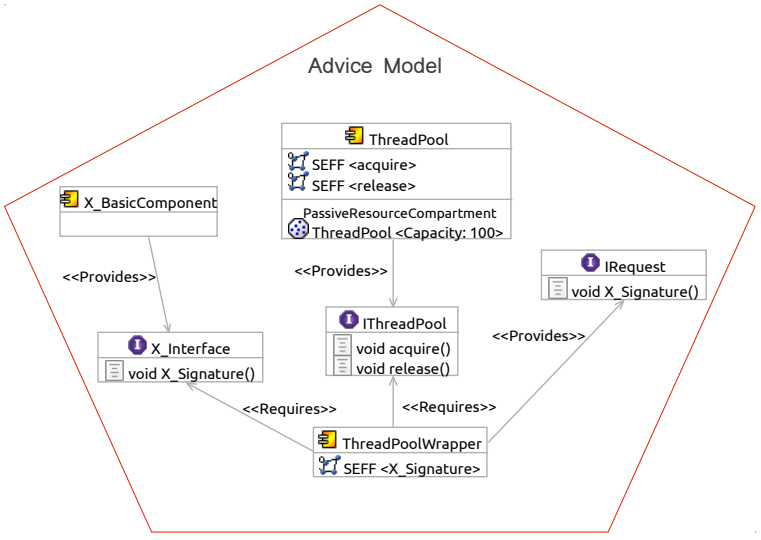


Fig. 5.10.: Advice repository model resulting of the application of the Static aspect fragment on the advice repository model of the ThreadPool aspect fragment

As described in Section 4.1, the Static aspect fragment is first applied on the advice repository model of the ThreadPool aspect fragment. Fig. 5.10 shows the resulting advice repository model. The pointcut and mapping models of the ThreadPool aspect fragment are not modified. The resulting ThreadPool aspect fragment is then applied to the repository base model depicted in Fig. 5.6(a) to produce the repository model depicted in Fig. 5.7(a).

The second step of the thread pool completion consists in weaving the system model. At

this stage of the completion execution, we consider the woven repository model as a part of the base model. The considered base model consists of the repository model depicted in Fig. 5.7(a) and the system model depicted in Fig. 5.6(b). System aspect fragment corresponding to the four selected features are woven to produce the final system aspect to apply on the base model. However, only the feature ThreadPool is linked to an aspect fragment. Indeed, the other features selected modify the inner structure of the components. These modifications have no influence on the system model. Thus the final aspect is the same as the aspect fragment corresponding to the ThreadPool feature.

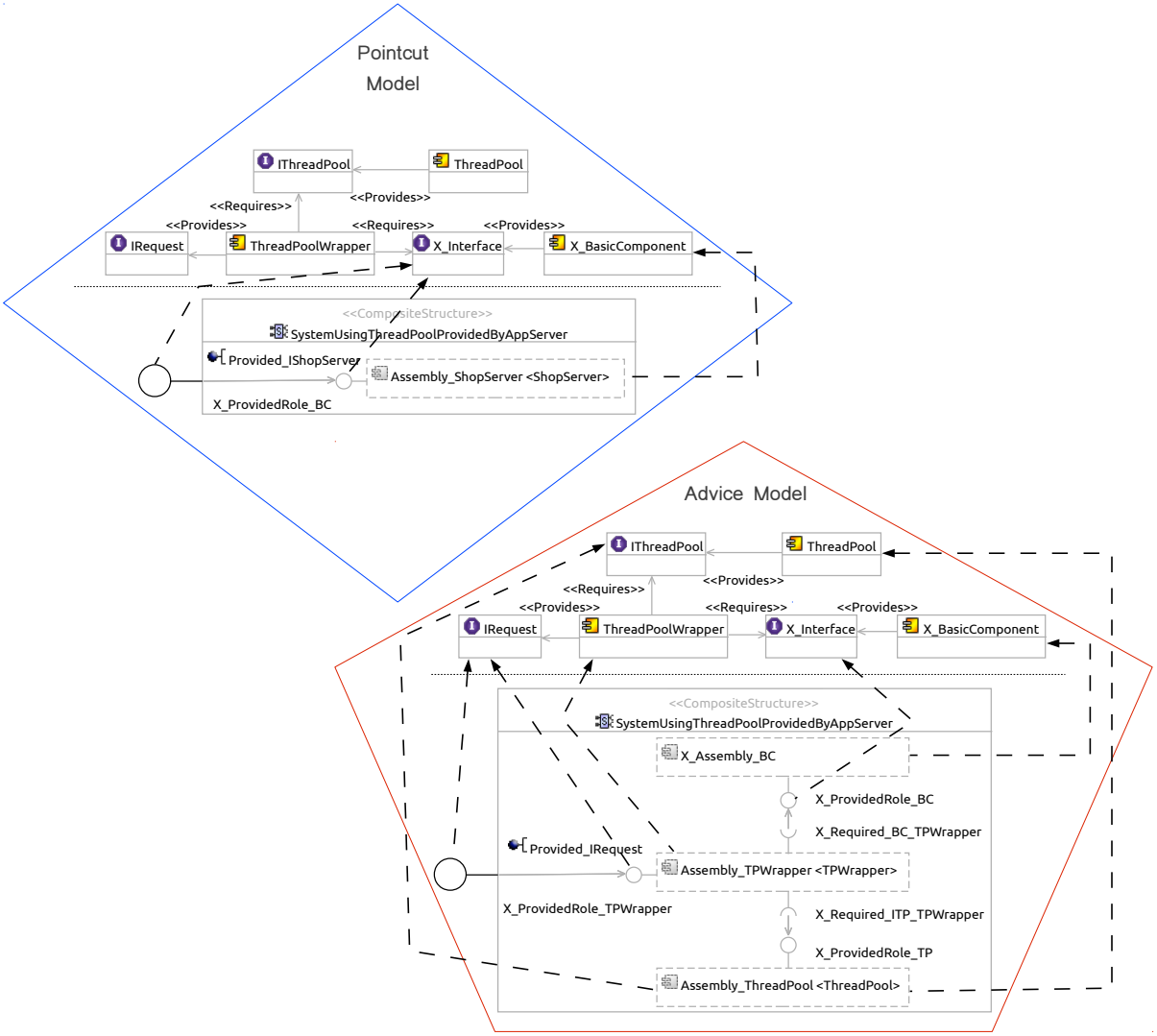


Fig. 5.11.: System aspect fragment corresponding to the feature ThreadPool

The aspect applied on the system base model is depicted in Fig. 5.11. For readability reasons, the mapping between the pointcut and advice models is omitted. We focus on the references between the repository and system models. The repository model is not modified during this weaving phase, the same elements appear in pointcut and advice models. However, repository model elements targeted in the pointcut model are needed to build the advice system model. The assembly context Assembly_TPWrapper, created during the weaving, encapsulates the basic element ThreadPoolWrapper contained in the repository pointcut model. Similarly, the two added assembly connectors link provided and required roles defined in the repository model. At the end of this weaving phase,

the PCM instance is composed of the woven repository and system models depicted in Fig. 5.7(a) and (b). However, the allocation model has not been modified yet and is still the model depicted in Fig. 5.6(d).

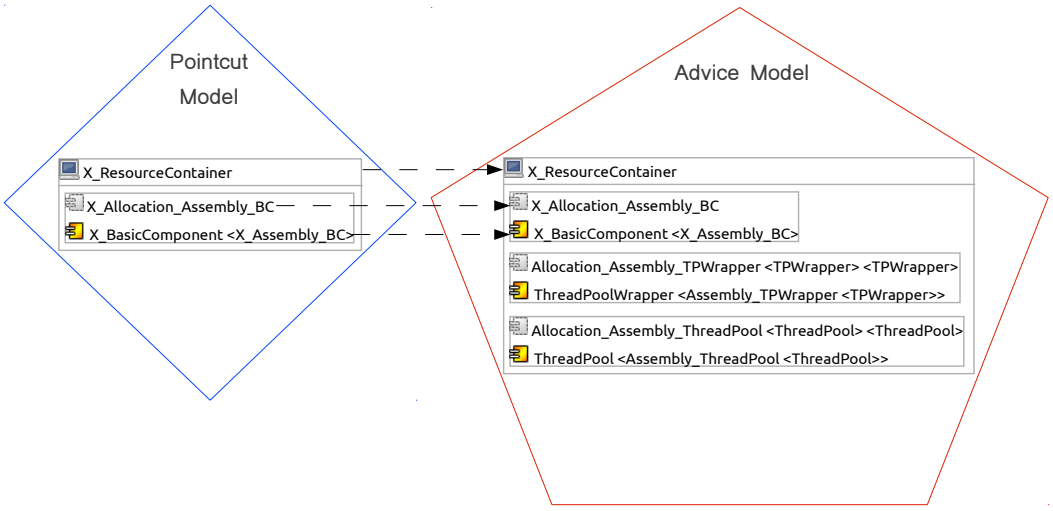


Fig. 5.12.: Allocation aspect fragment corresponding to feature ThreadPool

The last weaving phase consists in weaving the allocation model. For the same reasons as for the system model, only the ThreadPool feature is linked to an allocation aspect fragment. This aspect fragment is defined in Fig. 5.12 and is applied to a base model composed of the woven system model depicted in Fig. 5.7(b) and the original allocation model depicted in Fig. 5.6(d). For readability reasons, the referenced system elements are not depicted in the pointcut and advice elements. In this third phase, the two assembly contexts created in the second phase, Assembly_TPWrapper and Assembly_ThreadPool are allocated to the resource container X_ResourceContainer. This resource container is the resource container that contains the original assembly context X_Assembly_BC. In our example, applying the pointcut model to the base model, the resource container X_ResourceContainer matches AppServer and the original assembly context X_Assembly_BC Assembly_ShopServer. Once this third weaving phase is executed, the repository, system and allocation are modified. The PCM instance depicted in Fig. 5.7 results of these three weaving phases.

5.2.1.4.b. Fixed-Size Unbounded-Waiting-Queue Thread Pool

Section 5.2.1.4.a illustrates steps performed to design a fixed-size thread pool. This section describes steps performed to realize a fixed-size thread pool with an unbounded waiting queue. The base models is described in Fig. 5.6 and the wanted woven model in Fig. 5.8. As the thread pool considered has a static size, steps described in Section 5.2.1.4.a are performed to realize this configuration too. Aspects considered in this section are applied on aspects and thus are similar to repository aspect corresponding to feature Static introduced by Fig. 5.9. They apply on aspects corresponding to a fixed-size thread pool. Fig. 5.10 describes the advice part of this aspect for a repository model and Fig. 5.9 its corresponding pointcut model. Fig. 5.11 and Fig. 5.12 illustrate the system and allocation aspects considered.

First weaving step consists in weaving the repository model. Fig. 5.13 describes the aspect corresponding to an unbounded waiting queue. This aspect adds to the repository model a basic component representing the waiting queue. An interface connects the thread pool

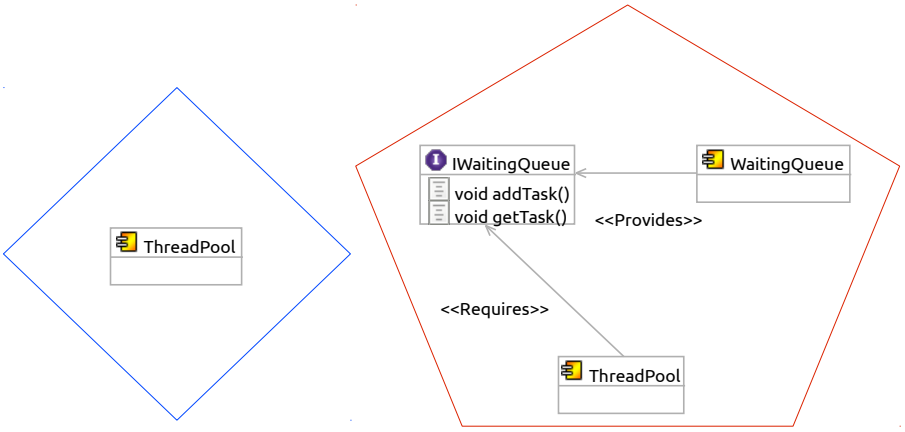


Fig. 5.13.: Repository aspect fragment corresponding to feature Unbounded

component with the waiting queue component introduced. This interface possesses two signatures. If a new task comes in and that no thread in the thread pool can be used or created to execute it, the thread pool calls the first signature operation to store the incoming task in the waiting queue. The second signature operation is called to get back a task stored in the waiting queue and execute it within an available thread. This signature is called when a task finishes its execution and a lets a thread available. The getting task is then executed within this thread. As the waiting queue is unbounded, it is able to store all tasks given by the thread pool component.

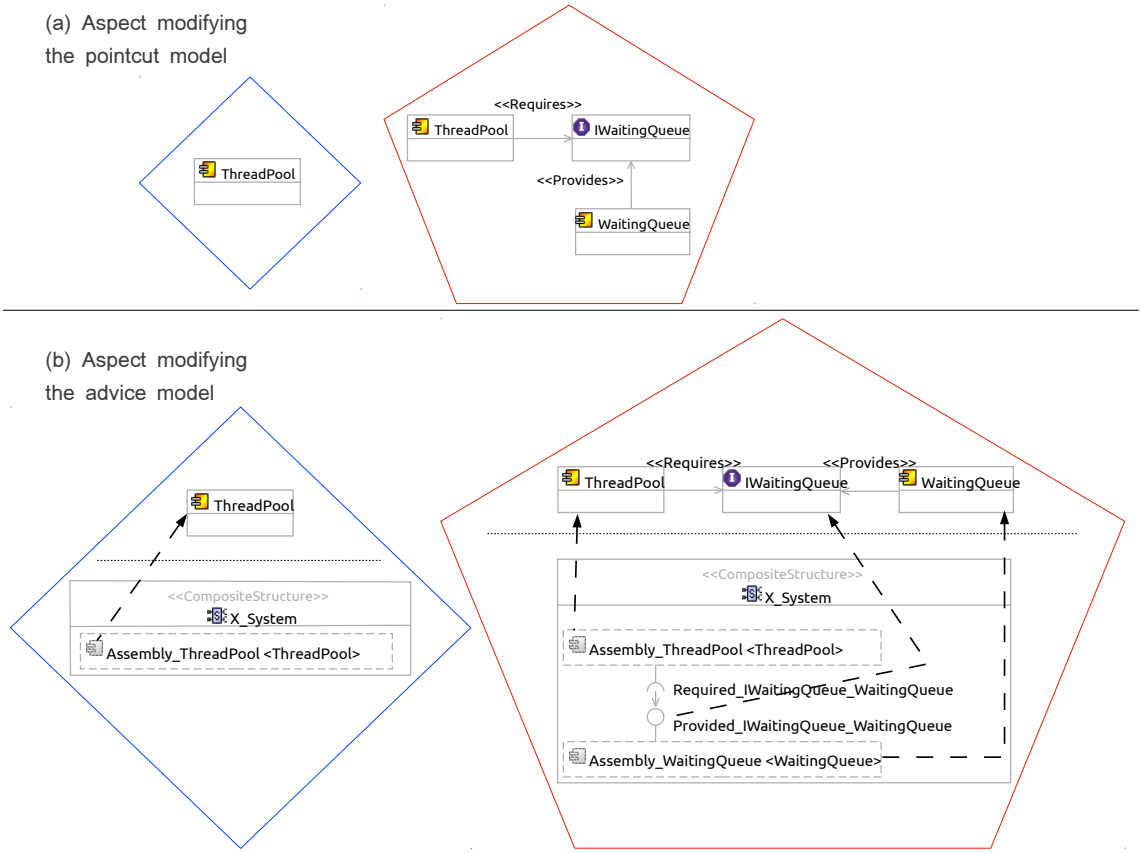


Fig. 5.14.: System aspect fragment corresponding to feature Unbounded

Once the repository model is woven, modifications are performed on the system model. Fig. 5.14 illustrates the aspects used to perform these modifications. These aspects are applied on aspect corresponding to a fixed-size thread pool, depicted in Fig. 5.11. Modifications performed consist in allowing the added waiting queue component to an assembly context. To perform this modification, the pointcut model of the resulting aspect needs to search for the waiting queue component in the pointcut model. This is why the pointcut model of aspect corresponding to a fixed-size thread pool is modified by aspect depicted in Fig. 5.14(a). Aspect depicted in Fig. 5.14(b) is applied on the advice model of aspect corresponding to a fixed-size thread pool. Aspect resulting of the application of aspects depicted in Fig. 5.14 to aspect depicted in Fig. 5.11 is then applied on PCM models. This resulting aspect is applied on the woven repository model depicted in Fig. 5.8(a) and the base system model depicted in Fig. 5.6(b). The repository model is not modified. System model depicted in Fig. 5.8(b) is obtained.

Fig. 5.15.: Allocation aspect fragment corresponding to feature Unbounded

Fig. 5.15 introduces the aspect dealing with the allocation model. These aspects are applied on aspect corresponding to a fixed-size thread pool, depicted in Fig. 5.12. It is similar to the aspect modifying the system model as it also modifies pointcut and advice models of aspect corresponding to a fixed-size thread pool. Modifications performed consist in allowing the assembly context created by the weaving of the system model in an allocation context. The structure of the two aspects is similar to the structure of aspects defined for the system model. Aspect resulting of the application of aspects depicted in Fig. 5.15 to aspect depicted in Fig. 5.12 is then applied on PCM models. This resulting aspect is applied on the woven system model depicted in Fig. 5.8(b) and the base allocation model depicted in Fig. 5.6(c). Allocation model depicted in Fig. 5.8(c) is obtained.

Altogether nine aspects are needed to perform the two configurations of the thread pool completion. Four aspects simulate a fixed-size thread pool. Five more aspects add to a

fixed-size thread pool an unbounded waiting queue.

5.2.2. Connector Completion

The connector completion simulates the network when two assembly contexts are linked with an assembly connector and deployed on two different resource containers. Section 5.2.2.1 describes this completion. It has been realized using Java and GeKo. The two implementations are introduced respectively in Section 5.2.2.2 and Section 5.2.2.3.

5.2.2.1. Description

In context of PCM models, assembly connectors are linked with assembly connectors. All informations exchanged between assembly connectors are propagated via these assembly connectors. In the following we consider two assembly connectors *Comp1* and *Comp2* linked with an assembly connector *Conn1*. This example is depicted in Fig. 5.16(a). If the two assembly connectors are deployed on the same resource container the communication is considered instantaneous. If they are deployed on two different resource containers, the information transmission's speed depends on network characteristics. These network characteristics are low-level details needed to perform accurate performance predictions and thus this refinement of PCM models is a performance model completion. The connector completion is defined by Becker [Bec08].

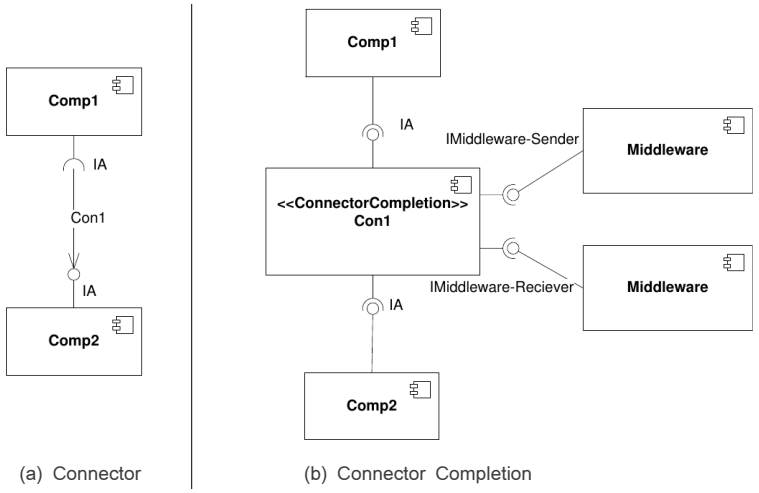


Fig. 5.16.: Replacing a Connector with a ConnectorCompletion, from [Bec08]

Fig. 5.16(b) illustrates the network simulated between two assembly connectors *Comp1* and *Comp2* linked with an assembly connector *Conn1* and deployed on two different resource containers. A connector completion assembly connector is added between the two assembly connectors. The connector completion offers roles complying to those linked by the replaced assembly connector. This connector completion assembly connector is linked with two middleware assembly connectors representing sender and receiver sides. All messages transmitted to the connector completion component are forwarded to middleware components to process all actions needed to process the message. On sender's side middleware components simulate a resource demand. In a second stage a demand on the network is performed to transmit the message. On receiver's side they simulate a message extraction and service call initialization to receive the message. Same actions are then performed in the reverse order to send the computed result to the sender.

In the following this completion is used to refine the media store example furnished in Palladio. This example is built after the iTunes store and allows a user to download

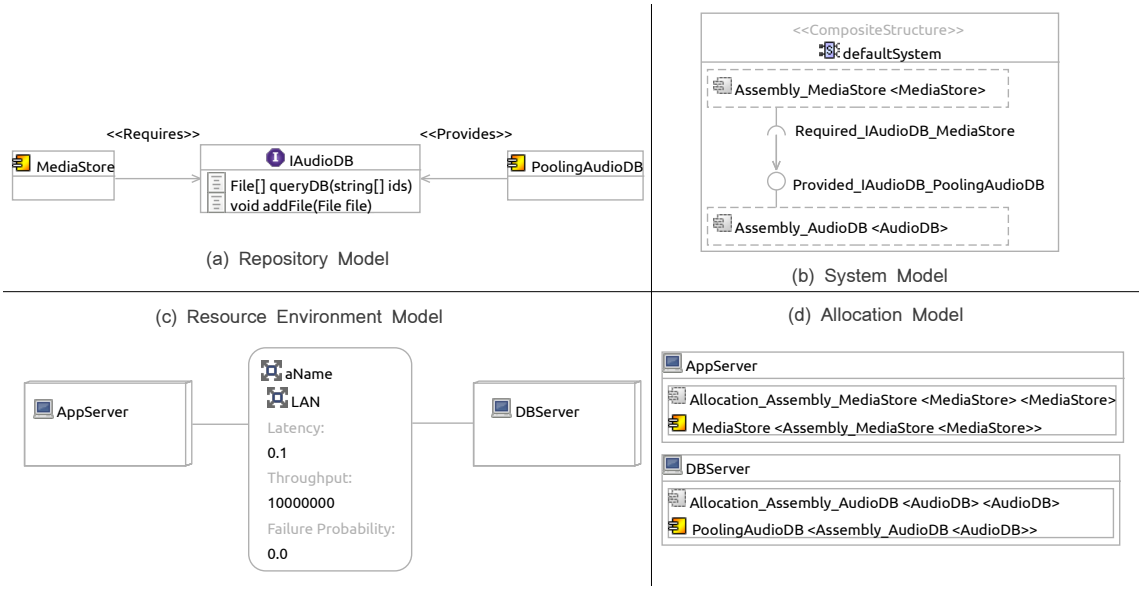


Fig. 5.17.: Part of a PCM instance on which the connector completion is performed

files from a a web-accessible server application. The resource environment model of this example is composed of two resource containers, one corresponding to the application server and the other to the user database server. Fig. 5.17 focuses on elements of the media store example implied in the connector completion. The two assembly connectors depicted in Fig. 5.17(b) are linked with an assembly connector and allocated to allocation contexts deployed in different resource containers. The connector completion is used to introduce a connector completion between these two assembly connectors.

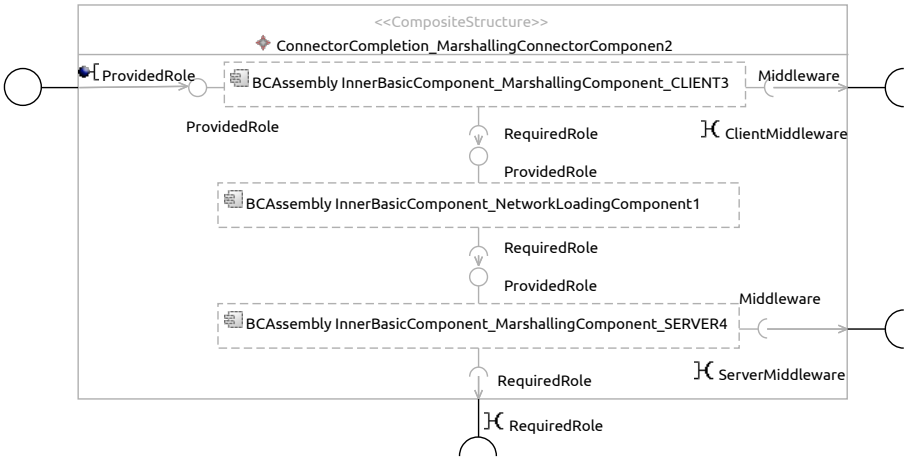


Fig. 5.18.: Structure of the connector completion component

Fig. 5.18 depicts the structure of the completion used in our example. Required and provided roles complying to those linked by the replaced assembly connector are recognizable. In this example we consider that messages are marshalled before being transmitted via the network. For this reason two assembly contexts encapsulate components performing marshalling and demarshalling operations. The marshalling component corresponds to the client side of the example and contains a required role addressing the client middleware. Required role addressing server middleware is contained in the demarshalling component. Marshalling and demarshalling operations are simulated via SEFFs of encapsulated com-

ponents. The third assembly context encapsulates a component that simulates the network load.

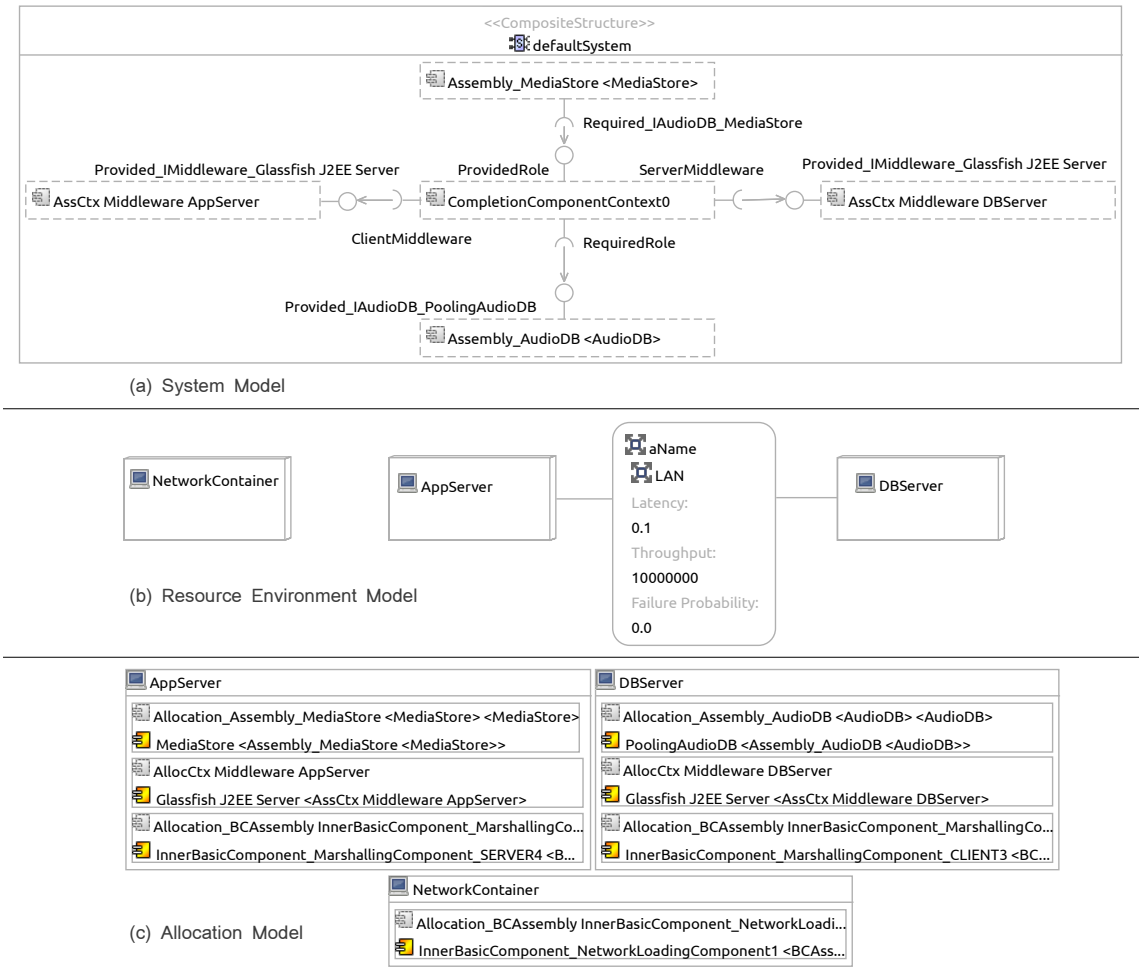


Fig. 5.19.: PCM instance depicted in Fig. 5.17 refined with the connector completion

Fig. 5.19 depicts PCM models obtained after applying the connector completion on the PCM instance introduced by Fig. 5.17. The repository model is not modified due to the connector completion and thus is not depicted by this figure. Three assembly contexts are added in the system model illustrated by Fig. 5.19(a). One encapsulates the completion illustrated by Fig. 5.18. It is linked to the two assembly connectors considered in the base model. Fig. 5.18 shows that the a connector component is composed of three assembly contexts. These three assembly contexts are allocated in three different resource containers. Assembly context corresponding to client side is allocated to the database resource container and assembly context corresponding to server side to the application resource container. Assembly context corresponding to the network load simulation component is allocated to an new resource container. In this example we assume that both application and database server are Glassfish J2EE servers. A Palladio common model is used to simulate these servers. Two assembly contexts, representing application and database servers, encapsulate this Glassfish server. They are linked to the completion assembly context via client and server middleware required roles. They are allocated in the database resource container for the client assembly context and application resource container for the server assembly context.

Following sections compare two implementations of this connector completion. The first,

introduced in Section 5.2.2.2, is a Java implementation integrated in Palladio. The second uses model weaving and GeKo and is described in Section 5.2.2.3.

5.2.2.2. Implementation in Java

The connector completion is fully integrated in Palladio. When running a simulation, user chooses to use this connector completion or not. If selected, the connector completion simulates network between all pairs of assembly contexts linked with an assembly connector and deployed on two different resource containers. This substitution is transparent to the user. Input models are temporarily modified and temporarily contain elements simulating the network. Performance predictions are performed on these temporary models, taking into account network specificities. Once the performance predictions are performed, elements simulating the network are removed. The whole process is transparent to the user.

In this whole process only the creation of temporary models is considered here. To build these temporary models a completion repository is first created. Client and server middlewares components from Palladio default models are introduced in the PCM instance. A third step replaces considered assembly connectors with the completion component and links two considered assembly contexts to the completion. In this thesis we apply the connector completion on the media store example furnished in Palladio. Fig. 5.17 offers a simplified representation of elements of this example and focuses on elements implied in the connector completion.

5.2.2.3. Realized using GeKo

This section describes the realization of the connector completion using model weaving. As this completion is not configurable, simple aspects are used to implement it. Configurable aspects introduced in Chapter 4 are not used here. Aspects introduced in the following add in the PCM instance described by Fig. 5.17 elements to simulate the network joining resource containers on which mediastore and audioDB components are deployed. These aspects modify system, resource environment and allocation models of this instance. The PCM instance illustrated in Fig. 5.19 is produced.

Changes performed in this completion are very similar to those performed for the second configuration of the thread pool completion, described in Section 5.2.1.4.b. For this reason all aspects are not described in detail here. Main difference between the two realizations is that the repository model is not modified in the connector completion. First changes are performed on the system model. Further elements added in the system model result from external models. Element representing client and server middlewares come from Palladio default models. Completion element comes from a pre-built repository model. Fig. 5.20 illustrate the pointcut of the aspect modifying the system model. This pointcut model enhances the fact that elements from three different base models are targeted. The advice model of this aspect is very similar to advice models described previously and thus not detailed here.

Resource environment and allocation aspects are very similar to aspects described previously. The resource environment aspect adds to the existing resource environment an extra resource container to contain elements that simulate the network. The allocation server aspect allocate in the resulting resource environment model, as well two new assembly contexts of the system model, that three assembly contexts contained in the completion component.

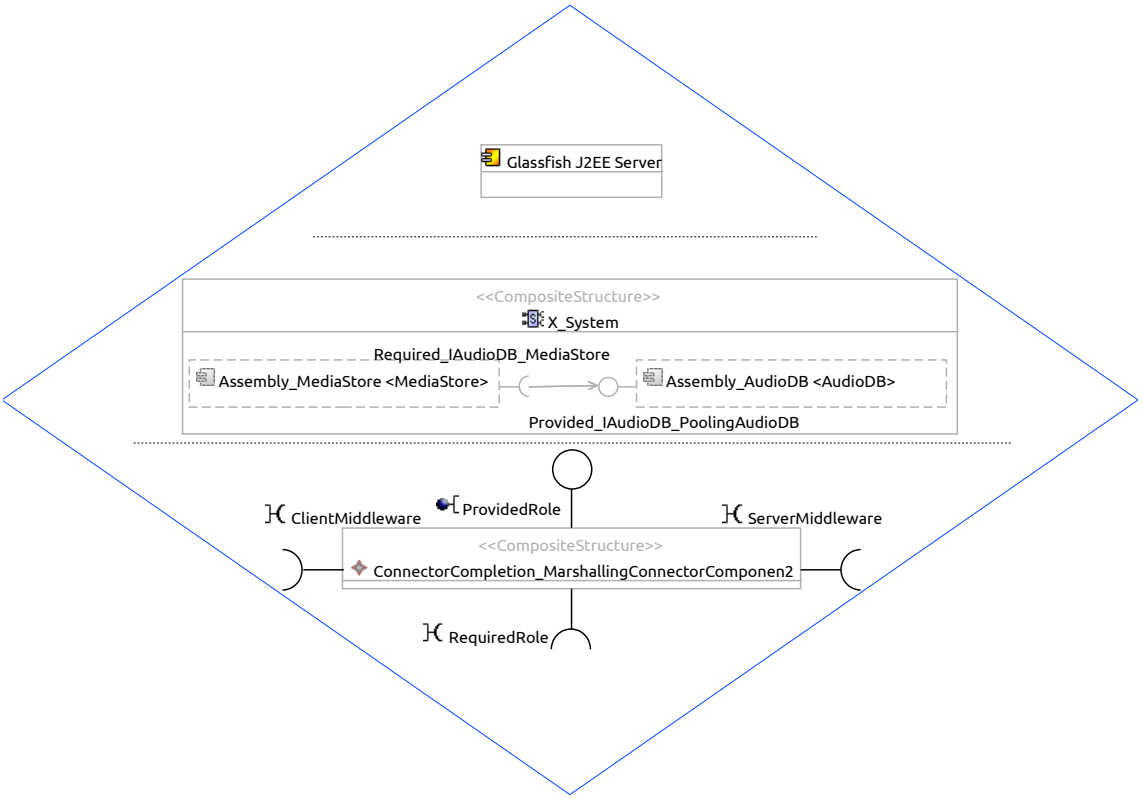


Fig. 5.20.: Pointcut model of the system aspect for the connector completion

5.2.3. Comparison

Section 5.2.1 and Section 5.2.2 describe various realizations of the performance completions considered. These realizations are implemented with four different techniques: QVT-R, ATL, Java and GeKo. Quantitative measurements are first introduced in order to compare these different techniques. Some qualitative remarks complete this comparison.

Four measurements are defined to perform this comparison. First, execution time of various model completions is calculated. To measure the execution time, average over fifty repetitions of a same operation is calculated. This execution time concerns only automatized operations. Comparing two execution times for two different techniques, eventual manual operations have to be considered. For model weaving operations, we notice that the first execution of a model weaving operation takes always more time as following executions. This is probably due to caching but we did not look further to discover exact causes. We do not consider this first execution in measured execution time and perform repetitions about stable executions. Other three metrics are defined originally for model transformation languages. Model weaving uses a graphical representation and thus considered metrics are adapted to this particularity. This adaptation is explained in a second time. Number of lines of code and number of rules are computed for files implied in a model completion. The term rules designates relations in QVT-R, helpers and rules in ATL and methods in Java. Two last measurements considered are average fan-in and fan-out. Fan-in depicts the number of calls to a rule and fan-out the number of calls made by a rule to other rules.

Metrics defined for model transformation languages are adapted to fit model weaving and express similar concepts. Measurements are preformed on all aspects needed to perform the considered configuration. Length of the xmi files is measured to express number of lines

of code. For each aspect, length of pointcut, advice and mapping models are counted. In context of model weaving a rule is considered as a unitary model. An aspect composed of pointcut, advice and mapping models is composed of three rules. Considering an aspect, its fan-out corresponds to the number of aspects it depends on, the number of aspects it needs to fit the considered configuration. For example fan-out of repository model corresponding to the thread pool feature is one for a fixed-size thread pool and two for a fixed-size thread pool with an unbounded waiting queue. Fan-in of an aspect is the average number of calls to this aspect.

Table 5.1 indicates all measurements obtained for different configurations and different realizations of model completions. First remark concerns execution times for GeKo that are significantly higher than execution times for model completions using model transformation languages. Two factors may cause it. First, configuration steps for the chosen model weaving approach is more time consuming than those of the ATL HOT. Using GeKo, configuration steps consist in weaving aspects on aspects, which take longer than the Java HOT considered for the ATL implementation. In regular uses, these configuration steps are performed only once to get the desired aspect. Once the desired aspect is build, models are woven with a lower execution time. Second explanation is that a PCM instance is composed of several models. In the chosen model weaving approach, these model are woven independently. As PCM models reference each other, some models are considered several times. For example, weaving a fixed-size thread pool, the repository model is first woven. Then it is considered a second time at the weaving of the system model. A repository model is not modified at the weaving of a system model but its elements are considered for the join point detection. Thus, two iterations are performed over its elements. These two reasons are ways to explain higher execution times obtained using model weaving.

			QVT-R	ATL	Java	GeKo
ThreadPool	Fixed-Size	Execution Time[1]		0.018		1.110
		Lines of Code	6043	298		245
		Number of Rules	456	4		12
		Fan-In	0,487	1		1
		Fan-Out	0,831	0.33		0.25
	Fixed-Size & Unbounded WQ	Execution Time[1]		0.026		2,241
		Lines of Code		357		268
		Number of Rules		4		27
		Fan-In		1		1
		Fan-Out		0.33		0.667
Connector		Execution Time[1]			0.004	1.241
		Lines of Code			464	279
		Number of Rules			20	9
		Fan-In			1.4	0
		Fan-Out			7.1	0

Table 5.1.: Measurements for three model completion configurations realized using model transformation languages and model weaving

Considering the fixed-size thread pool configuration, number of lines and number of rules of the QVT-R transformation is significantly higher than those of other realizations. Main reason for it is that rules that copy all elements of input models have to be explicitly specified in QVT-R. Using ATL and GeKo, tools provide a possibility to copy all non-modified elements of input models and thus copy rules are not explicitly implemented.

[1]Execution time is measured is seconds.

Table 5.1 shows that, in this example, QVT-R rules are shorter than ATL rules. They are composed of fifteen lines on average whereas an ATL rule is composed of seventy-five lines. This size difference explains higher fan-in and fan-out in QVT-R compared to ATL. In QVT-R, smaller rules are implemented and these rules interact more with each other.

In the second considered configuration of the thread pool completion, all features of the first configuration are used and unbounded waiting queue features are selected too. In GeKo, this addition leads to fifteen new rules, corresponding to five new aspects. These new aspects configure pointcut of aspect applied on repository model and pointcut and advice models of aspects applied on system and allocation models. On the contrary, number of rules in ATL is not modified as the structure of the file is not modified. New statements are added in existing rules. This explains that execution time of the ATL transformation remains almost unchanged compared to the first configuration. Using GeKo, the execution time of model completion is doubled. This increase is due to the addition of five configuration aspects. In this case, 1,437 seconds are necessary to weave all configuration aspects and produce configured aspects. Then, weaving a PCM instance with already configured aspects takes only 0.804 seconds. In this example, 64% of the execution time is dedicated to configuration steps. In regular uses, these configuration steps are performed only once. Thus 0.804 seconds may be considered as the execution time. It is still higher than the considered ATL implementation but significantly less than execution time including aspects configuration.

To understand why this execution time is so high, we weave a PCM instance with a configured aspect and measure execution times corresponding to various weaving phases and type of models composing the PCM instance. To run this experiment we consider the second configuration of the thread pool completion. As all unit execution times are constant, ten weavings are executed to get average execution times corresponding to various weaving phases and type of models. Table 5.2 illustrates the execution time distribution according to weaving phases and model type. This table shows that weaving of the system model takes a little longer than weaving of repository or allocation model. Reason for this is that, compared to weaving a repository model, weaving a system model includes several models of the PCM instance. In this example, more modifications are performed on the system model than on the allocation model. These two differences explain that weaving the system model takes slightly more time than weaving other models of the considered PCM instance. Main characteristic enhanced by the table is that the join point detection represents 76% of the execution time. A first way to reduce execution time of model weaving based completions is to investigate alternative join points detection mechanisms. The second way is to weave a PCM instance and not separated models. In this case, the join point detection is performed only once over all models composing the PCM instance.

	PCM Instance	Repository	System	Allocation
PCM Instance		31%	42%	27%
Preparatory Steps	11%	13%	10%	10%
Join Point Detection	76%	67%	80%	80%
Weaving	13%	20%	10%	10%

Table 5.2.: Weaving a PCM instance from a configured aspect: execution time distribution by model type and weaving phase

Last configuration considered by Table 5.1 handles the connector completion. This model completion does not contain variability. For this reason, number of rules of the model weaving realization is that low. As all designed aspects are applied on input models and no aspect depends on other aspects, fan-in and fan-out are null. The gap between execution times of Java and GeKo realizations is in this case extreme. Length of both realizations

balances it. The Java completion is composed of 464 lines whereas the completion realized with model weaving contains only 279 lines. Besides, model weaving using a graphical representation, modifications performed by a completion are more comprehensible considering its model weaving based realization than its Java realization.

One one hand, measurements performed in this section highlight the fact that model completions realized using GeKo are similar to model completions realized using ATL HOTs in terms of length and complexity. To implement same model completions using QVT-R HOTs, rules to copy all elements of input models need to be created. These copy rules account for length of QVT-R realizations of model completions. However, compared to the Java implementation of the connector completion, its model weaving realization is shorter. One the other hand, this comparison enhances that model completions realized using GeKo are significantly slower. This result is nuanced by the observation that configuration steps form a big part of this execution time and that the join point detection phase is the most time-consuming weaving phase. A first solution to reduce this execution time is to compute once aspects corresponding to used configurations and then weave input models directly with a configured aspect. Second solution is to reduce the join point detection execution time weaving all aspects composing a PCM instance at one time or investigating alternative join points detection mechanisms.

5.3. Practical Limitations

After comparing model weaving and model transformation on a conceptual level and with two case studies, the following section enhances practical limitations due to tool support. The case studies have been implemented using three model transformation languages, described in Section 2.1.2. Section 5.3.1 describes tools used to perform the QVT-R and ATL transformations and highlights their respective maturity. Section 5.3.2 shows how variability is supported in the aspects in practice. This part also makes a distinction between operations supported by a tool and operations performed manually. As a part of the thesis is to make GeKo compatible with the weaving of PCM models, we do not discuss technical limitations of the implementation for GeKo here. Adjustments performed on GeKo are described in Chapter 6.

5.3.1. Model Transformation Tools

The first performance model completions we consider are realized using QVT-R. They are taken from the Chilies approach described in Section 2.3.2 and have been implemented in 2010. Medini QVT [IKV13] has been used to run theses transformations. This tool implements the OMG's QVT-R standard and is integrated into Eclipse. It provides an editor with syntax highlighting editor and code completion. A graphical debugger allows step to step debugging. It also supports traces to record model elements involved in the transformation. This tool is not mature yet and deals with bug fixes. It is not updated regularly and thus is incompatible with new versions of Eclipse and PCM. For these reasons we are unable to run the transformations implemented in QVT-R for the thread-pool completion case study, introduced in Section 5.2.1. Thus the QVT-R transformations are statically analyzed.

To realize ATL transformations, we use the tools of the Model-to-Model Transformation project of Eclipse [Ecl13b]. Using the ATL Integrated Environement, the developer benefits from standard development tools like syntax highlighting and a debugger. The ATL engine also manages the input and output models and meta-models. This eases the development of ATL transformations. Useful to implement first transformations, the ATL Transformation Zoo [Ecl13a] provides a lot of examples. It contains basic examples, like Families to Persons, but also more complex one, like the Simple Class to Simple RDBMS transformation. Compared to Medini QVT that support QVT-R, tools used to realize ATL transformations provide a better support. The many examples offered in the ATL Transformation Zoo constitute another advantage of ATL over QVT-R.

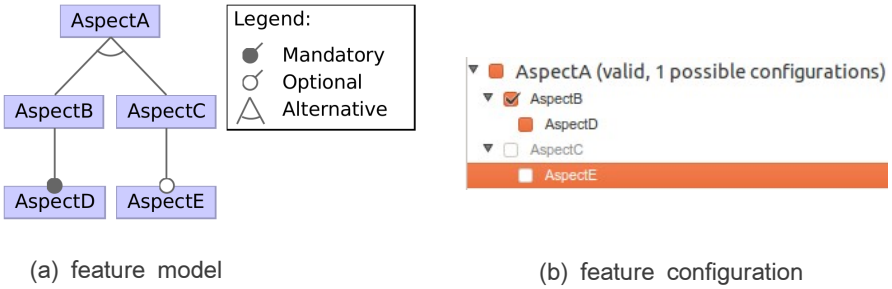
Case studies introduced in Section 5.2 contain two model transformations implemented in Java. The first is the connector completion, initially realized using Java, and the second is the configuration part of the ATL HOT. To run Java transformations, only a classical Java environment is needed.

5.3.2. Variability Support in Aspects

To perform the perform model completions with model weaving, we used the configurable model aspects described in Chapter 4. We use GeKo, described in Section 2.2.1, to support the weaving of two models. However, other tools are needed to support variability introduced in the aspects.

FeatureIDE [TKB+12] provides support for the feature model design. This tool provides support for the design of feature diagrams, using a graphical syntax, and the specification of cross-tree constraints. Fig. 5.21 uses the basic example composed of five aspects, introduced in Section 4.3. Fig. 5.21(a) depicts the graphical representation used in FeatureIDE

to design a feature model. FeatureIDE supports the derivation of valid feature configurations selecting mandatory features and not allowing the selection of excluding features. A valid feature configuration is depicted in Fig. 5.21(b). For valid configurations, the output is a list of all selected features. The features are ordered from more general to more specific aspects.



(a) feature model (b) feature configuration

Fig. 5.21.: FeatureIDE support for feature model design (a) and feature configuration derivation (b)

Each feature of the feature model is linked with an aspect fragment. A xml file is introduced to specify paths of pointcut, advice and mapping models composing the aspect fragment corresponding to each feature. In this xml file, potential attributes may also be defined. These attributes allow a parametrization of aspect fragments. For example, in the thread pool completion, in the static size aspect, the length of the thread pool is a parameter. This way, if the feature static size is chosen, the length of the thread pool can be specified using this parameter.

To summarize, configurable aspect models introduced in Chapter 4, are realized using feature models and configurations produced by FeatureIDE. They support the variability with SPL concepts. A XML file is used to configure aspect fragments localization and to support the parametrization of these aspect fragments. Finally, GeKo is used to design aspect fragments and to weave selected aspect fragments in order to produce the configured aspect. The weaving of these produced configured aspects and the base model is also performed using GeKo.

5.4. Conclusion

This chapter offers a comparison between a model weaving approach and model transformation languages in the context of performance model completions realized for PCM models. This comparison is based on conceptual aspects as well as two case studies and technical limitations. Main equivalence between model transformation languages and model weaving is that they are able to perform same model transformations although use different logics to achieve it. As we consider a model weaving approach including configuration-based variability, configuration mechanisms of various techniques considered are equivalent.

Model transformation languages and model weaving differ in many points. In term of length, model completions realized using model weaving are shorter than their realizations using QVT-R or Java. The difference with QVT-R transformations is extreme and is due to the necessity of specifying explicitly copy rules in QVT-R. In this comparison QVT-R also suffers from tools lacking maturity. But model weaving and model transformation languages differentiate mainly from each other by their representations. Model transformation languages use a textual representation whereas model weaving uses a graphical representation. For considered model completions, major drawback of model completions realized using model weaving is their execution time that is significantly higher.

To summarize, considering model completions in the context of PCM models, main advantage of model weaving compared to model transformation languages is its graphical representation. However, on considered examples, execution time of model weaving based model completions is too important and thus they are not equivalent to realizations performed using model transformation languages. To reduce this execution time, aspects corresponding to used configurations could be computed once so that input models are then woven directly with a configured aspect. As the join point detection phase is the most time-consuming weaving phase, a second solution consists in reducing the join point detection execution time weaving all aspects composing a PCM instance at one time or investigating alternative join points detection mechanisms.

6. Integrating GeKo in Palladio

GeKo is a generic model weaver and operates on the meta-model level. Thus it can easily be adapted to any meta-models. However, the latest implementation of GeKo, provided by Kramer et al. [KKS+12b], has not be applied to many meta-models yet. It has been tested with LTS and Basic Sequence Diagram (BSD) models and applied to Industry Foundation Classes (IFC). On one side, the LTS and BSD meta-models are small models. The LTS meta-model contains three elements: LTS, States and Transitions. BSD models are a simplified version of UML sequence diagrams. They contain eight elements like lifelines, events and messages and allow to model a communication between two entities. BSD models do not possess constructs like conditions, loops etc. On the other side, the IFC meta-model is a more significant meta-model composed of 6131 lines. It is a format defining objects for building modeling. This meta-model is derived from ASCII models and is automatically created by an application. Therefore its structure is very consistent and not many different structural patterns occur.

In this thesis, we applied GeKo on PCM. PCM and IFC meta-models are two significant meta-models but have not been designed in the same way. Whereas the IFC meta-model is derived from a tool, the PCM meta-model is manually extended over the years. Moreover, in PCM, an instance can be seen from different views and is composed of several models. IFC models do not have this point of view notion. As a result, the structures of the two models differ from each other. This is why applying GeKo on PCM models allowed us to discover some unexpected behaviors in GeKo.

As no extension point was needed to adapt the weaving for PCM, this application insures that GeKo is a generic model weaver. However, this second application let us highlight some weaknesses of the current implementation. Most weaknesses are requirements that GeKo implicitly assesses and that PCM models do not fulfill. Section 6.1 introduces functionnalities needed to perform model weaving that GeKo did not furnish. Section 6.2 describes the modifications performed at the meta-model code generation phase. In Section 6.3, the improvement performed on weaving operations is explained.

6.1. Added Functionalities

In most cases, the functionalities added to GeKo are combinations of existing operations. These new functionalities use the operations furnished by GeKo in a new way to correspond to a new use case. We are convinced that these use cases appear frequently in other weaving applications. Thus, the added functionalities may also be used in further applications.

6.1.1. Pointcut and Advice Editors Generation

The first thing to do when applying GeKo to a new meta-model is to derive pointcut and advice meta-models from the application meta-model. GeKo offers a command generating base, pointcut and advice editors that can be called for any meta-model. Executing this command, pointcut and advice meta-models are derived from the selected meta-model. Then model, edit and editor code is generated for all three meta-models.

In the case of PCM, the model, edit and editor code for the application meta-model is already implemented. We want to use this furnished code and not the new code generated by GeKo. That is why we add a new functionality to GeKo that generates code only for the pointcut and advice meta-models. This added interface is only a combination of existing operations but makes GeKo conforming to an accurate behavior. Working with real-life applications, an editor conforming to the application meta-model is already furnished. Using GeKo, new pointcut and advice meta-models and editors are created. However, it is important to keep the furnished application editor.

6.1.2. Weaving a Base Model Referencing other Base Models

As described in Section 2.3.1, four types of developer roles are defined in Palladio. Each developer role produces specific models, corresponding to specific views on the application to model. For example, a component developer produces a repository model where the components specification and implementation is determined. Then, a software architect designs the architecture of the system in an architecture model. At this point, a system is composed of assembly contexts referencing components designed in the repository model. As a result, a PCM instance is composed of several models referencing each other.

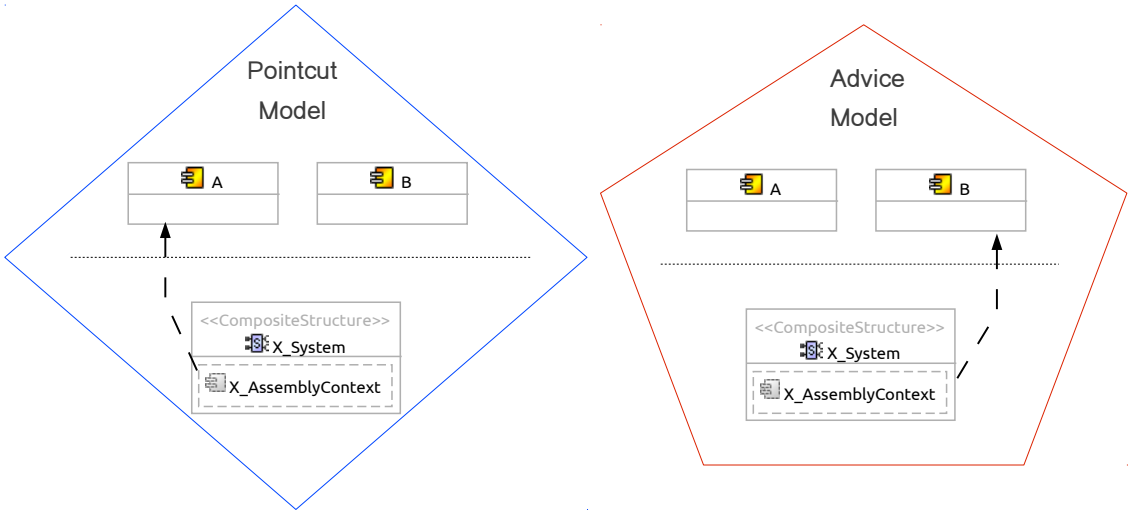


Fig. 6.1.: Pointcut and advice models targeting more than one base model in PCM

As the various models composing a PCM instance reference each other, more than one base model is needed to perform weaving operations. Fig. 6.1 depicts an example where the pointcut and advice models target two base models. In this example, the component referenced by an assembly context has to be updated. In the domain of performance model completions, component A would be a generic component and B the same component enhanced with performance annotations. The repository model stays unchanged, only the system model is modified. However, the repository model is needed to perform the changes and is targeted by pointcut and advice models.

In the previous version of GeKo, a weaving operation takes as input only one base model. Considering example depicted by Fig. 6.1, base model is composed of the system model only. As the base model does not contain repository elements, no join point is detected. To solve this, we give the possibility to the user to specify several base models. However, only one base model is modified. It is identified adding a *_ToWeave* extension to its name. Other models are referenced and not modified during the weaving. Considering example depicted by Fig. 6.1, base models are repository and system models. The system model is identified as model to weave. The repository model is used as referenced model and not modified. During the join point detection, a list containing elements of repository and system models is run through. All assembly contexts referencing an A element are identified as join points if the repository containing A also contains a B element. For each detected join point, the system model is then updated so that the assembly context does not reference the component A any more but targets element B.

6.1.3. Perform In-Place Transformations

As mentioned in Section 2.1.1, in-place transformations use the same model as input and output, whereas out-place transformations produce a new output model. PCM models composing a PCM instance reference each other. For example a system model targets elements of a repository model. Modifying a repository model with an out-place transformation, the created repository model replaces the existing repository model in the PCM instance. To keep the PCM instance coherent, references made by the system model on elements of the repository model have to be updated to target elements of the new repository model. Modify the repository model using in-place transformations, same repository model is contained in the PCM instance. It avoids to change references in the system model. As PCM models reference each other, weave various models of a PCM instance using in-place transformations is important to keep the global consistency of the PCM instance.

6.2. Pointcut and Advice Meta-Models Derivation

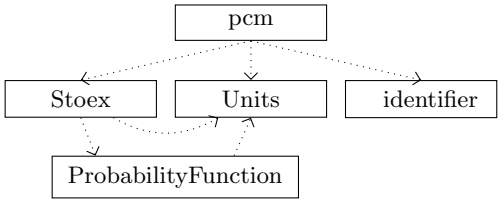
Before performing weaving operations, pointcut and advice meta-models and editors are derived from the application model. In contrast to the meta-models to which GeKo has already been applied, the PCM meta-model has two characteristics: it depends on other meta-models and is split into several packages. Section 6.2.1 and Section 6.2.2 describe how GeKo was adapted to consider these characteristics.

6.2.1. Meta-Models Depend on other Meta-Models

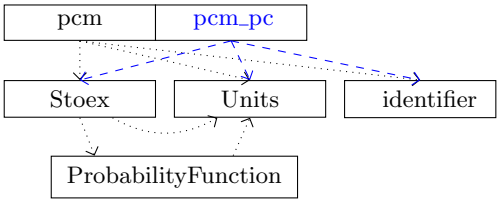
Fig. 6.2(a) depicts the meta-models the PCM meta-model depends on. It directly depends on three meta-models, Stoex, Units and identifier. The meta-model identifier handles uniquely identifiable elements. The Stoex meta-model handles stochastic expressions and depends on ProbabilityFunction, that handle probability expressions. These two meta-models depend on Units.

When deriving pointcut and advice meta-models from the PCM meta-model, the initial version of GeKo kept the references to the original meta-models. Fig. 6.2(b) illustrates the obtained pointcut meta-model, referencing original meta-models. Fig. 6.2(c) shows the wanted result: a pointcut is derived for all meta-models. For space reasons we chose to focus on dependencies of pointcut meta-models and not to represent dependencies between the original meta-models.

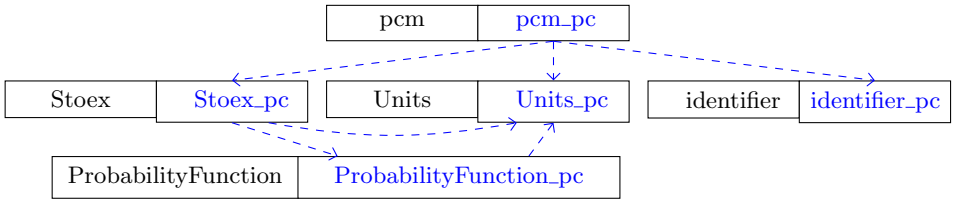
As the derivation of the pointcut and advice meta-models need to be performed only once per new meta-model considered, we chose to manually correct the dependencies between



(a) pcm meta-model dependencies



(b) pcm_pc depends on original meta-models



(c) pcm_pc depends on pointcut meta-models

Fig. 6.2.: PCM meta-model dependencies and their influence for the pointcut meta-model derivation

pointcut and advice meta-models. To build correct pointcut and advice meta-models, following manual operations are needed:

1. With GeKo, generate pointcut and advice meta-models for each model. In the case of the PCM meta-model, ten models are generated: five original meta-models times two derived meta-models, a pointcut and an advice, for each meta-model.
2. When deriving pointcut and advice meta-models, GeKo also produces generator models. As the meta-models referenced has not been corrected yet, the produced generator models for ProbabilityFunction, Stoex and PCM have to be deleted.
3. Recreate manually these six generator models. Do it for ProbabilityFunction first, as the generator model for the meta-model it depends on has already been produced, Stoex then and finally for PCM. During the generation, select only the current package as package to generate and reference the already produced generator models for the other packages.
4. As we generated these generator models manually, some properties have to be manually updated: the model, edit, editor and test directories and plug-in ID. To distinguish them from the original code, the extension _pc or _av is added to the meta-model name, in the directories and plug-in IDs.
5. For all generator models, ten in our case, the *basePackage* property have to be changed: the _pc or _av extension is added to each part of the original base package. This way, for the pointcut meta-model of PCM, *de.uka.ipd.sdq* becomes *de_pc.uka_pc.ipd_pc.sdq_pc*.
6. Once all these modifications are performed, the model, edit and editor code is generated from the generator models. The dependency order must be respected. In our case, generate the code first for the identifiers and Units meta-models, then for ProbabilityFunction, Stoex and finally PCM meta-models.

6.2.2. Meta-Models Organized in Multiple Packages

Another particularity of the PCM meta-model, compared to IFC meta-model, is its division in several packages, illustrated in Fig. 6.3. The meta-model pcm.ecore contains a

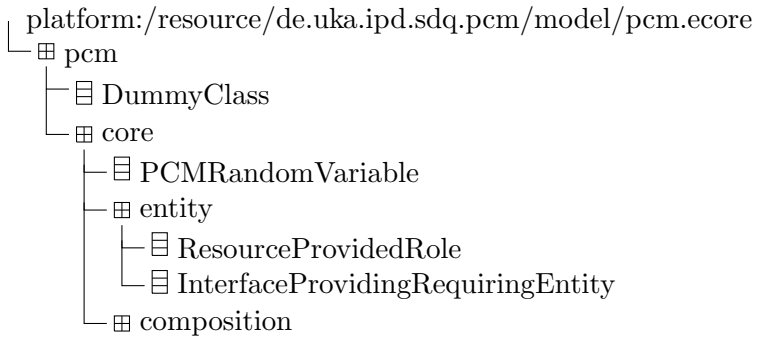


Fig. 6.3.: A part of the PCM meta-model, illustrating its division in packages

root package pcm, divided in core, usagemodel, repository etc. packages. For example, the core package is then divided in entity and composition packages. During the derivation of pointcut and advice meta-models, the name, NsPrefix and NsURI of packages are modified. So far, GeKo considered that meta-models contain only a root package. Thus, modifications were performed only for this root package. The correct behavior is, for each package, to get all packages it contains and also update their name, NsPrefix and NsURI. We rectify the way GeKo derives pointcut and advice meta-models, so that this division in several packages is considered.

The join point detection was also erroneous because of the division in packages. The full class names is composed of names of all packages the class is contained in. According to Fig. 6.3, the full class name of class *ResourceProvidedRole* is *pcm.core.entity.ResourceProvidedRole*. The name of the pointcut variant of this class is then: *pcm_pc.core_pc.entity_pc.-ResourceProvidedRole*. During the join point detection, the pointcut version of a class is compared to its base version, to detect if the two compared classes are equivalent. As GeKo considered that a class is contained only in the root package, the created base version of the pointcut class *ResourceProvidedRole* was *pcm_pc.core_pc.entity.ResourceProvidedRole*. Only the last occurrence of the extension *_pc* was deleted. But this name does not correspond to any existing class. As the element is searched in wrong package, no corresponding element is found and an error is thrown. We rectify the conversion of pointcut objects into their base equivalent, so that all occurrences of the extension *_pc* are deleted from the full class name, during this conversion.

6.3. Weaving Operations

In three cases the behavior implemented in GeKo did not match the expected behavior. Section 6.3.1 highlights that GeKo implicitly requires that all elements are contained in the root package. However, PCM models contain sub-packages and GeKo is rectified to handle them. GeKo was not tested with enumeration attributes. Section 6.3.2 describes why these attributes have to be handled differently from classic attributes. Section 6.3.3 shows the changes performed on GeKo when duplicating an element.

6.3.1. Containment Hierarchy

During the join point detection, drools rules are created. These rules declare all elements contained in the pointcut model together with their properties. The root element is first declared, and then an iteration is performed over all elements it contains to declare them. In LTS models, and all models GeKo was applied up to now, only one containment level is considered. In LTS models, the root element is a LTS element, that contains states. A transition is contained is the state that triggers it. A transition does not contain any

element. Therefore, up to now, the root element was declared and then all elements it contains.

Considering PCM models that possess more containment levels, contained elements are declared several times. This multiple declaration of variable is due to the fact that, when declaring an element, an iteration is performed on all its descendants to declare them. To solve this problem, when an element is declared, the iteration is now performed over its children only.

6.3.2. Enumeration Type

Fig. 2.2 depicts the central concepts of Ecore. All models handled in this thesis conform to a meta-model that conforms to the Ecore meta-modeling language. Thus, their classes contain references and attributes. A reference targets another class of the model and attributes are simple-typed attributes like integer or string. An enumeration is also considered as an attribute. However, the enumeration attributes differ from other attributes as they reference the enumeration literal they take as value. Moreover, all attributes, aside from enumeration attributes, stay unchanged in the pointcut and advice versions of the base model. A string always stays a string. On the contrary, as enumeration classes are part of the designed model, pointcut and advice versions of the enumeration attributes are derived. For this reason, enumeration attributes need to be handled a bit differently from other attributes. Next sections explain the errors encountered during the join point detection first, and during the weaving phase then. The third section mentions the possibility to create a neutral enum. To illustrate the problems encountered, we consider a repository containing a basic component. This basic component has an attribute ComponentType of type enumeration.

6.3.2.1. Join Point Detection

During the join point detection, drools rules are created. These rules were not specified correctly for the enum type. Considering the attribute ComponentType of a basic component, following rule is generated: $s1Decl: de.uka.ipd.sdq.pcm.repository.BasicComponent(componentType == ”BUSINESS_COMPONENT”). This rule contains the string representing the attribute but not the attribute itself. To represent the enumeration attribute, the full-qualified name of the enumeration has to be contained in the rule. The correct rule to create is: $s0Decl: de.uka.ipd.sdq.pcm.repository.BasicComponent(componentType == de.uka.ipd.sdq.pcm.repository.ComponentType.BUSINESS_COMPONENT). The drools rule creation phase in GeKo is rectified so that a correct rule is specified for enumeration attributes.

6.3.2.2. Weaving Phase

As said in Section 6.3.2, unlike classic attributes, pointcut and advice versions of enumeration attributes are derived. During the weaving, advice elements are derived in base elements. If a string attribute is modified, like the name of the repository, the value of the advice attribute is directly copied in the base attribute. Indeed, both elements are from type string. However, considering an enumeration attribute, a base version of the advice attribute has to be created. For this reason, during the weaving, enumeration attributes have to be handled differently from classic attributes. They are handled similarly to references, because the objects targeted by the references are also derived in pointcut and advice versions.

6.3.2.3. Neutral Enumeration Literal

For all classic attributes, a null version of this attribute can be specified in pointcut and advice elements. Indeed, all classic types possess a neutral element. This way, in the pointcut model, if the name of a basic component is not specified, the name of the basic components is not a selection criterion during the join point detection. However, for an enumeration attribute, this neutral element does not exist. For example, the default component type is a business component. If we do not modify this value, only the basic components of type business component are selected. If we change this value, the basic component type can only have one of the types described in class de.uka.ipd.sdq.pcm.repository.ComponentType. A basic component may be a business component or an infrastructure component. Thus, it is impossible to specify a neutral attribute, as the enumeration type ComponentType does not contain a neutral enumeration literal. A solution to this would be to create an empty enumeration literal for all enumeration types, during the creation of the relaxed pointcut and advice meta-models. We did not implement this solution.

6.3.3. Duplication of an Element

As described in Section 5.1.2, when a pointcut element is linked with two advice elements, this element is duplicated. Fig. 6.4 represents a LTS example where the state *b* is duplicated. Fig. 6.4(a) depicts the base model and Fig. 6.4(b) and (c) the aspect applied on this base element. The mapping between the pointcut and advice models is symbolized with arrows and shows that state *b* is linked with states *b1* and *b2*. Thus, the state *b* is duplicated.

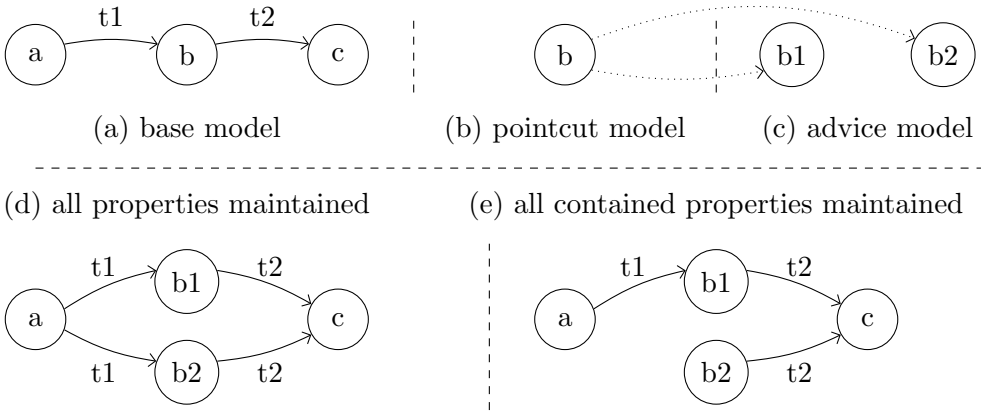


Fig. 6.4.: LTS example illustrating the two kinds of duplicating an element

Fig. 6.4(d) and (e) show two possible woven models, representing two interpretations of an element duplication. Up to now, in GeKo, the woven model depicted in Fig. 6.4(d) is performed. New states *b1* and *b2* exhibit all properties of state *b*: each state has as incoming transition a *t1* transition, and as outgoing transition a *t2* transition. To achieve it, all attributes and references of *b* are copied. But the objects referencing *b* are also considered. In LTS, the transitions are contained by the state that trigger them. In our example, *t1* is contained in *a* and *t2* in *b*. In order that duplicated states are referenced by the same objects than state *b*, the transition *t1*, contained in *a* is also duplicated. In this scenario not only the duplicated element, but also the elements referencing the duplicated element are modified.

Fig. 6.4(e) depicts a second variation of a woven model. In this duplication scenario, only the duplicated element is modified. The duplicated elements do not exhibit all properties of state *b* but they exhibit all properties contained in *b*. Attributes of duplicated states

conform to attributes of the state *b*. All references contained in *b* are also contained in duplicated elements. In our example, only the *t1* transition, contained in *a* is missing in this scenario. It can easily be added by specifying it in pointcut and advice models. We modify GeKo implementation so that when an element is duplicated, only elements it contains are duplicated. Elements referencing the element to duplicate are not duplicated. Indeed, in real-world models like PCM models, an element is referenced by many other elements. Duplicate all these elements can lead to unwanted side-effects. For this reason, we modified the duplication of elements, so that it modifies only the element to duplicate.

This chapter presents practical comments about the integration of GeKo in Palladio. Most modifications performed are implied by the fact that a PCM instance is stored in several models and by specificities of the PCM meta-model. Indeed, as the PCM meta-model is manually extended over the years, its structure differ from those of meta-models derived from tools. Some adjustments are performed to handle situations not considered in the previous implementation of GeKo, like enumeration attributes or in-place transformations.

7. Conclusions & Future Work

The goal of this thesis was to realize configurable model completions for PCM models using model weaving and to compare them with model completions realized using model transformation languages. To achieve this goal, three aspects were addressed. As model transformation languages possess variability techniques whereas model weaving approaches lack configurability, configuration-based variability was first introduced in the chosen model weaving approach, GeKo. It was chosen as model weaver because it was a generic and extensible model weaver. Furthermore, it operates only on the meta-model level and thus enables the transformation of models that are instances of various meta-models. To enable configuration-based variability in GeKo, an aspect corresponding to a configuration is built from the set of aspect fragments linked to all features selected to form this configuration. The configuration is managed with feature models. An aspect corresponding to a configuration is built weaving pairwise aspect fragments included in this configuration. Aspects fragments modify pointcut or advice models of another aspect or aspect fragment to specialize the behavior of the latter.

In the central part of the thesis, model weaving and model transformation languages were compared in the context of performance model completions realized with PCM instances. Whereas model transformation language use a textual representation and the abstract syntax of a model, the generic model weaver can reuse the concrete syntax and graphical representation of models. We are convinced that this will be easier for a user not familiar with the abstract syntax of a model to use model weaving to model a completion. Comparing the verbosity of both techniques, we noted that both techniques are similar although they use different logics to perform a same operation. This comparison was pursued with two case studies already implemented for PCM models using model transformation languages, the thread pool and connector completions. Equivalent completions that produce identical output models for identical inputs and provide comparable configuration mechanisms were realized using the model weaver. Resulting implementations were compared in terms of length, execution time and complexity. In terms of length and complexity model completions realized using considered model transformation languages and model weaving are comparable. However execution times of model completions realized with model weaving are significantly higher. Configuration steps and join point detection in particular are time-consuming. Finally limitations of each approach due to tool support concluded this comparison.

In the third part of the thesis we describe the practical adjustments performed on GeKo to weave PCM models. As the chosen implementation of GeKo has not be applied to

many meta-models yet, it implicitly assesses requirements that the PCM does not fulfill. Specificities of PCM models that GeKo does not consider were discussed in this part. To realize wanted model completions, functionalities were added to GeKo, like weaving models referencing each other. Finally, this section highlighted cases where GeKo did not match the expected behavior and considered reasons of this unexpected behavior as well as solutions.

Future work could focus on two concerns addressed in this thesis. Considering introduction of configuration based variability in aspects, the described approach could be automatized in the future. From a set of aspect fragments linked to a feature model, aspects corresponding to selected configuration is derived automatically from the set of features forming the configuration. The tool could also support design of aspect fragments detecting possible conflicts between aspects and offer automatic solutions to solve these conflicts. Solve incompatibilities between aspects implies also to specify a weaving order between aspects.

Other directions of future work address in particular model refinements in the context of PCM models. First performance of model completions realized using model weaving could be improved upgrading the considered model weaving approach. To accelerate the join point detection, alternative join points detection mechanisms could be investigated. Furthermore, all aspects composing a PCM instance could be woven at one time. Finally ways to perform model refinements using model weaving could be integrated in the Palladio process. To fit a large number of model completions and configurations, a library of aspects fragments should first be realized. Integrate model weaving based model completions in the Palladio process allows to a user to mark in its PCM instance elements to refine and to link them with a furnished configuration. As the configuration is composed of aspect fragments designed with the concrete syntax of PCM models, the user could visualize the influence of all configuration possibilities. To exactly fit the desired behavior, a user could also design new aspect fragments, using concepts he is familiar with.

We hope that these avenues for future research help to improve completion tasks in Palladio or other modelling approaches.

Bibliography

[AABS+13] W. Al Abed, V. Bonnet, M. Schöttle, E. Yildirim, O. Alam, and J. Kienzle, "Touchram: A multitouch-enabled tool for aspect-oriented software design," in *Software Language Engineering*. Springer, 2013, pp. 275–285.

[BC04] E. Baniassad and S. Clarke, "Theme: An approach for aspect-oriented analysis and design," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 158–167.

[Bec08] S. Becker, *Coupled model transformations for QoS enabled component-based software design*. Univ.-Verlag Karlsruhe, 2008.

[Béz05] J. Bézivin, "On the Unification Power of Models," *Software & Systems Modeling*, vol. 4, pp. 171–188, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10270-005-0079-0

[Bie10] M. Biehl, "Literature Study on Model Transformations," Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK/R-10/07-SE, Jul. 2010.

[BKB+08] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing multi-view aspect models," in *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*. IEEE, 2008, pp. 43–52.

[BKR07] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Proceedings of the 6th international workshop on Software and performance*. ACM, 2007, pp. 54–65.

[BKR09] ——, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[BWWB12] G. Besova, S. Walther, H. Wehrheim, and S. Becker, "Weaving-based configuration and modular transformation of multi-layer systems," in *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 776–792.

[Cha13] Chair for Software Design and Quality of the Karlsruhe Institute of Technology. (2013, May) Chilies. [Online]. Available: http://sdqweb.ipd.kit.edu/wiki/Chilies

[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.

[DFBV06] M. Del Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the eclipse amw plugin," in *Eclipse Modeling Symposium, Eclipse Summit Europe*, vol. 2006. Citeseer, 2006.

[Ecl13a] Eclipse Foundation. (2013, May) ATL Transformations. [Online]. Available: http://www.eclipse.org/atl/atlTransformations/

[Ecl13b] ——. (2013, May) Model-to-Model Transformation (MMT). [Online]. Available: http://projects.eclipse.org/projects/modeling.mmt

[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.

[GMS06] V. Grassi, R. Mirandola, and A. Sabetta, "A model transformation approach for the early performance and reliability analysis of component-based systems," *Component-Based Software Engineering*, pp. 270–284, 2006.

[GV07] I. Groher and M. Voelter, "Xweave: models and aspects in concert," *Aspect-oriented software development*, vol. 209, pp. 35–40, 2007.

[GW08] T. Goldschmidt and G. Wachsmuth, "Refinement transformation support for qvt relational transformations," in *3rd Workshop on Model Driven Software Engineering, MDSE*, 2008.

[HBR+10] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric Performance Completions for Model-Driven Performance Prediction," *Performance Evaluation*, vol. 67, no. 8, pp. 694–716, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2009.07.006

[HFBR08] J. Happe, H. Friedrich, S. Becker, and R. H. Reussner, "A Pattern-Based Performance Completion for Message-Oriented Middleware," in *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*. New York, NY, USA: ACM, 2008, pp. 165–176.

[IKV13] IKV++. (2013, May) Medni QVT. [Online]. Available: http://projects.ikv.de/qvt

[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*. ACM, 2006, pp. 719–720.

[JBo13] JBoss Community. (2013, May) Drools - The Business Logic integration Platform. [Online]. Available: http://www.jboss.org/drools/

[JK06] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[KAAF+10] J. Kienzle, W. Al Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein, "Aspect-oriented design with reusable aspect models," in *Transactions on aspect-oriented software development VII*. Springer, 2010, pp. 272–320.

[Kap11] L. Kapova, "Configurable Software Performance Completions through Higher-Order Model Transformations," Ph.D. dissertation, KIT, 2011.

[KHCB12] F. Khennouf, A. Hettab, A. Chaoui, and M. Babahenini, "Composing activity aspect diagrams using graph transformation approach," in *Digital Information Processing and Communications (ICDIPC), 2012 Second International Conference on*. IEEE, 2012, pp. 174–179.

[KK07] J. Klein and J. Kienzle, "Reusable aspect models," in *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA*, 2007.

[KKS+12a] E. Kalnina, A. Kalnins, A. Sostaks, E. Celms, and J. Iraids, "Tree based domain-specific mapping languages," *SOFSEM 2012: Theory and Practice of Computer Science*, pp. 492–504, 2012.

[KKS+12b] M. E. Kramer, J. Klein, J. R. H. Steel, B. Morin, J. Kienzle, O. Barais, and J.-M. Jézéquel, "Achieving Practical Genericity in Model Weaving through Extensibility," 2012.

[KR10] L. Kapova and R. Reussner, "Application of Advanced Model-Driven Techniques in Performance Engineering," *Computer Performance Engineering*, pp. 17–36, 2010.

[Kra12] M. E. Kramer, "Generic and Extensible Model Weaving and its Application to Building Models," 2012.

[Kur08] I. Kurtev, "State of the art of qvt: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*. Springer, 2008, pp. 377–393.

[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*. Springer, 2007, pp. 498–513.

[MAN+12] D. Mouheb, D. Alhadidi, M. Nouh, M. Debbabi, L. Wang, and M. Pourzandi, "Aspect weaving in uml activity diagrams: a semantic and algorithmic framework," in *Formal Aspects of Component Software*. Springer, 2012, pp. 182–199.

[MBJ+07] B. Morin, O. Barais, J. Jézéquel, R. Ramos *et al.*, "Towards a generic aspect-oriented modeling framework," in *Models and Aspects workshop, at ECOOP 2007*, 2007.

[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, <ce:title>Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)</ce:title> <xocs:full-name>Graph and Model Transformation 2005</xocs:full-name>. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066106001435

[MPL+09] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.-M. Jézéquel, "Weaving variability into domain metamodels," in *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 690–705.

[MVL+08] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jézéquel, "Managing variability complexity in aspect-oriented modeling," in *Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 797–812.

[Nol09] S. Nolte, *QVT-Relations Language: Modellierung mit der Query Views Transformation*. Springer, 2009.

[Obj13] Object Management Group. (2013, May) MOF 2.0 Query/View/Transformation, v1.1. [Online]. Available: http://omg.org/spec/QVT/1.1/

[PBvdL05] K. Pohl, G. Böckle, and F. J. van der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[RBH+07] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg, "The palladio component model," *Interner Bericht*, vol. 21, 2007.

[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.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. [Online]. Available: http://books.google.de/books?id=sA0zOZuDXhgC

[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*. Springer, 2008, pp. 766–781.

[SV06] T. T. Stahl and M. Voelter, *Model-driven software development*. John Wiley & Sons Chichester, 2006.

[SVC06] T. Stahl, M. Voelter, and K. Czarnecki, "Model-Driven Software Development: Technology," *Engineering, Management, John Wiley & Sons*, 2006.

[TJF+09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 18–33.

[TKB+12] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, 2012.

[VDGD05] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester, "Automatic inclusion of middleware performance attributes into architectural uml software models," *Software Engineering, IEEE Transactions on*, vol. 31, no. 8, pp. 695–711, 2005.

[Wag08] D. Wagelaar, "Composition techniques for rule-based model transformation languages," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 152–167.

[WK03] J. Warmer and A. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[WPS02] M. Woodside, D. Petriu, and K. Siddiqui, "Performance-related completions for software specifications," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE, 2002, pp. 22–32.

[WW04] X. Wu and M. Woodside, "Performance modeling from software components," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 290–301, 2004.

List of Figures