Modeling and Analysis of Software Product Line Variability in Clafer

by

Kacper Bąk

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

© Kacper Bąk 2013

I hereby declare that I am the sole author of this thesis, except where noted. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Both feature and class modeling are used in Software Product Line (SPL) engineering to model variability. Feature models are used primarily to represent user-visible characteristics (i.e., features) of products; whereas class models are often used to model types of components and connectors in a product-line architecture.

Previous works have explored the approach of using a single language to express both configurations of features and components. Their goal was to simplify the definition and analysis of feature-to-component mappings and to allow modeling component options as features. A prominent example of this approach is cardinality-based feature modeling, which extends feature models with multiple instantiation and references to express component-like, replicated features. Another example is to support feature modeling in a class modeling language, such as UML or MOF, using their profiling mechanisms and a stylized use of composition. Both examples have notable drawbacks: cardinality-based feature modeling lacks a constraint language and a well-defined semantics; encoding feature models as class models and their evolution bring extra complexity.

This dissertation presents *Clafer* (class, feature, reference), a class modeling language with first-class support for feature modeling. Clafer can express rich structural models augmented with complex constraints, i.e., domain, variability, component models, and meta-models. Clafer supports: (i) class-based meta-models, (ii) object models (with uncertainty, if needed), (iii) feature models with attributes and multiple instantiation, (iv) configurations of feature models, (v) mixtures of meta- and feature models and model templates, and (vi) first-order logic constraints. Clafer also makes it possible to arrange models into multiple specialization and extension layers via constraints and inheritance. On the other hand, in designing Clafer we wanted to create a language that builds upon as few concepts as possible, and is easy to learn. The language is supported by tools for SPL verification and optimization.

We propose to unify basic modeling constructs into a single concept, called *clafer*. In other words, Clafer is not a hybrid language. We identify several key mechanisms allowing a class modeling language to express feature models concisely. We provide Clafer with a formal semantics built in a novel, structurally explicit way. As Clafer subsumes cardinality-based feature modeling with attributes, references, and constraints, we are the first to precisely define semantics of such models.

We also explore the notion of partial instantiation that allows for modeling with uncertainty and variability. We show that Object-Oriented Modeling languages with no direct support for partial instances can support them via class modeling, using subclassing and strengthening multiplicity constraints. We make the encoding of partial instances via subclassing precise and general. Clafer uses this encoding and pushes the idea even further: it provides a syntactic unification of types and (partial) instances via subclassing and redefinition.

We evaluate Clafer analytically and experimentally. The analytical evaluation shows that Clafer can concisely express feature and meta-models via a uniform syntax and unified semantics. The experimental evaluation shows that: 1) Clafer can express a variety of realistic rich structural models with complex constraints, such as variability models, meta-models, model templates, and domain models; and 2) that useful analyses can be performed within seconds.

Acknowledgements

I would like to thank my supervisor, Krzysztof Czarnecki, for giving me the opportunity to learn how to do research and to design a language. I was fortunate to work both on theoretical and practical aspects of language design. I am grateful for providing the guidance and direction that has led to the completion of this dissertation. I am also thankful for all the assistance that made my life in Canada a great experience.

Thank you to my colleagues in the Generative Software Development Lab and co-authors. A special thanks to Michał Antkiewicz and Zinovy Diskin. Michał always found the time to discuss my research and was full of non-conventional ideas. I admire Zinovy's dedication, patience, and structured thinking. His commitment to detail and work ethics influenced the way I think and present ideas. I am also grateful for all the refreshing discussions with GSD lab folks.

I would like to thank my external examiner, Prof. Bernhard Rumpe, for finding time to visit Waterloo and for his comments on this dissertation. I am very grateful to my internal thesis committee, Prof. Derek Rayside, Prof. Joanne Atlee, and Prof. Nancy Day, for helping me shape this research.

Thank you to proof-readers of this dissertation: Ed Zulkoski, Paul Sulzycki, Tom Rozdeba, and Agata Stepkowa. Your help is much appreciated.

Finally, a big thanks to my family, friends, Gzowski Club, and travel buddies. My parents, Jurek and Wiesia, were always very supportive and respectful. A special thanks to Mirosław Słowakiewicz who deepened my interests in science and inspired to pursue a PhD degree abroad. I am very happy that I met so many free-spirited, cool, and valuable people. It would be difficult to finish PhD without having a balanced life. Thanks for all the parties, adventures, road-trips, and random events. Much international love to you all.

Acknowledgments of Contributions

Person	Contributions
Kacper Bąk	Language design, formal semantics, implementation, testing, evaluation, and documentation. Development of Clafer compiler. Translation of variability and meta-models to Clafer. SPL exercise for students. Development of FSMLs in Prolog.
Krzysztof Czarnecki	Input on all aspects of design and language evaluation.
Andrzej Wąsowski	Input on all aspects of design and language evaluation.
Michał Antkiewicz	Input on all aspects of design and language evaluation. Development of FSMLs, ClaferWiki, ClaferMOO visualizer, and tool testing.
Zinovy Diskin	Formal semantics.
Jimmy Liang	Development of ClaferIG, and Clafer compiler. Translation to Choco.
Rafael Olaechea	Development of ClaferMOO.
Alexandr Murashkin	Development of ClaferMOO visualizer.
Leonardo Passos	Translation of CDL models to Clafer. Research on defaults.
Steven She	Development of FSMLs in Prolog. Translation of KConfig model to Clafer.
Marko Novakovic	Translation of CDL models to Clafer.
Wenbin Ji	Translation of meta-models to Clafer.
Dina Zayan	Usability study on Clafer and UML.
Bo Wang	Encoding of the Android SPL model in Clafer.
Christopher Walker	Development of ClaferWiki.
Luke Michael Brown	Development of Clafer compiler and instance generator.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software Product Line (SPL) engineering is a systematic way of building customizable software families from shared assets. SPLs bring important benefits to software development, such as improved productivity, software quality, and reduced time to market [108, 126, 73]. While traditional software development deals with one software variant at a time, SPL engineering considers many possible variants simultaneously. For a given input configuration, a SPL delivers a single software variant. The number of variants grows exponentially with the number of configuration options. The research on SPLs is concerned with methods, languages, and tools for managing variability and delivering correct software variants.

Software Product Lines are often described in terms of *problem space*, *solution space*, and a *mapping* between them (see Fig. 1.1). Both feature and meta-modeling have been used in SPL engineering to model variability. Feature models are tree-like menus of mostly Boolean — but sometimes also integer and string — configuration options, augmented with cross-tree constraints [82]. These models are typically used to show the variation of *user-relevant* characteristics of products within a product line. The models belong to the *problem space*.

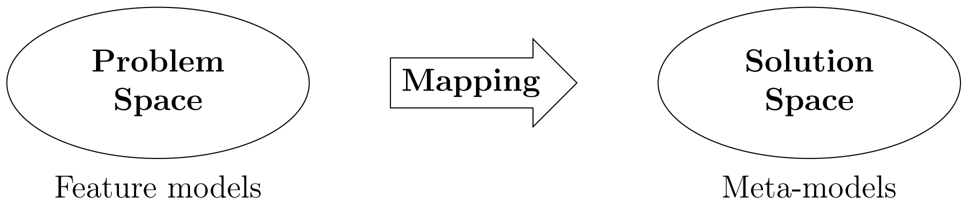


Figure 1.1: Conceptual view of a Software Product Line

In contrast, meta-models, normally represented as class models as supported by MetaObject Facility (MOF) [102], specify concepts representing more detailed aspects of products; including behavioral and architectural aspects. They belong to the *solution space*. For example, meta-models are often used to specify the component and connector types of *product line architectures* and the valid ways to connect them. The nature of variability expressed by each type of models is different: feature models capture selections from predefined choices within a fixed tree structure; meta-models support making new structures by creating multiple instances of classes and connecting them via links.

Over the last decade, the distinction between feature models and meta-models has been blurred in the literature due to (i) feature modeling extensions, such as *cardinality-based feature modeling* [42, 10], and (ii) attempts to express feature models as class models in the Unified Modeling Language (UML) [35, 44]. In fact, a number of practitioners use UML-based representations to model variability [19]. A key driver behind some of these developments has been *the desire to express both configuration options and variability of component architecture instantiations in one notation* [40, 69, 70]. Cardinality-based feature modeling achieves this by extending feature models with multiple instantiation and references. Class modeling, which natively supports multiple instantiation and references, enables feature modeling by a stylized use of containment (UML's composition) and the profiling mechanisms of MOF or UML, e.g., as in [64].

Both developments have notable drawbacks, however. An important advantage of feature modeling as originally defined by Kang et al. [82] is its simplicity; several respondents to a recent survey confirmed this view [81]. Extending feature modeling with multiple instantiation and references diminishes this advantage by introducing additional complexity. Models that contain significant amounts of multiply-instantiatable features and references can hardly be called feature models in the original sense; they are closer to class models rather than easy-to-use menus guiding configuration decisions. On the other hand, although class modeling handles the model parts requiring multiple instantiation and references naturally, the feature modeling aspects can only be clumsily simulated using composition hierarchy and certain modeling patterns. Even worse, such a solution requires inconvenient model refactorings, while according to a recent survey [19], *evolvability of variability models is one of the main challenges faced by practitioners*.

We are not aware of any notation that naturally supports integrated modeling of both problem and solution spaces of SPLs. Although specialized notations can be used, such as feature and class models, they have incompatible semantics. Consequently, it is difficult to mix such heterogeneous models, to specify and evaluate constraints over them, and it is unclear how to handle consistency between such models.

1.1 Clafer Overview

This dissertation presents *Clafer* (class, feature, reference), a class modeling language with first-class support for feature modeling. Clafer can express rich structural models, i.e., domain, variability, component models, and meta-models, augmented with complex constraints. Clafer supports: (i) class-based meta-models, (ii) object models (with uncertainty, if needed), (iii) feature models with attributes and multiple instantiation, (iv) configurations of feature models, and (v) mixtures of meta- and feature models and model templates [39], and (vi) first-order logic constraints. Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. On the other hand, in designing Clafer we wanted to create a language that builds upon as few concepts as possible, and is easy to learn.

The novelty of Clafer design is founded on:

Unification of classes, associations, and properties, and arbitrary property nesting. Clafer is based on a single concept, *clafer* (written in small letters), that unifies basic constructs of structural modeling, namely *class*, *association*, and *property* (which includes *attribute*, *reference*, *association end*, and *role*) via reification of associations, i.e., representing associations as association classes. Such a concept has the characteristics of classes (ability to nest properties under it), associations (ability to navigate over it), and attributes (ability to store values of primitive types in it). Thus, clafers can be arbitrarily nested and can play the multiple roles of classes, attributes, and references simultaneously. Clafer facilitates constructing hierarchical models, similarly to other tree-based languages, e.g., XML and feature models. Additionally, Clafer integrates subclassing into hierarchical modeling and encodes (partial) instances by classes.

Hierarchical modeling with subclassing. Clafers at any nesting level in the containment hierarchy can subclass and redefine other clafers (think: simultaneous subclassing of a unidirectional association and its target class in class diagrams). Prominent hierarchical modeling languages, e.g., XML and feature models, offer no subclassing. UML class diagrams, on the other hand, have two separate mechanisms: subclassing among classes and subclassing among associations. Clafer unifies both into subclassing among clafers.

Support for partial instances and (partial) completions. The cornerstone of Clafer's view of modeling is that (partial) instances and their (partial) completions can be encoded, without losing any information, as class models using subclassing and strengthening multiplicity constraints. Clafer uses the same syntax for specifying class and object models. One can interpret such an encoding as a syntactic unification of types and partial instances, and their completions.

Concise concrete syntax. Due to unification, Clafer offers concise concrete syntax for feature and meta-modeling. A special naming discipline for elements that make up a clafer and name resolution rules contribute to the conciseness of the constraint language.

Throughout the dissertation, if the word Clafer begins with upper case, then it refers to the language, otherwise it refers to the unifying concept of *clafer*.

1.2 Implementation and Evaluation

Designing a new language is always a major effort. It requires a good understanding of the problem domain, definition of concrete and abstract syntax, precise specification of semantics, tool development, and proper evaluation.

We provide Clafer with a formal semantics specified in terms of *sets* and *mappings*. We provide a mathematical model of Clafer's syntactical mechanism (meta-models and the overall architecture) using *formal class diagrams* [53], which is a structural modeling formalism based on category theory and diagrammatic logic [54]. The formalism has allowed us to define semantics concisely by specifying mappings between artifacts and specifying operations over the artifacts and mappings. The structures of syntactical and semantic constructs are aligned and made explicit rather than flattened and hidden in a multitude of first-order logic formulas.

The traditional notion of instantiation in OOM requires objects to be complete, i.e., be fully certain about their existence and attributes. We explore the notion of partial instantiation, which allows the modeler to omit some details of instances. It enables modeling with uncertainty and variability. We present a simple theory of partial instantiation and completion. We show that OOM languages with no direct support for partial instances can support them via class modeling, using subclassing and strengthening multiplicity constraints. Clafer uses this encoding extensively.

The language is supported by tools for model analyses[1]. Reasoners based on SAT, SMT, and CSP solvers provide rapid feedback on model correctness and enable non-trivial

[1]More information about tool support and documentation is available at http://clafer.org.

analyses. *ClaferIG* verifies, instantiates, and automatically completes partial models [7]. *ClaferMOO* computes optimal instances using multi-objective optimization [99]. Both tools rely on a Clafer compiler that is part of this dissertation. The compiler translates Clafer models to Alloy [76] and Choco [1]. The former is a class modeling language with a concise constraint notation. The latter is a constraint programming library. The compiler gives Clafer precise translational semantics and enables model analyses [23, 91].

We evaluate Clafer analytically and experimentally. The analytic evaluation argues that Clafer meets its design objectives. It shows that Clafer can encode feature and meta-models at least as concisely as state-of-the-art structural modeling languages. The experimental evaluation shows that a wide range of realistic feature models, meta-models, model templates, and domain models can be expressed in Clafer and that useful analyses can be run on them within seconds. Many analyses, such as consistency checks, element liveness, configuration completion, and reasoning on model edits can be reduced to instance finding by combinatorial solvers [18, 31, 46]; thus, we use instance finding and element liveness as representatives of such analyses.

1.3 Research Contributions

This dissertation contributes to the design of modeling notations.

Hierarchical Modeling

- we propose a language that unifies the concepts of feature and class modeling, both syntactically and semantically. In other words, it is not a hybrid language (Chapter 3).
- in this language, the basic constructs of structural modeling: *class*, *association*, and *property*, are unified into a single concept, called *clafer* (Sect. 4.1);
- this language has a compact syntax for encoding these rich semantic capabilities to make class models concise (Sect. 4.1.1 and Sect. 6.1);
- we evaluate Clafer and show that, despite its expressivity, realistic models can be analyzed within seconds (Sect. 6.2).

Partial Instances

- we demonstrate the usefulness of partial instances in OOM (Sect. 5.2);
- we present a theory of partial instantiation and completion, and precisely define an encoding of partial object models as class models via subclassing and strengthening multiplicity constraints (Sect. 5.4);
- we demonstrate that such an encoding provides syntactic unification of (partial) instances and types in a single concept *clafer* (Sect. 5.5).

Formal Semantics

- we formally specify the semantics of Clafer in a novel, structure-explicit way, by mapping to formal class diagrams, which we use as a notation for our semantic domain (Sect. 4.2);
- we identify several key mechanisms allowing a meta-modeling language to express feature models concisely, namely: concept unification, instance composition and type nesting, default singleton multiplicity, group constraints, constraints with default quantifiers, and navigation over optional clafers (Sect. 6.1);
- as Clafer subsumes cardinality-based feature modeling with references, attributes, and constraints, we are the first to precisely define semantics of such models.

1.4 Outline of the Dissertation

The dissertation is organized as follows. We introduce our running example and (i) define the challenges of representing the example using either only class modeling or only feature modeling, (ii) the challenges of representing partial instances, (iii) the challenges of mapping feature configurations to component and option configurations, and (iv) a set of design objectives for Clafer in Chapter 2. We then present Clafer in Chapter 3 and show that it naturally supports unified feature and class modeling. Chapter 4 (i) explains semantic foundations of Clafer, (ii) shows how it implements the unification of classes, associations, and properties, and (iii) shows that a Clafer model is a hierarchical view on a formal class diagram. Chapter 5 (i) shows the usefulness of partial instances in OOM, (ii) specifies their encoding as class diagrams, and (iii) demonstrates its realization in Clafer.

We evaluate Clafer analytically and experimentally in Chapter 6 and demonstrate that it satisfies the design objectives. We conclude in Chapter 8, after having compared Clafer with related work in Chapter 7.

1.5 Publications

This dissertation contains material from the following publications:

- Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and meta-models in Clafer: mixed, specialized, and coupled. SLE, 2010
- Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: Unifying Class and Feature Modeling. In *Submitted to SOSYM*, 2013
- Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Partial Instances via Subclassing. In *SLE*, 2013
- Kacper Bąk, Dina Zayan, Krzysztof Czarnecki, Michał Antkiewicz, Zinovy Diskin, Andrzej Wąsowski, and Derek Rayside. Example-Driven Modeling. Model = Abstractions + Examples. In *ICSE*, 2013
- Michał Antkiewicz, Kacper Bąk, Krzysztof Czarnecki, Dina Zayan, Andrzej Wąsowski, and Zinovy Diskin. Example-Driven Modeling Using Clafer. In *MDEBE*, 2013

Chapter 2

Challenges of Modeling Variability in Software Product Lines

In this chapter we introduce our running example of a telematics product line. We show how feature models naturally express the problem space and how class-based meta-models express the solution space. We discuss three issues of modeling variability in SPLs: 1) the necessity to merge feature and class models in a single notation, 2) the necessity to represent partial instances, and 3) the need to support relating (mapping) feature configurations to component and option configurations. We demonstrate the challenges of representing the solution space with feature models and the problem space with class models. Finally, we postulate design goals for a language that could naturally express both the problem and the solution space.

2.1 Modeling Variability in SPLs: An Example

Vehicle telematics systems integrate multiple telecommunication and information processing functions in an automobile, such as navigation, driving assistance, emergency and warning systems, hands-free phone, and entertainment functions, and present them to the driver and passengers via multimedia displays. Configurations of a telematics system may differ among car models (see Fig. 2.1). The bigger blue boxes indicate displays; the smaller boxes in the middle indicate Electronic Control Units (ECUs) that control the displays. The car in Fig. 2.1a has only one display for the driver. The car in Fig. 2.1b has two displays: one for the driver and one for the front-seat passenger; both displays are controlled

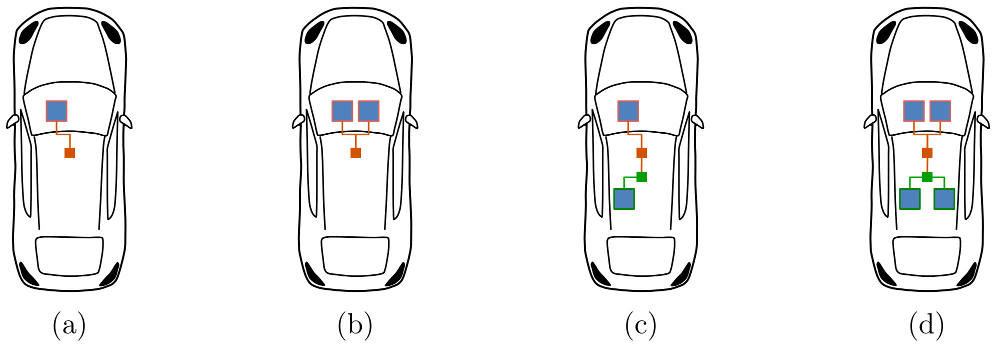


Figure 2.1: Sample configurations of a telematics system

by the same ECU. The car in Fig. 2.1c has one display in the front and one in the back; the displays are controlled by separate ECUs. The car in Fig. 2.1d has a separate display for each person in the car.

Figure 2.2 shows a variability model of a telematics product line—our running example. The features offered are summarized in the *problem-space* feature model (Fig. 2.2a). It is a tree, whose root telematics refers to the product to be configured. The children are the product's features related by the *sub-feature* relationship, which expresses hierarchical dependencies. A feature is either mandatory (indicated by a filled circle), e.g., channel, or optional (indicated by an empty circle), e.g., extraDisplay. A feature is, basically, a Boolean choice (sometimes with a numeric or string attribute) that can be either selected or excluded. Mandatory features are always selected, provided that their parent is also selected. Alternative choices are gathered under the xor-group channel, marked by the arc between edges. By default, each channel has one associated display (as in Fig. 2.1c); however, we can add one extra display per channel (as in Fig. 2.1d), as indicated by the optional feature extraDisplay. Finally, we can choose large or small displays (size). Any configuration allowed by the feature model in the problem space (the left half of Fig. 2.2) must be somehow realized in the solution space, whose model is presented in the right half of the figure. The solution space consists of three major parts.

The first one is a high-level abstract meta-model of components making up a telematics system (Fig. 2.2b). There are two types of components: *ECU*s and *display*s. Each *display* has exactly one *ECU* as its server. All components have a version.

Components themselves may have options, like the display size or cache, which constitute the second part of the solution space (Fig. 2.2c). We can also specify the cache size and decide whether it is fixed or can be updated dynamically. Thus, the solution space should include a class model of component types and a feature model of component options.

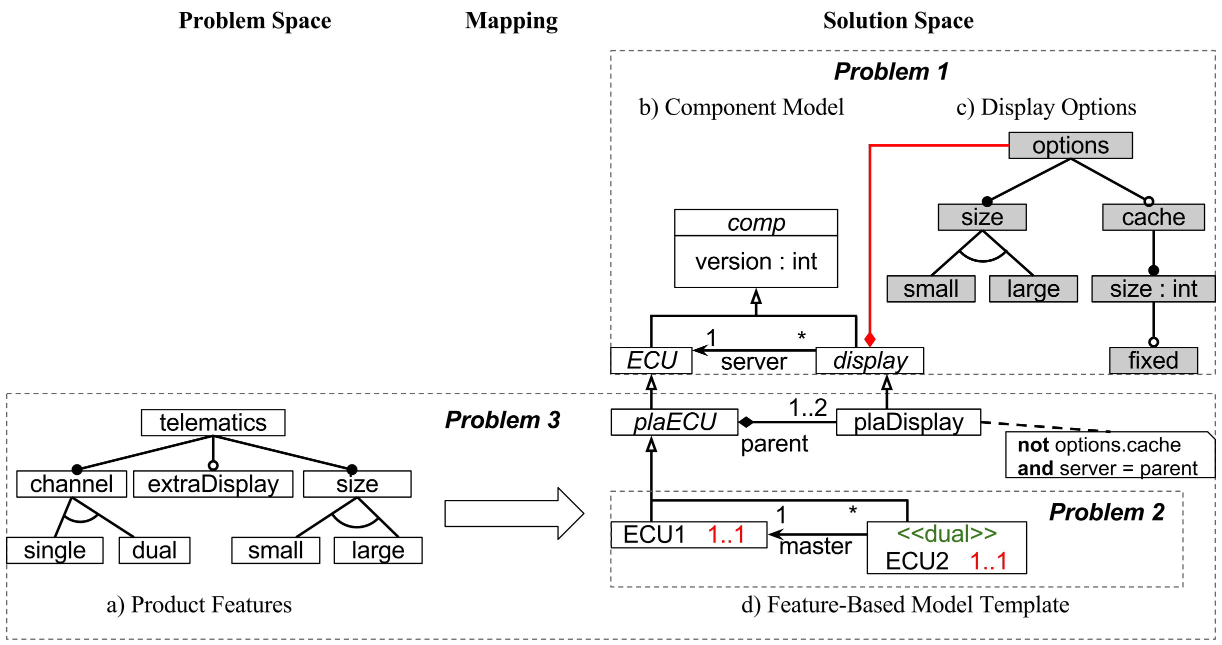


Figure 2.2: Telematics product line

Moreover, the component meta-model in Fig. 2.2b is too generic and is not well aligned with the actual products specified in the problem space. We want to add more information to specialize and extend the component model to create a particular *template* for products offered by the product line. A template fixes most of the architectural structure, but leaves some points of variability to match the variability offered by the product line. Figure 2.2d shows an extension of the abstract component model to serve as a template. The abstract class *plaECU* (product line architecture ECU) specifies that each ECU will have either one or two plaDisplays. We *specialize* the component meta-model by adding extra information via constraints: none of the displays has cache, and we constrain the server reference in each plaDisplay, so that it points to its associated ECU. A concrete product must have at least one ECU. Hence, there are two singleton subclasses, ECU1 and ECU2 (with multiplicities 1..1), that serve as a specification of objects. We choose to specify objects by singleton classes because for ECU2 we need to *extend* the base type with new property master that was not specified in the high-level component class. Modeling ECU2 instance as a singleton class solves this problem (a more detailed discussion of modeling objects by singleton classes can be found in Chapter 5).

We need to endow the class model with a variability mechanism aligned with variability provided by the feature model (from the solution space). One way of doing this is to make the existence of some classes in the template optional by annotating them with propositional formulas composed from features offered by the feature model (so called *presence conditions* introduced in Feature-Based Model Templates (FBMTs) [39]). In Fig. 2.2d, class ECU1 is mandatory and class ECU2 is optional, because its presence in the model is regulated by the condition «dual», which refers to a feature from Fig. 2.2a. The presence condition means that in a concrete configuration ECU2 will be included if the feature dual is selected; it will be removed otherwise. The two configurations will be ordinary class diagrams, i.e., instances of the UML meta-model of class diagrams. The model template, however, represents multiple class diagrams and can be considered a *partial instance* of the meta-model. Thus, FBMTs relate the problem-space feature configurations to the solution-space component and option configurations. A block-arrow in Fig. 2.2 represents this mapping. We will provide a precise specification of the complete mapping later in this dissertation. Note that the mapping requires that the solution space models should be partially specified (are partial instances) to represent a family of models.

The example demonstrates three issues of modeling variability in SPLs: 1) the necessity to merge feature and class models in a single solution space, 2) the necessity to represent partial instances, and 3) the need to support relating (mapping) feature configurations to component and option configurations. In the next section, we will show that managing the issues is not straightforward and challenging. Correspondingly, we will refer to them as *Problem 1*, *Problem 2*, and *Problem 3*.

2.2 Three Problems

2.2.1 Problem 1: Merging Feature and Class Models

The solution space in Fig. 2.2 contains a class-based meta-model and a feature model. To capture our intention, the models are connected via UML composition. As the precise semantics of such notational mixture is not clear, this connection should be understood only informally for now. Basically, we have two choices to model components and options in a single notation: either enrich feature modeling to allow it to capture class modeling, or encode feature models as class models. We will consider them in the two consecutive subsections.

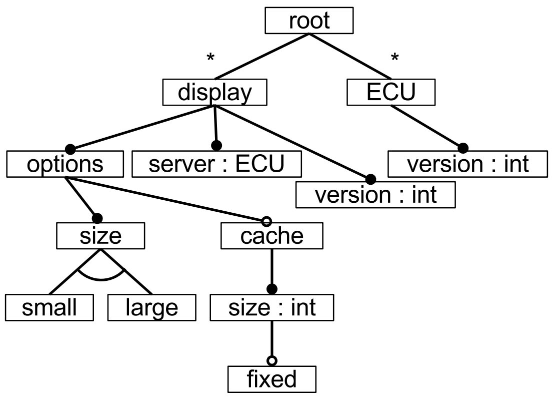


Figure 2.3: Cardinality-based feature model of components

Class Modeling via Feature Modeling

Figure 2.3 shows the component part of the model. The model introduces a synthetic root feature; display and ECU can be multiply instantiated, but cannot be *abstract*; and display has a server subfeature representing a reference to instances of ECU. Versions are added to display and ECU to match the meta-model in Fig. 2.2b, or we could extend the notation with inheritance. The latter would bring the cardinality-based feature modeling notation very close to class modeling, posing the question whether class modeling should not be used for the entire solution space model instead. Furthermore, the semantics of such an extended notation is unclear.

We may conclude that cardinality-based feature modeling blurs the distinction between feature modeling and class modeling. It encompasses mechanisms characteristic of class modeling, such as multiple instantiation and references, and could even be extended further toward class modeling, e.g., with inheritance; however, the result can hardly be called 'feature modeling' in its classical sense, as it clearly goes beyond the original scope of feature modeling [82].

Feature Modeling via Class Modeling

Figure 2.4 shows only the options model, as the component model remains unchanged (as in Fig. 2.2b). A subfeature is either an attribute (if it has no other subfeatures) or a class (if it has features) – we choose the simplest suitable language construct. Subfeature relationships are represented either as property nesting or as UML composition. Feature multiplicities correspond to property multiplicities. The xor-group is encoded by enumeration.

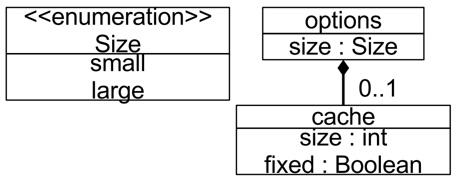
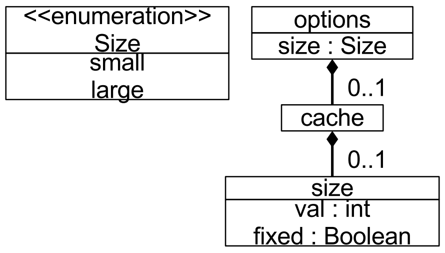
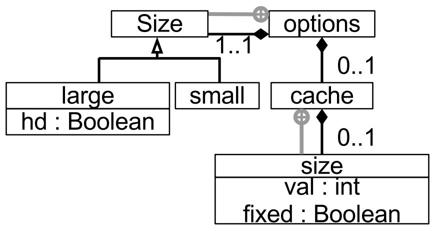


Figure 2.4: Meta-model of display options



(a) Iteration 1



(b) Iteration 2

Figure 2.5: Evolved meta-model of display options

Representing a feature model as a UML class model works reasonably well for our small example; however, it does have several drawbacks:

- *Limited property nesting and model refactoring.* The feature model shows fixed as a property of size by nesting. This intention is lost in the class model, in which fixed is a property of cache rather than size; particularly, if the attribute size was optional, the attribute fixed could exist even if size was eliminated. To fix this drawback, we could add an OCL constraint expressing the dependency, but hiding an important structural information within constraints is not generally advisable.

  A better solution would be to *reify* attribute size as a class contained in cache as shown in Fig. 2.5a, in which the structural dependency is explicit. Note also that in some cases, attributes' reification would naturally go through subclassing rather than containment. Suppose, for example, that we need to add an optional property hd (high definition) to large displays. A natural way to do this is shown in Fig. 2.5b, which is again a substantial refactoring of the initial class model from Fig. 2.4. In contrast, adding property hd to the feature model in Fig. 2.2a amounts to plain nesting subfeature hd under feature large.

These examples show a general drawback of ordinary class modeling in the context of gradual model development. Modeling properties by attributes is very compact, but does not allow further nesting. On the other hand, modeling properties by classes leads to bulky models; even worse, there are several ways of such modeling, which may present a problem for an inexperienced modeler.

- *Name clashes.* By default, class diagrams offer a single namespace for class names. Feature names, however, often repeat in different parts of the feature model, e.g., the name size is used three times in Fig. 2.5a. Name repetitions may easily lead to name clashes. For example, if we make the enumeration Size a class, the name of the new class would clash with the class size representing the display size; thus, we would have to rename one of them, or use nested classes (Fig. 2.5b), which would further complicate the model.

- *Limited support for groups.* Converting an xor-group to an or-group in feature modeling is simple: the empty arc is replaced by a filled one. For example, size (Fig. 2.2a) may be converted to an or-group in a future version of the product line to allow systems with both large and small displays simultaneously. Such a change is tricky in class models: we need to refactor size to a class with two subtypes: small and large (Fig. 2.5b). Then we would either allow one to two objects of type size and write an OCL constraint forbidding two objects of the same subtype (small or large), or use overlapping inheritance.

Thus, existing class modeling notations, e.g., UML class diagrams, are inadequate for feature modeling, especially in the context of gradual model development and evolution. Similar arguments apply to other existing class-based modeling languages, such as MOF and Alloy.

2.2.2 Problem 2: Representation of Partial Instances

Annotations, such as «dual» in Fig. 2.2d, are a mechanism of introducing variability into models, which makes the models partially specified. Such models can be viewed as partial instances of their meta-models (here, of the UML meta-model of class diagrams). In our example, it is unclear whether the class ECU2 actually instantiates its meta-class CLASS or not. The modeler can make the choice later as more knowledge becomes available. He would then complete the model (reduce variability in it) to arrive at a single configuration. Although class diagrams provide syntactic support for annotations, the semantics of annotations expressing variability goes beyond the semantics of standard class diagrams. Thus, the problem is that the partiality of models is expressed outside the notation of class diagrams. Standard tools cannot correctly interpret such enriched models, and specialized tools are required [46, 59].

2.2.3 Problem 3: Mapping Features to Component Configurations

Relating heterogeneous models by a mapping is a non-trivial task. For example, a FBMT in Fig. 2.2 relates a feature model, a class model (of components), and (implicitly) their metamodels. As annotations, such as «dual», change the class model itself, complicated syntactic checks are needed to guarantee the correctness of all template configurations. For example, when «dual» is deselected and ECU2 is consequently removed, then master may become a dangling association (because it is mandatory and has no presence condition). Thus, the configured template does not conform to the UML meta-model for class diagrams. Verification of annotative model templates is a non-trivial task and requires specialized tools [46].

2.2.4 Toward a Solution

We conclude that a solution to the aforementioned issues is to design a *(class-based) meta-modeling language with first-class support for feature modeling.* We postulate that such a language should satisfy the following design goals:

1. *Provide a concise notation for feature modeling*
2. *Provide a concise notation for class modeling*
3. *Allow mixing of feature models and class models*
4. *Provide support for partial instances*
5. *Use a minimal number of concepts and have a uniform semantics*

The last requirement is aimed at a language that unifies the concepts of feature and class modeling as much as possible, both syntactically and semantically. In other words, we do not want a hybrid language. We see the following advantages of unification: 1) the ability to encode a variety of models, especially allowing flexible mixing of feature and class models as shown by the example, 2) the ability to relate feature and class configurations via ordinary constraints, 3) a common infrastructure to support analyses of these models, and 4) simplified implementation of the tools.

2.3 Concluding Remarks

In this chapter we introduced the running example. We showed the deficiencies of representing it fully using either cardinality-based feature modeling or class-based meta-modeling. The former notation becomes complex and lacks precise semantics. The latter lacks first-class support for hierarchical models, which impedes their evolution. We also presented the challenge of relating the problem and the solution space models.

We propose to address these issues by designing a language that unifies feature and meta-modeling. Such a unified language would be based on first principles that govern both formalisms. An alternative would be to design a hybrid language that integrates feature and meta-modeling, or to design an infrastructure that enables reasoning on the two types of models together. First, syntactic integration of two notations in a hybrid language is likely to cause ambiguities. For example, an edge that relates two features represents a sub-feature relationship; an edge that relates two classes represents a bidirectional association. The former expresses an implication between feature selections, while the latter does not have to express an (analogous) existential dependency between instances of classes. Second, in the end, both alternative solutions would require a uniform semantics to support reasoning about the two notations. Hence, a uniform semantics is unavoidable.

Chapter 3

Clafer: Unifying Class and Feature Modeling

In this chapter we model the telematics example in Clafer. We show how the language addresses the three problems introduced in the previous chapter. First, we encode the feature model of component options and the meta-model of components. We then show that both models can be arbitrarily mixed via inheritance and references. Inheritance allows us to extend models with new elements and to specialize them by adding constraints. Constraints are a very flexible mechanism that support (partial) configuration of variability models (e.g., feature models) and models with variability (e.g., model templates). We also demonstrate that constraints can couple feature and class-based meta-models, i.e., when a feature model is configured, then it is reflected in a specific configuration of components. Finally, we present an instance of a Clafer model encoded at the class level via subclassing.

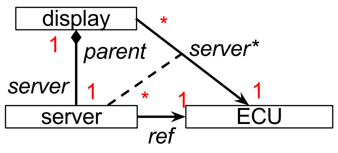
3.1 Clafer vs. the Three Problems

3.1.1 Clafer in a Nutshell

Clafer has a minimalistic syntax but rich semantics that unifies *class*, *association*, and *property* (which includes *attribute*, *reference*, and *role*) into a single concept called *clafer*. For example, Fig. 3.1a shows a sample Clafer model consisting of two clafer declarations. First, a clafer display is declared, then the declaration of the clafer server is nested under the first declaration (the latter, implicitly, is nested under the *synthetic root* clafer). A

```
display *
  server → ECU 1..1
```

(a) Clafer model



(b) Rendering as a UML Class Diagram

Figure 3.1: Clafer model and its meaning

clafer declaration includes multiplicities, and may optionally contain a superclafer or a reference to a clafer or both. In the example, clafer display has multiplicity *: there can be any number of instances of this clafer. Clafer server refers to clafer ECU to be defined somewhere in the model, and has multiplicity 1..1: each display has exactly one ECU server.

A clafer declaration (server) specifies a relationship between its parent class (display) and its target class (ECU), and declares the following (see Fig. 3.1b):

1. A new class server and a bidirectional composition association, which enables navigation to the introduced class via the end *server* and back to the parent/owner;

2. A unidirectional association *ref* to navigate to the target class from the new class;

3. A unidirectional association *server** to navigate to the target from the parent class (note that the * mimics a dereferencing operator on *ref*). By default, it is assumed that for any instance this of class display, the equality this.*server*.*ref*=this.*server** holds. In fact, this condition means that class server can be considered as a *reification* of association *server** in the sense of UML's association classes. Figure 3.1b uses the UML syntax for representing an association class (the dashed line from the association to server) to indicate this fact.

If a clafer has no reference (i.e., has neither class ECU nor maps *ref* and *server**), then it corresponds to pure containment (server would be still contained by display). We call clafers with references *reference clafers*, otherwise they are *basic clafers*. Further in Sect. 4.1 we will make these concepts more precise.

```
1 abstract options
2   xor size
3     small ?
4     large ?
5   cache ?
6     size → int
7       fixed ?
8   [ small && cache ⟹ fixed ]
```

Figure 3.2: Feature model of component options in Clafer

3.1.2 Solving Problem 1: Merging Feature and Class Models

Let us model the running example in Clafer. In general, a Clafer model consists of three types of constructs: *clafers*, *constraints*, and *goals*. In this dissertation, we are only concerned with *clafers* and *constraints*; goals are discussed elsewhere [99]. Each line in Fig. 3.2 declares a new clafer (lines 1-7) or a constraint (in square brackets, line 8). Clafers can be arbitrarily nested (in the containment hierarchy) using indentation: clafer options is at the top level; clafers size (line 2), cache, and the constraint are nested under options; clafers small and large are nested under size; etc. Below we discuss the Clafer model step by step. Appendix C.1 provides the entire model of the running example for reference.

Feature Modeling

The Clafer model in Fig. 3.2 corresponds to the model of display options in Fig. 2.2c. The clafer options is *abstract* (cf. keyword **abstract**)—it has no direct instances, that is, all its instances are instances of some other *concrete* (i.e., non-abstract) clafer extending it (similar to abstract class in UML). One of the applications of abstract clafers is to support reuse, as we will show shortly.

The clafer options contains a hierarchy of features and a constraint. Each clafer can contain any number of children clafers and constraints, shown by indentation. Clafers can be preceded by *group cardinality*, which constrains the number of instances of children clafers. For example, the keyword **xor** means that size allows either small or large but not both (nor none of the two) as a child instance.

Clafers are constrained by *multiplicity* constraints: a multiplicity is an interval $m..n$, where $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$, assuming that $i < *$ for all $i \in \mathbb{N}$. A clafer can only have the number of instances l from this interval: $m \leq l \leq n$. Besides direct notation m..n, some syntactic sugar exists. For example, the clafer cache is followed by the question mark ? (meaning 0..1), i.e., cache is optional. By default the multiplicity for clafers is 1..1, so size is mandatory; for top-level abstract clafers it is 0..*, so there is no restriction on options. As the examples will show, such a design choice smoothly integrates feature and class models.

Similarly to feature models, Clafer models have an important property: a *child* clafer cannot exist unless its *parent* also exists. The clafer size → int corresponds to a feature with an attribute of type integer; it also nests another clafer fixed (cf. Fig. 2.2b). If cache is eliminated, then its children size → int and fixed are eliminated too.

Constraints

Constraints express dependencies among clafers or restrict string and integer values. For example, the constraint in line 8 requires that a small display with cache must have cache of fixed size. The clafer small is found within size; the full path inserted by the compiler will be this.size.small. We designed constraints after Alloy; its notation is elegant, concise, and expressive enough to restrict both feature and class models. We fully define the constraint language in Appendix B.2.

Each clafer introduces a local namespace. For example, the two different clafers named size exist in different namespaces (one is within options, and one within cache). In general, names are path expressions, used for navigation like in OCL or Alloy. Clafer has name resolution rules; they are important when resolving clafer names used in constraints and clafer definitions. A name is resolved in the context of a clafer using the following rules. First, it is checked whether it is a special reserved name, such as this. Second, it is looked up in descendants in the containment hierarchy in a breadth-first-search manner. If it is not found, the algorithm searches within the ancestor clafers. Otherwise, the name is looked up in all top-level definitions. If the name cannot be resolved or is ambiguous within one rule, an error is reported.

Class-Based Meta-Modeling

Figure 3.3 shows a component meta-model (from Fig. 2.2b) encoded in Clafer. Clafer version (line 2) corresponds to the attribute of the class *comp* in Fig. 2.2b; and clafer server (line 7) corresponds to the unidirectional association pointing to the class ECU in Fig. 2.2b. All concrete clafers are contained by their parents (the synthetic root is a parent of top-level clafers). Clafers declared using the arrow notation (version and server) are *reference clafers*,

```
abstract comp
  version → int

abstract ECU : comp

abstract display : comp
  server → ECU
  ‘options // shorthand for options : options
  [ version ≥ server.version ]
```

Figure 3.3: Component meta-model in Clafer

OOM Concepts	Clafer Concepts
class property reified association	clafer
single inheritance	single inheritance
containment	clafer nesting
unidirectional association bag-valued set-valued	reference clafer bag set
multiplicity	multiplicity

Table 3.1: Corresponding concepts in OOM and Clafer

i.e., they hold references to instances (note that primitive types are clafers, too). All other clafers are called *basic clafers*—they have no references. Table 3.1 summarizes OOM and Clafer constructs. Although association reification is a transformation in UML, in Clafer it is captured by the concept of clafer, as associations are always reified.

Clafer supports single inheritance. If clafer A extends clafer B (written A : B), then every instance of A is also an instance of B. In Fig. 3.3, clafer ECU inherits version from comp. The clafer display additionally extends comp by adding two clafers and a constraint stating that display's version cannot be lower than server's version. Inheritance in Clafer is disjoint by default.

Quotation (see ‘options in Fig. 3.3) is a syntactic sugar for inheritance. Syntactically, quotation ‘A expands to A : A. It introduces a clafer that extends another clafer (defined elsewhere in the model) and reuses its name. In the running example, quotation will

```
abstract plaECU : ECU
  plaDisplay : display 1..2
    [ !cache
      server = parent ]

ECU1 : plaECU

ECU2 : plaECU ?
  master → ECU1
```

Figure 3.4: Architectural template in Clafer

effectively include options from Fig. 3.2 as a part of display in Fig. 3.3.

While inheritance is about sharing a supertype, reference clafers enable sharing instances. The reference clafer server will point to some ECU instance. In general, there may be several displays that will reference the same ECU instance as a server.

Mixing class and feature models in Clafer is done via inheritance or reference clafers. If a clafer has a supertype, the supertype can be any clafer, regardless of whether it plays the role of a feature or a class. Similarly, any clafer can be the target of a reference clafer. The concept of clafer is flexible. It can model (reified) unidirectional associations, set- and bag-valued collections, and containment among clafers, which mitigates the problems discussed in Sect. 2.2.

3.1.3 Solving Problem 2: Representation of Partial Instances

Clafer natively supports partially specified models. It can encode model templates in a way that configurations are always syntactically correct. Figure 3.4 encodes the template from Fig. 2.2d in Clafer. The predefined keyword parent points to the current instance of *plaECU*, which is either ECU1 or ECU2. Instead of making existence of a class optional, it is assumed that the class exists, but has multiplicity 0..1. Its presence condition becomes a normal constraint that regulates instantiation — as it is typically done in class modeling. The constraint can easily relate feature and class models, because they are in a unified notation. Thus, the model template contains variability and is always a valid instance of the assumed meta-model. Later, in Chapter 5, we show a general mechanism of encoding partial instances at the level of models so that partiality can be expressed without changing

```
telematics
  xor channel
    single ?
    dual ?

  extraDisplay ?

  xor size
    small ?
    large ?

  [ dual ⇔ ECU2
    extraDisplay ⇔ #ECU1.plaDisplay = 2
    extraDisplay ⇔ (ECU2  ⟹  #ECU2.plaDisplay = 2)
    large ⇔ !plaECU.plaDisplay.options.size.small
    small ⇔ !plaECU.plaDisplay.options.size.large ]
```

Figure 3.5: Feature model with mapping constraints

the semantics of the base language.

3.1.4 Solving Problem 3: Mapping Feature to Component Configurations

Having defined the architectural template, we can expose the variability points present in it as a product-line feature model. Figure 3.5 shows this model (cf. Fig. 2.2a) along with constraints coupling its features to the variability points of the template. The template in Fig. 3.4 allows the number of displays (plaDisplay under ECU1 and ECU2) and the size of every display to vary independently. We want to further restrict the variability as stipulated in the feature model, however, requiring either all present ECUs to have two displays or all to have no extra display, and either all present displays to be small or all to be large. We opted to explain the meaning of each feature in terms of the model elements to be selected rather than defining the presence condition of each element in terms of the features. Both approaches are available in Clafer, however.

Constraints allow us to restrict the model to a single instance (to configure it). Figure 3.6 shows top-level constraints specifying a single product, with two ECUs, two large displays per ECU, and all components in version 1. The configuration corresponds to the one in Fig. 2.1d. Tools, such as *ClaferIG* [7], can automatically instantiate the product

```
// concrete product
[ dual && extraDisplay && telematics.size.large ]
[ all c : comp | c.version = 1]
```

Figure 3.6: Constraint specifying a single product

```
e1 : ECU1
  d1 : plaDisplay
    s1 : server → e1
    o1 : options
      s1 : size
        l1 : large
    v1 : version → 1
  d2 : plaDisplay
    s2 : server → e1
    o2 : options
      s2 : size
        l2 : large
    v2 : version → 1
  v3 : version → 1
e2 : ECU2
  m1 : master → e1
  d3 : plaDisplay
    s3 : server → e2
    o3 : options
      s3 : size
        l3 : large
    v4 : version → 1
  d4 : plaDisplay
    s4 : server → e2
    o4 : options
      s4 : size
        l4 : large
    v5 : version → 1
  v6 : version → 1
```

Figure 3.7: Configuration in Clafer

line by deriving a configuration of the architectural template.

Clafer offers the same syntax for encoding models and instances (configurations), and the latter can be partial. Figure 3.7 shows a Clafer model that encodes exactly one configuration that was previously specified by constraints. The encoding is done by hierarchical *redefinition* among clafers — subclassing among clafers and subclassing among references. For example, in Fig. 3.4 the clafer plaDisplay is nested under *plaECU*, thus in Fig. 3.7 the singleton clafer d1 subclasses plaDisplay and is nested under e1. For reference clafers, the target of the reference gets redefined. For example, in Fig. 3.4 the clafer master points to ECU1 and is nested under ECU2; in Fig. 3.7 the clafer m1 (line 16) subclasses master, is nested under e2, and points to e1, which is a subclass of ECU1 (line 1). Note that redefinition of basic or reference clafers allows refining their multiplicities.

3.2 Concluding Remarks

In this chapter we introduced Clafer. We showed, on the telematics example, that Clafer syntactically unifies feature and meta-modeling, and that both types of models can be mixed (using inheritance and references), specialized (partially completed), and coupled (related using constraints). The constraint language plays a very important role: it specifies (partial) configurations of models and provides a mapping between feature and meta-models. Finally, we encoded an instance of a Clafer model as a configuration of classes related via subclassing. Thus, the same syntax is used for representing class and (partial) object models. We make this encoding precise and general in Chapter 5.

Chapter 4

Concept Unification and Arbitrary Property Nesting

In this chapter we show that Clafer's concise concrete syntax encodes a rich semantics. First, we present the abstract syntax of Clafer models. We formalize the abstract syntax so that its structure follows the concrete syntax. We then define the concept *clafer* that unifies classes, associations, and properties. We discuss different kinds of clafers and explain how arbitrary property nesting is realized. Next, we present the integration of inheritance into hierarchical modeling. We formally specify the aforementioned mechanisms by defining meta-models expressed as formal class diagrams. Finally, we show that Clafer models are hierarchical views of class diagrams, and, similarly, instances of Clafer models are hierarchical views of object diagrams.

4.1 Anatomy of a Clafer Model and Its Instantiation

This section discusses the basic ingredients of Clafer syntax and presents instantiation of Clafer models. The technical details of concrete syntax can be found in Appendix B.1. We start with the abstract syntax.

4.1.1 Clafers as Views to Class Diagrams

The rich semantics of Clafer models is expressible in a concise syntax. We designed the concrete syntax so that it hides the complexity of its semantics. The mechanism is realized

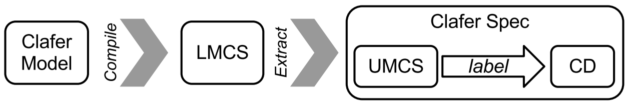
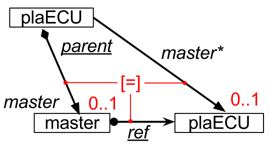


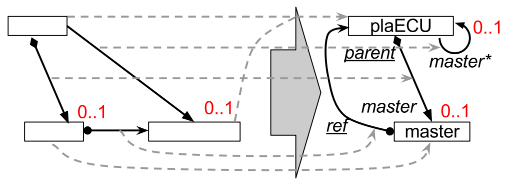
Figure 4.1: Architecture of Clafer's syntax

```
plaECU *
  master → plaECU ?
```

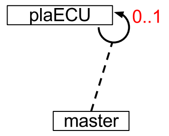
(a) Clafer model



(b) LMCS



(c) Labeling as a graph mapping



(d) UML Class Diagram

Figure 4.2: A Clafer model (a), its compilation (b), view extraction (c), and the rendering using reified association (d)

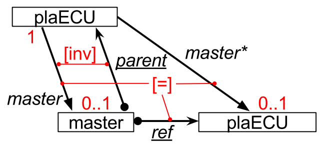
via the architecture of Clafer's syntax shown in Fig. 4.1. A Clafer model has a front-end—the Clafer model as such—, an intermediate representation—a Labeled Multi-Clafer Shape (LMCS)—, and a back-end—a class diagram. The Clafer spec provides a hierarchical view on the class diagram. It explicates that LMCS encodes this view by a *labeling mapping* from an Unlabeled Multi-Clafer Shape (UMCS) (which corresponds to an LMCS with all labels/names stripped). We provide an example to motivate and explain each component.

Figure 4.2a shows a Clafer model where plaECU is a top-level basic clafer with an unrestricted multiplicity. The optional reference clafer master is contained within plaECU and, simultaneously, has plaECU as a reference target. Furthermore, it is possible to navigate from the top-level plaECU to the one pointed to by master. Figure 4.2d shows an intuitive meaning of the model, i.e., a UML class diagram with a reified association being a loop. This intuitive meaning is precisely captured in Fig. 4.2b by an LMCS that follows the concrete syntax of Clafer. LMCS defines Clafer models in terms of *formal class diagrams* [53] (formal CDs or just CDs for short), which we use as a notation for our semantic domain. In general, an LMCS is a tree-like structure composed by joining Labeled Clafer Shapes (LCSs). A single LCS defines a clafer declaration. In the example the LMCS is composed of only one LCS.

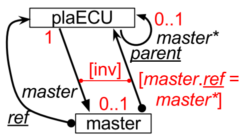
A formal CD is a graph with additional labels encoding constraints. For the CD in Fig. 4.2b, the graph encompasses three nodes and three edges, and the constraint labels denote multiplicities and a commutativity constraint ([=]); special arrow tails and heads also encode constraints that we will explain shortly. Nodes of the graph (think of UML classes) are interpreted as *sets* (of their instances). Edges (think of UML associations) are interpreted as *mappings*, i.e., sets of links mapping elements from the source set to the elements of the target set. The commutativity constraint denotes that the mapping *master** is the sequential composition of *master* followed by $\underline{ref}$. In fact, this equality means that class master together with its pair of associations ($\underline{parent}$, $\underline{ref}$) can be considered as reification of association *master** (which is shown by a dashed line in the UML diagram in Fig. 4.2d).

We use the following notation for maps. Predefined maps, e.g., $\underline{parent}$, are underlined. By default, maps are partially defined and multi-valued, and are denoted by arrows with a black triangle head, see, e.g., the shapes of arrows *master* and *master** in Fig. 4.2b. An open arrow head (e.g., arrow $\underline{ref}$) means that the map is *single-valued*: each master points to at most one plaECU. A black bullet arrow tail means that the map is *total*: each master points to at least one plaECU. A black diamond arrow tail (arrow *master*) denotes *containment* considered as a conjunction of two conditions: multiplicity 1 (there is one and only one plaECU for a master) and *existence dependency* (master deletion implies its plaECU deletion). The first condition is often referred to as *non-sharing* (and multiplicity is sometimes relaxed to 0..1). The second condition is also referred to as *cascade deletion*. Although it is not expressible in the CD formalism described in this work (which does not have any means of expressing dynamic constraints), it is an important part of Clafer semantics. Existence dependency can be formalized in the framework of CD with dynamic predicates described in [49]. Our arrow notation for maps is summarized in Tab. A.1 in the Appendix.

Notice, however, that the LMCS in Fig. 4.2b is not a valid class diagram, because it contains two classes named plaECU. What is the meaning of this strange diagram then? In contrast with class diagrams (where names are unique), the Clafer model (and the corresponding LMCS) distinguishes two different *roles* that instances of class plaECU can play: (i) being the parent of reference master, and (ii) being the target of the reference. What Fig. 4.2b actually encodes is a *mapping* from a diagram of *roles* to a diagram of classes and associations, as shown in Fig. 4.2c. The source of the mapping is the carrier graph of the diagram from Fig. 4.2b with all labels/names stripped. The target is a formal class diagram that makes the meaning of the class diagram from Fig. 4.2d precise. Indeed, as an object of class master is supposed to reify a *master* link, such an object must have



(a) Labeled Clafer Shape



(b) Class Diagram

Figure 4.3: Sample LCS and corresponding CD

a *source projection* reference (to the source of the association) that returns the source component of the link, and a *target projection* reference (to the target of the association) that returns the target component of the link. In Fig. 4.2c, these projection references are given by *parent* and *ref* associations (maps) in the target CD.

The mapping specified in diagram Fig. 4.2c consists of links assigning labels to roles; they are shown with dashed lines. The two links targeting at the same class plaECU show that class plaECU plays the two roles of being both the parent and the target of association *master*. Note that the mapping preserves the graph structure: it maps edges to edges so that their sources and targets are respected. This preservation is an important condition to be respected by labeling.

We will say that Fig. 4.2c describes a view on the class diagram, and call the mapping a *view mapping*. For a complex Clafer model consisting of multiple clafers, the role graph has the shape of multiple triangles joined together into a hierarchical structure that we call an UMCS; labeling again amounts to a mapping into a complex class diagram behind the Clafer model (see, for example, Fig. 4.5c). Thus, a Clafer model is compiled into an LMCS, whose labeling encodes a mapping from an (unlabeled) UMCS to a class diagram. We will again call this mapping a *view mapping*, and say that it is *extracted* from the LMCS. The view mapping will be crucial for reverse encoding of object diagrams that instantiate the back-end class diagrams into instances typed over LMCS, in which different roles played by the same object are explicit.

Below we consider Clafer syntax and LMCS compilation in more detail.

4.1.2 Clafer Shape

The diagram in Fig. 4.3a is a more detailed representation of the diagram in Fig. 4.2b. Nodes of the diagram denote roles played by the involved classes, and edges are roles played by mappings (unidirectional associations). The bidirectional containment association is split into two mappings, which are declared as mutually inverse. Note the predicate

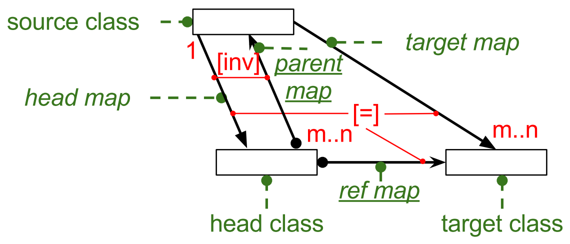


Figure 4.4: Unlabeled Clafer Shape

declaration [inv]($master$, $\underline{parent}$), which is visually shown by the label [inv] hung on the two arrows. Furthermore, the diagram carries the multiplicities of the associations $master$ and $master*$, which are equal because mapping $\underline{ref}$ is single-valued.

Thus, the diagram of roles is a graph endowed with predicate labels declaring certain properties of the mappings involved. We will call such graphs *DP-graphs*, meaning *graphs with diagram predicates*. In fact, formal CDs are nothing but DP-graphs in some predefined signature of diagram predicates required to express static semantics of UML class diagrams. Now we can say that a clafer declaration is compiled into a specific DP-graph (formal CD) of a specific shape specified in Fig. 4.4. We call this specific DP-graph an Unlabeled Clafer Shape, and name its elements as shown in the figure.

The clafer shape has a standard visual layout: the source class is always above the head class; the target class is to right of the head class. The head class indicates the clafer introduced by the compiled declaration; the source class relates the introduced clafer to its parent in the containment hierarchy. The $\underline{ref}$ map and the target indicate the target type of the reference. Note that mappings $\underline{ref}$ and $\underline{parent}$ are always single-valued. Among predicate declarations embodied into a clafer shape, all but multiplicity m..n is automatically assumed by default; multiplicity m..n is declared by the user (and is assumed to be 1..1 unless explicitly stated in concrete syntax to be otherwise).

4.1.3 Kinds of Clafers

There are two kinds of user-defined clafers: basic clafers and reference (bag and set) clafers. We describe them and specify their semantics via generic examples below. Table 4.1 summarizes the discussion and shows sample models and their LCSs. Although, the figures use concrete names, such as display and server, the compilation works analogically for any clafer of a given kind. Each of the clafers can be either abstract or concrete.

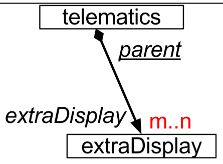
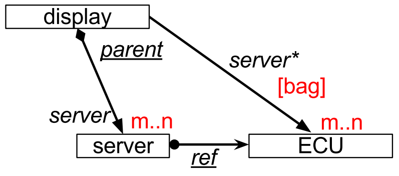
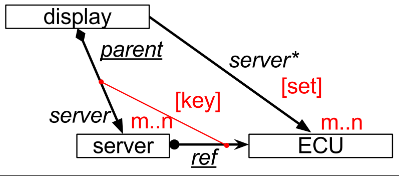
Clafer Kind	Clafer Model	LCS
Basic	telematics extraDisplay m..n	 telematics parent extraDisplay m..n extraDisplay
Reference bag	display server ↠ ECU m..n	 display parent server* [bag] server m..n m..n server ref ECU
Reference set	display server → ECU m..n	 display parent server* [set] [key] server m..n m..n server ref ECU

Table 4.1: The meaning of a clafer declaration

Basic Clafers

They establish containment hierarchy among clafers by nesting (same as composition in UML). An example is shown in the first row of Tab. 4.1. A distinction of a basic clafer's shape is that the maps *ref* and *target* and the target class are excluded.

Reference Bag Clafers

They correspond to bag-valued references, i.e., two or more references from the same source instance can point to the same target instance. The target clafer name follows the double arrow symbol (↠). For example, there may be several connections from a display to ECU. The second row of Tab. 4.1 illustrates a compilation of the reference bag clafer server to a corresponding LCS. In fact, reference bag clafers follow the structure of basic clafers, but additionally have the target class. In reference bag clafers the map playing the role of *target* is bag-valued, hence the annotation [bag] on *server**.

Reference Set Clafers

They are set-valued references and their name is followed by the arrow symbol ($\rightarrow$). They are similar to reference bag clafers, but the same source instance cannot point to the same target instance multiple times.

The compilation of reference bag and set clafers to LCSs is similar, but the latter have an additional predicate declaration [key]($\underline{parent}$, $\underline{ref}$) (see the last row of Tab. 4.1). The predicate means that each instance this of the head class is identified by a pair of instances (this.$parent$, this.$\underline{ref}$) from the source and target classes, and hence "rows" in the "table" server are not duplicated. Then navigation from the source to the target results in a set-valued mapping. In the example, for a given instance of display, each instance of server points to a different instance of ECU, and the mapping $server*$ is set-valued. Conversely, if the target mapping is set-valued, the pair of projections is a key.

Abstract Clafers

Abstract clafers define only a type (no direct instances). We distinguish nested and top-level abstract clafers. The LCS of the former is the same as of previously described clafers. Top-level abstract clafers, on the other hand, have slightly different LCSs: they have no parent in the containment hierarchy, i.e., the LCS excludes the head class and corresponding maps. When a top-level clafer is declared abstract, then all its descendants (in the containment hierarchy) are declared abstract by default. The descendants represent the case of nested abstract clafers.

Predefined Clafers

There are several clafers that are predefined in the language: clafer $\underline{\text{Sing}}$ and a family Dom of clafers representing primitive domains (e.g., int for integers, and string for strings of characters). $\underline{\text{Sing}}$ is very important although it does not appear in the concrete syntax. Each Clafer model by default has $\underline{\text{Sing}}$ as the root of the containment hierarchy, and hence $\underline{\text{Sing}}$ is the parent of top-level concrete clafers. It also is important in the context of top-level abstract clafers that have no parent. The synthetic root embodies the concept of existence. In Clafer, being reachable from the synthetic root is necessary for a clafer to exist when the model gets instantiated. $\underline{\text{Sing}}$ only has the head class, which is some predefined singleton class $\{\underline{*}\}$ also called $\underline{\text{Sing}}$ and has neither a parent nor a target (see the upper triangle in Fig. 4.5b). Any clafer in Dom is a child of $\underline{\text{Sing}}$, its head class is the class of the respective values (integers, strings, etc.), and the target class is absent.

4.1.4 Clafer Nesting

A Clafer model is a tree of clafer declarations. The concrete syntax expresses that hierarchy via indentation. For example, master is a child of plaECU in Fig. 4.5a. Clafers can arbitrarily nest clafers irrespective of their kind. In particular, reference clafers can nest clafers, which corresponds to nesting properties under UML association classes. In contrast to property nesting in UML, clafers can be nested arbitrarily deeply, however.

Clafer nesting is realized through cojoining LCSs that form an LMCS. If a clafer B is a child of (nested under) clafer A, then the head class of A is the source class of B, so that the B's LCS is plugged into A's LCS at its head class:

$$\mathsf{A.head} = \mathsf{B.source}$$

For example, Fig. 4.5b shows three cojoined LCSs where plaECU is a parent of master and Sing is a parent of plaECU.

Section 4.1.1 showed that a single LCS amounts to a *labeling mapping* from a UCS to a CD. Correspondingly, an LMCS amounts to a *labeling mapping* from a UMCS to a bigger CD. Figure 4.5c presents the extraction of this mapping for our example. The UMCS (left part of the figure) has no labels and follows the tree-like structure of the LMCS. The mapping itself is specified in Fig. 4.5c'. The names C1, C2, *M1*, *M2*, etc., represent unique class or map identifiers, respectively. The CD, generated by the mapping *label*, demonstrates that labeling glues together some nodes from the UMCS.

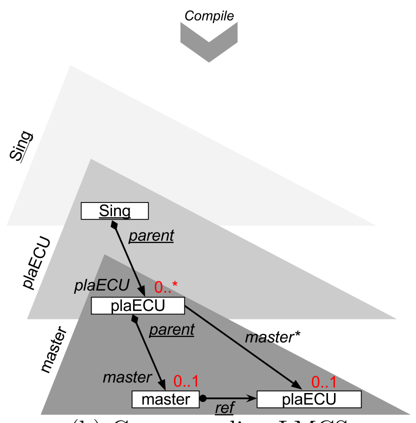
4.1.5 Inheritance

Inheritance between two clafers is defined at the level of their LCSs and means their inclusion: the corresponding classes in LCS are related by inclusions, as in Fig. 4.6. Inclusion maps are denoted by hollow-triangle heads, which resemble UML notation for inheritance. Formally, a map m with a hollow-triangle head means a predicate declaration [incl](m). The idea of modeling subsetting via inclusion maps is borrowed from category theory. Figure 4.7 shows an example. The clafer master under ECU2 specializes master under plaECU. Effectively, for reference clafers inheritance is a redefinition. The redefinition for maps also holds due to commutativity condition of maps: $super_s.head = head.super_h$. If an LCS has no target (is a basic clafer) or no parent (an abstract clafer), then there is no inclusion between missing classes.

Inheritance among any types of clafers is allowed, but is subject to the following restrictions:

```
plaECU 0..*
    master → plaECU 0..1
```

(a) Clafer model

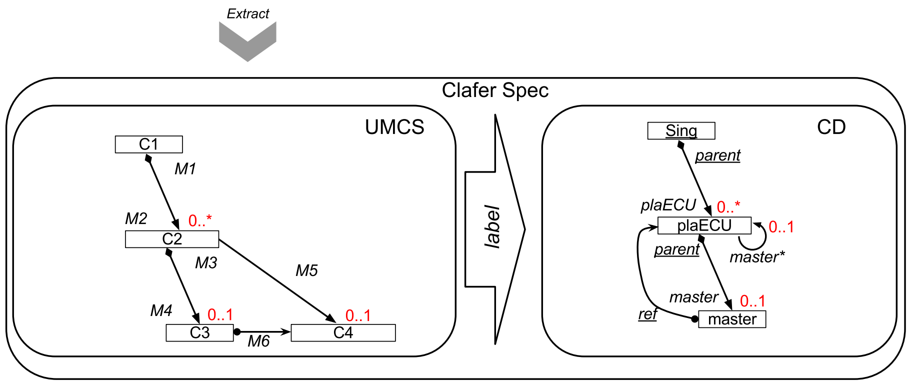


(b) Corresponding LMCS

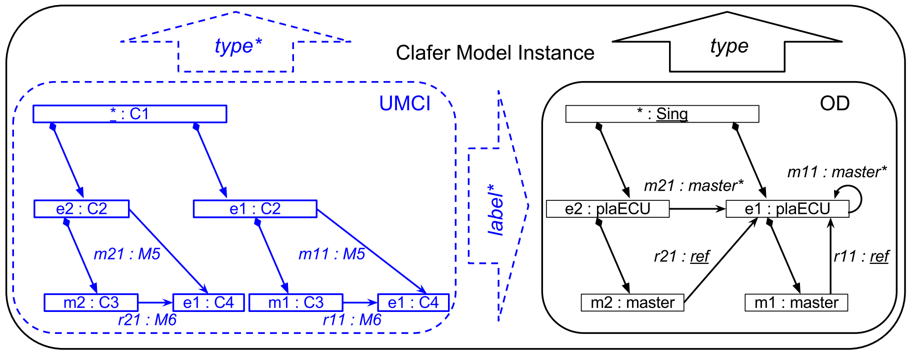
ClassRole	Label
C1	Sing
C2, C4	plaECU
C3	master

MapRole	Label
M1	*parent*
M2	*plaECU*
M3	*parent*
M4	*master*
M5	*master**
M6	*ref*

(c') Definition of mapping *label*



(c) Corresponding Clafer Spec containing the mapping *label* defined in (c')



(d) Sample Clafer model instance

Figure 4.5: Compilation of a Clafer model to LMCS, extraction of CD, and instantiation

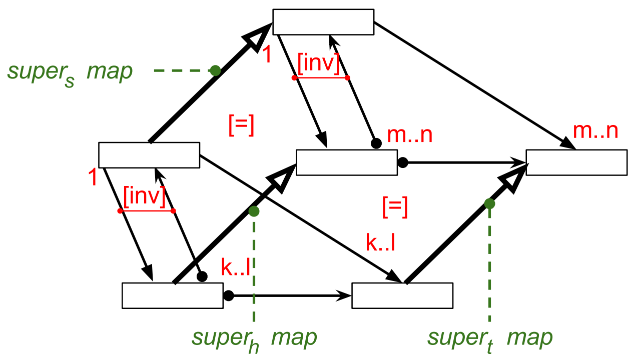


Figure 4.6: Inheritance among two UCSs

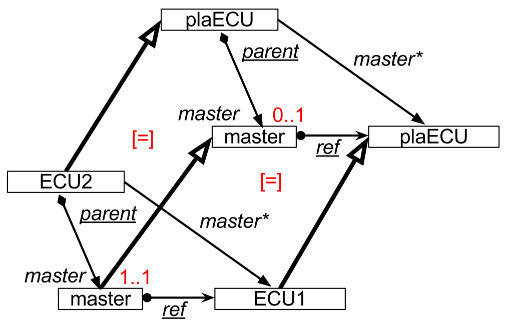


Figure 4.7: An example of clafer inheritance

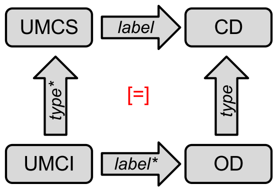
- a basic clafer cannot inherit from a reference clafer (because the subtype would remove the reference);
- a bag clafer cannot inherit from a set clafer (because the subtype would remove a constraint);
- if the super-clafer is not a top-level abstract clafer (i.e., it has a parent), then both the sub- and super-clafer must have the same parent in the containment hierarchy (because a clafer cannot have multiple parents).

When an abstract clafer A is a direct super type of clafers $\mathsf{B}_1, \ldots, \mathsf{B}_n$, then the constraint [cover] is declared in the LMCS on the $super_h$ maps relating head classes of $\mathsf{B}_1, \ldots, \mathsf{B}_n$ with their super classes. That way all instances of an abstract clafer must be instances of its concrete subtypes.

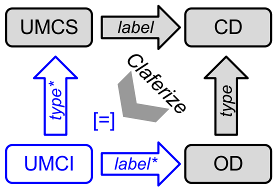
4.1.6 Instantiation

Many non-trivial model analyses (e.g., checking model consistency) can be reduced to the problem of finding a model instance by combinatorial solvers. Therefore, instantiation of Clafer models is the primary functionality of the Clafer toolchain. The toolchain works as follows. The Clafer compiler [7] transforms a Clafer model (Fig. 4.5a) to an Alloy [76] model that corresponds to a class diagram CD (the right part of Fig. 4.5c). Classes are translated to Alloy's signatures, maps to relations, and constraints to facts. Alloy uses a SAT solver to generate an instance of the CD, i.e., an object diagram OD typed over the CD. A sample instance is shown in the right part of Fig. 4.5d. Finally, *ClaferIG* [7] takes that OD and, using the links in the labeling mapping, represents (unfolds) it into a multi-clafer-shaped OD typed over the UMCS, as shown in the left part of Fig. 4.5d. We call such a multi-clafer-shaped OD an Unlabeled Multi-Clafer Instance (UMCI). Elements of a UMCI are object and link *roles*, which are labeled by real objects and links in the corresponding OD. Similarly to how the same class may play different roles in a UMCS, the same object can play different roles in a UMCI. For example, object e1 from the example in Fig. 4.5 plays three roles: an independent plaECU (shown by its position at the role box e1:C2), the master of object e2 (e1:C4), and the master of itself (e1:C4). Mapping *label** maps roles in a UMCI to the OD's objects and links that perform these roles.

Figure 4.8a gives a general definition of a UMCI. As a Clafer model is a hierarchical view on a class diagram, which is defined by a labeling mapping *label*: UMCS→CD, we consider an instance of a Clafer model to be a *similar* hierarchical view on an object diagram OD



(a) Definition



(b) Derivation

Figure 4.8: Clafer model instance

instantiating CD. That is, a *Clafer model instance* is a graph UMCI typed over the graph UMCS and labeled by elements of the OD (via mapping *label**: UMCI→OD) so that the square diagram in Fig. 4.8a commutes.

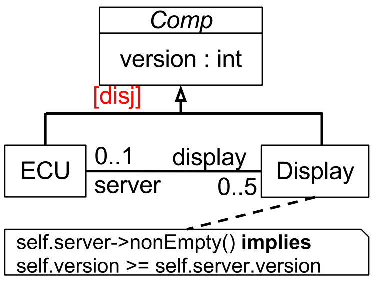
Moreover, for a given Clafer Spec *label*: UMCS→CD and any instance OD of the CD (shaded elements in Fig. 4.8b), it is possible to generate a correct UMCI and mappings *type** and *label** (blank elements with blue frames) by applying to them an operation ***Claferize*** (shown by a chevron). There is a unique such UMCI. In Sect. 4.2.3 we will precisely specify how this operation works.

4.2 Formal Semantics

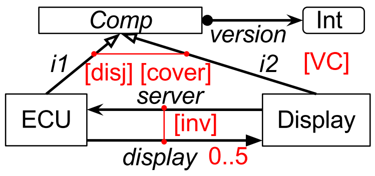
We will specify Clafer's syntax and semantics using formal class diagrams (CDs), a.k.a. DP-graphs, as our meta-meta-notation (which we have already briefly discussed in Sections 4.1.1, 4.1.2). That is, we will treat all models and meta-models involved as formal CDs, and mappings between them as structure-preserving mappings (morphisms) between formal CDs. (The adjective 'formal' will often be skipped.) Hence, we will begin with a precise description of CDs in the next section. Then we discuss a formalization of Clafer's syntactic mechanism (Sect. 4.2.2), and finally consider instantiation (Sect. 4.2.3).

4.2.1 Formal Class Diagrams and Their Instantiation

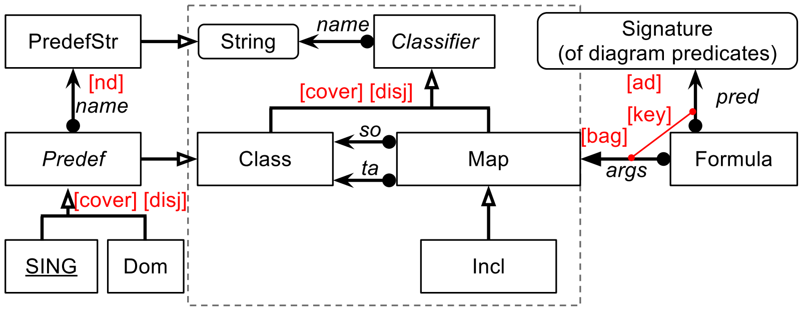
Figure 4.9a presents a simple UML class diagram D. An abstract class ***Comp*** has two disjoint subclasses (note the label [disj]), which are interrelated by a bidirectional association. In addition, version numbers of displays and their servers must satisfy a constraint [VC]: "the version number of the ECU serving a display must be not lower than the display's version number", which is written in the OCL format below the diagram. An abstract


Comp
version : int
[disj]
ECU
0..1
display
server
0..5
Display
self.server->nonEmpty() **implies**
self.version >= self.server.version

(a) UML CD, D



(b) Formal CD, F_D



(c) Meta-model of formal class diagrams ([nd] and [ad] are Name Discipline and Arity Discipline constraints resp.)

Figure 4.9: Formal Class Diagrams: an instance in UML (a), formal CD rendering (b), and the meta-model (c)

meaning of this diagram in terms of sets and mappings is that we have a set *Comp* partitioned into two disjoint subsets, which are interrelated by two mutually inverse mappings: *server* that maps displays to ECUs, and *display* that maps an ECU to the displays it serves. The attribute version can be also considered as a mapping that assigns an integer to each component. Finally, this configuration of sets and mappings must satisfy the constraint [VC].

This meaning is accurately specified by a formal diagram F_D in Fig. 4.9b. The latter is a directed graph encompassing four nodes and five arrows, which in addition carries several *predicate declarations* (constraints) shown in red square brackets. Thus, the diagram is a pair $F_D = (\Gamma[F_D], \Phi[F_D])$ with the first component being the carrier graph, and the second one being the set of constraints (we will also say *formulas*, hence, the symbol Φ), which will be formalized shortly. We call such formal diagrams *formal CDs*. We will first discuss the carrier graph and its instantiation, and then proceed to constraints. Next, we give a formal definition of formal CDs by specifying their meta-model. Formalization of the set-and-mapping semantics of CDs involves muny details, which we present in Appendix A.2.1. In the present section we will assume that the notions of a set and a (partial multi-valued) mapping between sets are intuitively understood; Appendix A.2.1 supports this intuition with a system of formal definitions.

Instantiation of Formal CDs, I: The Graph Structure.

Nodes in F_D are to be interpreted as sets: $[\![\mathit{Comp}]\!]$, $[\![\mathsf{Int}]\!]$, $[\![\mathsf{ECU}]\!]$ etc. We will often say "a component" for an element of set $[\![\mathit{Comp}]\!]$, "an ECU" for an element of $[\![\mathsf{ECU}]\!]$ etc. Arrows are to be interpreted by mappings (functions) between sets, which map elements from the source to sets of elements in the target. For example, an ECU is mapped to a set (perhaps, empty) of the displays it serves. We recall our convention about mappings that we have been using. If the mapping is defined for each element in the source (and results in a non-empty subset of the target), we call the mapping *total* and denote it by arrow with a bullet tail. If each element from the source, for which the mapping is defined, is mapped to a singleton, we say that the mapping is *single-valued* and denote it by arrow with an open-ended head. Thus, mapping *server* is partial single-valued, and *version* is total single-valued. In contrast, *display* is a general mapping, i.e., partial and multi-valued.

Very important and very special mappings are *inclusions*, which are denoted by arrows with a hollow triangle head — see arrows $i1$ and $i2$ in the diagram in Fig. 4.9b. An inclusion between two sets can be defined iff the source is a subset of the target; inclusion maps each element of the source to itself, but now considered as an element of the target. For example, inclusion $i1$ means that $[\![\mathsf{ECU}]\!] \subset [\![\mathit{Comp}]\!]$ and $i1(e) = e$ for all $e \in [\![\mathsf{ECU}]\!]$.

Thus, inclusion changes the role/type of ECU object e: object $i1(e)$ is a component with all its ECU-properties forgotten. In this way inclusions model inheritance. As there can be only one inclusion between a subset and its superset, we can name all inclusions by the same default name "isA" and omit it in concrete visualizations of formal CDs. Labels $i1$, $i2$ in Fig. 4.9b are IDs of the arrows rather than their names.

The discussion above can be summarized by saying that an instance of formal class diagram F_D is a *mega-mapping* $[\![..]\!] : \Gamma[F_D] \to \textsc{SetMap}$ from the carrier graph of F_D into a universe of sets and mappings, SetMap, also arranged as a directed graph: nodes are sets and arrows are mappings between sets. This mega-mapping preserves the graph structure (nodes are mapped to nodes and arrows to arrows so that their incidence is preserved), and respects mappings' properties: arrows with bullet tails are mapped to total mappings in SetMap, arrows with hollow-triangle heads are mapped to inclusion mappings in SetMap, etc.

A mega-mapping $[\![..]\!]$ is practically equivalent to a standard UML understanding of instantiation as having an object diagram typed over a class diagram. Indeed, sets $[\![\mathsf{ECU}]\!]$ etc. give us the objects, and mappings $[\![\mathit{server}]\!]$ etc. give us the links. If, for example, for an object $e \in [\![\mathsf{ECU}]\!]$, we have $[\![\mathit{display}]\!](e) = \{d1, .., dn\} \subset [\![\mathsf{Display}]\!]$, then in the object diagram we create n links from ECU e to displays $d1$, $d2$, $...dn$, all typed by mapping *display*. We will also have a link from $e \in [\![\mathsf{ECU}]\!]$ to $e \in [\![\mathsf{\mathit{Comp}}]\!]$ typed by $i1$, a link from $e \in [\![\mathsf{\mathit{Comp}}]\!]$ to an integer $[\![\mathit{version}]\!](e)$, and so on. In this way, a mega-mapping $[\![..]\!]$ gives rise to a directed graph $G_{[\![..]\!]}$ of objects and links, and a typing mapping $t_{[\![..]\!]} : G_{[\![..]\!]} \to \Gamma[F_D]$, which again respects the graph structure. Conversely, an object diagram O with object-link graph G_O and typing mapping $t_O \colon G_O \to \Gamma[F_D]$ gives rise to a mega-mapping $[\![..]\!]_O \colon \Gamma[F_D] \to \textsc{SetMap}$ by defining

$$\begin{aligned}
[\![\mathsf{ECU}]\!]_O &= \{n \text{ is a node (object) in } G_O : t_O(n) = \mathsf{ECU}\},\\
[\![\mathsf{Display}]\!]_O &= \{n \text{ is a node in } G_O : t_O(n) = \mathsf{Display}\},\\
[\![\mathit{server}]\!]_O &= \{a \text{ is an arrow (link) in } G_O : t_O(a) = \mathit{server}\}
\end{aligned}$$

and so on. An accurate formal definition of this construction, and a proof of equivalence of the two ways of instantiating formal class diagrams (via mega-mappings into SetMap and typing), are known in category theory under the name of the *Grothendieck construction* [15].

Instantiation of Formal CDs, II: The Constraints.

We have already discussed multiplicities—simple constraints assigned to single arrows. However, the diagram F_D also has four constraints shown in square brackets, which regulate instantiation of groups of arrows. Three of them, [disj,cover,inv], have a predefined meaning and their names are written in small font; the fourth, [VC], has a user-defined meaning specified by the OCL expression in Fig. 4.9a (links from the label [VC] to the four arrows, whose instantiation [VC] constrains, are not shown in the diagram to avoid line clutter).

In the abstract syntax, the four constraints encode the following expressions: [inv]$(server, display)$, [disj]$(i1,i2)$, [cover]$(i1,i2)$, and [VC]$(server, i1, i2, version)$ of the format $P(a_1, ...a_n)$ with P a predicate name and $a_1...a_n$ a list of arguments matching the predicate *arity*. The predicate [inv] can be declared only for two arrows between two classes going in the opposite directions, [cover] works for a group of arrows with a common target, and [disj] has the same arity. We can use these arities to check correctness of constraint declarations. Similarly, multiplicities are constraints for a single arrow, and diagram F_D actually declares several such constraints: [single-valued]$(server)$, [total]$(version)$, [0..5]$(display)$, etc., and also [incl]$(i1)$, [incl]$(i2)$.

In contrast, the arity of the user-defined predicate [VC] is given by the constraint declaration, and the arity condition is automatically true as soon as the expression is syntactically correct. In fact, any OCL, or another constraint language, expression written over a class diagram can be trivially considered as a respective diagram predicate declaration of the aforementioned format. Moreover, there exists a compact set of predefined diagram predicates, which allows one to express any FOL (and actually higher-order too) constraint as a composition of these predefined predicates [86]. This result may be useful for our future work on Clafer, but we do not need it here. The Clafer compiler treats a Clafer constraint expression as a property of the respective configuration of classes and mappings, like it is done in OCL.

Each predefined predicate has a certain semantics in terms of sets and mappings. For example, two mappings with a common target satisfy the predicate [cover] iff any element in the target belongs to the image of one of the mappings, or to both. The latter possibility is prohibited by predicate [disj]. Predicate [inv] holds iff the two mappings are mutually inverse. For example, for any ECU e and display d, $e \in [\![server]\!](d)$ iff $d \in [\![display]\!](e)$. Semantics of user-defined predicates is given by the user. Thus, a *legal* instance of diagram F_D is a mega-mapping $[\![..]\!]$ such that all constraints declared in $\Phi[F_D]$ are satisfied. Note that we have modeled abstractness of class *Comp* by requiring the two inclusions to be covering, i.e., stating that

$$[\![\mathit{Comp}]\!] = [\![\mathsf{ECU}]\!] \cup [\![\mathsf{Display}]\!]$$

and hence any component is either ECU or Display (but not both because of the [disj] declaration). In contrast to the UML class diagram in Fig. 4.9a, typing the name *Comp* in italic in the formal CD is a pure decoration without semantic meaning.

Meta-model

A meta-model of formal CDs is specified in Fig. 4.9c. It is itself a formal CD, M_{CD}, and any valid formal CD should have a valid instance $[\![..]\!]\colon \mathsf{M}_{\mathsf{CD}} \to \textsc{SetMap}$. The meaning of the central part (dashed-framed) of the formal CD is standard; we show how it works for our formal CD F_D in Fig. 4.9b. The latter is the following instance of the meta-model (we denote names of meta-classes in Small Capital font):

$$\begin{aligned}[\![\textsc{Class}]\!] &= \{\#\textsf{\textit{Comp}}, \#\textsf{Int}, \#\textsf{ECU}, \#\textsf{Display}\},\\ [\![\textsc{Map}]\!] &= \{\#\mathit{version}, \mathit{i1}, \mathit{i2}, \#\mathit{server}, \#\mathit{display}\},\end{aligned}$$

where $\#\textsf{xyz}$ denotes the ID of the classifier named $\textsf{xyz}$. This gives us the set $[\![\textsc{Classifier}]\!] = [\![\textsc{Class}]\!] \cup [\![\textsc{Map}]\!]$. Mapping $[\![\mathit{name}]\!]$ is defined as follows:

$$\begin{aligned}[\![\mathit{name}]\!](\#\textsf{ECU}) &= \text{“}\mathit{ECU}\text{”} \in [\![\textsc{String}]\!],\\ [\![\mathit{name}]\!](\#\mathit{display}) &= \text{“}\mathit{display}\text{”} \in [\![\textsc{String}]\!],\\ &\ldots,\\ [\![\mathit{name}]\!](i1) = [\![\mathit{name}]\!](i2) &= \text{“}\mathit{isA}\text{”} \in [\![\textsc{String}]\!],\end{aligned}$$

where $[\![\textsc{String}]\!]$ is the set of all possible strings, and string “isA” is the default name of inclusions.

Definition of mappings $[\![\mathit{so}]\!]$ and $[\![\mathit{ta}]\!]$ are also clear:

$$\begin{aligned}[\![\mathit{so}]\!](\#\mathit{display}) &= \#\textsf{ECU},\\ [\![\mathit{ta}]\!](\#\mathit{server}) &= \#\textsf{ECU},\end{aligned}$$

and so on. And $[\![\textsc{Incl}]\!]{=}\{\mathit{i1},\mathit{i2}\}$ so that $[\![\textsc{Incl}]\!] \subset [\![\textsc{Map}]\!]$ as required. It is also easy to check that mega-mapping $[\![..]\!]$ is a correct graph morphism, and all multiplicities are also respected.

Let us consider the meaning of the left part of the meta-model (to the left of the dashed frame). Meta-class $\underline{\text{SING}}$ is a singleton with a predefined (but optional) instantiation by a class $\underline{\textsf{Sing}}$, which, in turn, is instantiated (optionally) by a predefined object $\underline{*}$. That is, for any formal CD F instantiating M_{CD}, set $[\![\underline{\text{SING}}]\!]_F$ is (either empty or) the same fixed singleton class $\{\underline{\textsf{Sing}}\}$; for any object diagram O instantiating F, $[\![\underline{\textsf{Sing}}]\!]_O$ is (either empty or) the same fixed singleton $\{\underline{*}\}$. Meta-class $\underline{\text{SING}}$ is not instantiated in CD in Fig. 4.9b, but in formal CDs generated by the Clafer compiler, class $\underline{\textsf{Sing}}$ is always present as discussed in Sect. 4.1.3.

Similarly, meta-class $\textsc{Dom}$ is instantiated by (names of) predefined primitive-value domains like Int, String, or Bool, which, in turn, are instantiated by predefined sets of values. For example, for formal CD in Fig. 4.9b, $[\![\textsc{Dom}]\!]=\{\#\mathsf{Int}\}$, and instances of $\#\mathsf{Int}$ are predefined integer values. Thus, for a CD F, we have a set of predefined classes $[\![\textsc{Predef}]\!]_F$, which have predefined fixed names, say, $[\![name]\!](\#\mathsf{Int}) = \text{“}Int\text{”} \in [\![\textsc{PredefStr}]\!]$ and are common for all CDs. Constraint [nd] (read Name Discipline) requires that names of predefined classes be taken from predefined strings, and that names of user-defined classes be taken outside predefined strings.

Finally, the right part of the meta-model defines formulas. Meta-class $\textsc{Signature}$ is instantiated by predicates, which can be used in constraint declarations. For the diagram in Fig. 4.9b,

$$[\![\textsc{Signature}]\!] = \{\text{[cover]},\text{[disj]},\text{[inv]},\text{[incl]},\text{[set]},\text{[VC]}\}$$
$$\cup \{\text{[m..n]}{:}m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}\}$$

although not all multiplicities are used. Meta-class $\textsc{Formula}$ is instantiated by constraint formulas declared in the CD, and the constraint [key] states that a formula $\phi \in [\![\textsc{Formula}]\!]$ is uniquely determined by its predicate symbol $P = [\![pred]\!](\phi)$ and its list of arguments $(a_1, ..., a_n) = [\![args]\!](\phi)$ (we consider a list of formulas as a bag/family of formulas indexed by natural numbers, see Appendix A.2.1). That is, a formula is actually a pair $(P, (a_1...a_n))$, which we typically write as $P(a_1...a_n)$. For example, for the diagram in Fig. 4.9b, the set $[\![\textsc{Formula}]\!]$ is

$$\{\text{[disj]}(i1, i2), \text{[cover]}(i1, i2), \text{[inv]}(\#server, \#display),$$
$$\text{[incl]}(i1), \text{[incl]}(i2), \text{[0..5]}(\#display),$$
$$\text{[1..1]}(\#version), \text{[0..1]}(\#server)$$
$$\text{[VC]}(server, i1, i2, version)\}$$

plus three default declarations

$$\{\text{[set]}(\#version), \text{[set]}(\#server), \text{[set]}(\#display)\}.$$

Importantly, the list of arguments in the formula $P(a_1, .., a_n)$ must match the arity of predicate symbol P as it was discussed in Sect. 4.2.1. In detail, the mapping *args* is bag-valued, and the indexing set for a formula ϕ (Sect. A.2.1) is the list of the arrows of the arity graph of predicate $pred(\phi)$; a precise formal definition can be found in the Appendix. This condition is encoded by a meta-constraint [ad] (read Arity Discipline), which is a part of the meta-model like other meta-constraint declarations: [key]$(args, pred)$, [nd], etc.

Thus, legal instances of the meta-model Fig. 4.9c are *(formal) CDs* or, synonymously, *DP-graphs.*

4.2.2 Formalizing Clafer Syntax

Architecture of Clafer's syntactical mechanism

Figure 4.10 presents a schema of the mechanism, which refines the rough architecture discussed earlier in Fig. 4.1. There are three meta-models: Abstract Syntax Tree (AST) meta-model, Labeled Multi-Clafer Shape (LMCS) meta-model, and Class Diagram (CD) meta-model. They define three classes of syntactic artifacts that are important for Clafer: Clafer models, Labeled and Unlabeled Multi-Clafer Shapes (LMCSs and UMCSs), and (formal) Class Diagrams (CDs). The LMCS meta-model includes the AST meta-model and extends it with new properties of clafers; it is shown by the inclusion mapping i. The LMCS meta-model also includes the CD meta-model but names its elements differently; it is shown by the injection mapping j. We will describe the meta-models in Sections 4.2.2 and 4.2.2, respectively.

Formally, all nodes in the schema are CDs (i.e., DP-graphs), and horizontal arrows are mappings between them (DP-graph morphisms). The semantics of vertical arrows is different. They are graph morphisms typing elements in their source graphs by elements in the target graphs. In addition, these typing mappings must satisfy constraints declared in their target graphs (meta-models). We visualize this requirement by labeling the mapping with symbol $\models$.

The Clafer compiler takes a Clafer model that conforms to the AST meta-model, and extends it to an LMCS conforming to the LMCS meta-model. Formally, it can be seen as an operation shown in the schema by chevron labeled ***Compile***: the operation takes three CDs and two mappings (all shown shaded), and produces a new CD and two mappings (unfilled with blue contours).

Operation ***Extract*** provides Clafer Spec, i.e., a mapping from a hierarchically structured UMCS to a CD. In essence, ***Extract*** considers labels in LMCS as a definition of a mapping from UMCS to some formal CD encoded by the Clafer model, ignoring the AST structure.

We will discuss compilation and extraction in more detail in Sect. 4.2.2 and Sect. 4.2.2 resp.

Abstract Syntax Tree

The AST meta-model (see Fig. 4.11) specifies Clafer's abstract syntax. The AST corresponds to a grammar of Clafer models (Fig. B.2 in the Appendix). We discuss the meta-model starting from the left. Each clafer has a unique *name* (note the constraint [key]),

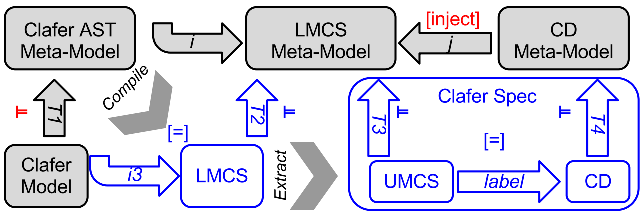


Figure 4.10: Clafer syntax mechanism.

some *multiplicity* and, optionally, a *super*-type. The class BASICCLAFER stands for a basic clafer declaration; REFCLAFER for reference clafer; and REFSETCLAFER for reference set clafer. The map *target* specifies the target of a reference clafer, and the map *abstract* indicates whether a clafer is abstract. The class DOM represents a family of primitive domain clafers (for example, clafer Int is a basic clafer whose parent is synthetic root); the singleton class $\underline{\text{SING}}$ represents the synthetic root clafer. The map *parent* establishes the containment hierarchy among CLAFERS. It is specified for each clafer besides the synthetic root and top-level abstract clafers: for any clafer C, if $C.parent = \bot$ and $C \neq \underline{\text{SING}}$, then $C.abstract = true$.

The class CONSTRAINT represents user-defined constraints in Clafer models. The map *context* points to the clafer, in which a constraint was declared. The map *scope* indicates a bag of clafers that each constraint refers to (besides the context clafer). For example, group cardinalities, such as the xor group cardinality in line 2 in Fig. 3.2, are simple constraints that relate a clafer with its children. There, the constraint c relates the clafer size with its children small and large; formally:

$$\begin{aligned} [\![context]\!](c) &= \mathsf{size}, \\ [\![scope]\!](c) &= \{\mathsf{small}, \mathsf{large}\}. \end{aligned}$$

The constraint language of user-defined constraints is specified in Appendix B.2. The language can be considered as a part of the meta-model.

Labeled Multi-Clafer Shape

The result of compilation is an LMCS. The meta-model in Fig. 4.12 defines the structure of an LMCS. As discussed in Sect. 4.1.2, in Fig. 4.4, each clafer actually declares a labeled clafer shape, i.e., a labeled graph of class and map roles. The three leftmost vertical arrows (*head*, *source*, and *target*) give three classes (i.e., class roles) that make up the shape of a clafer: the head class is always present, but the source (parent) and the target are optional

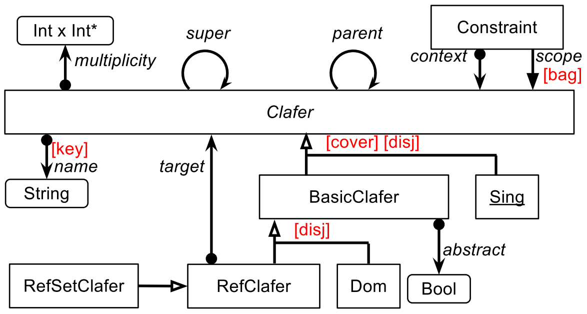


Figure 4.11: Clafer AST Meta-Model

depending on the kind of clafer. The next four vertical arrows in the middle (*head*, *parent*, *ref*, *target*) give four maps (all optional) that make up a clafer shape. The three rightmost vertical arrows give optional inclusion maps, if the clafer has a supertype. For example, the Clafer model in Fig. 4.5b amounts to the following instance of the meta-model:

$$\begin{aligned}
[\![\underline{\textsc{Sing}}]\!] &= \#\#\underline{\mathsf{Sing}},\\
[\![\textsc{Clafer}]\!] &= \{\#\#\underline{\mathsf{Sing}}, \#\#\mathsf{plaECU}, \#\#\mathsf{master}\},\\
[\![\mathit{parent}]\!](\#\#\mathsf{plaECU}) &= \#\#\underline{\mathsf{Sing}},\\
&\ldots\\
[\![\underline{\mathrm{SING}}]\!] &= \#\underline{\mathsf{Sing}},\\
[\![\mathit{source}]\!](\#\#\mathsf{plaECU}) &= \#\underline{\mathsf{Sing}},\\
[\![\mathit{head}]\!](\#\#\mathsf{plaECU}) &= \#\mathsf{plaECU}\\
&\ldots
\end{aligned}$$

where ##xyz refers to the clafer named xyz, and #xyz refers to the class role named (labeled by) xyz.

LMCS is augmented with diagram predicates over maps, which is represented by class FORMULA and is defined as for formal CDs. The predicates natively express predefined constraints, and also user-defined constraints. The latter are translated by the compiler into diagram predicates.

The meta-model as shown in Fig. 4.12 allows for any configuration of CLASSROLEs and MAPROLEs. They may form incorrect structures that are not LCSs. The meta-model is also underconstrained with respect to containment and inheritance hierarchies of clafers (LCSs may be cojoined incorrectly). Hence, we augment the metamodel with

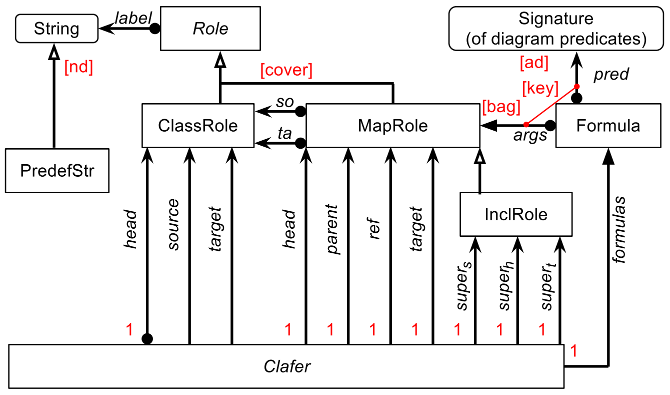


Figure 4.12: LMCS Meta-Model. It must satisfy the constraints from Tabs. B.1–B.4.

extra constraints to guarantee that its instances are valid LMCSs. There are four groups of constraints, all are given by conditional equations.

1) Incidence Equations define a correct graph structure of an LCS, i.e., the correct incidence of nodes and arrows. The constraints are specified in Tab. B.1 in the Appendix: first, a description is given, then its formalization follows. (To ease reading the formulas, we qualify map names in the meta-model from Fig. 4.12 by their targets.) For example, the first row requires that the map $head\_map$ in LCS has the source class as a source (so) and the head class as a target (ta).

2) Clafer Cojoining Equations specify the overlapping of LCSs. Clafers overlap iff there is parent-child relationship between them, or when a clafer is another clafer's target. Table B.3 lists the cojoining equations. For example, the first row requires that whenever one clafer is a parent of another clafer, then its head class plays the role of source class in child's LCS.

3) Clafer Kind/Shape Discipline Equations specify the structure of LCSs of different kinds of clafers. Table B.2 lists the clafer kind equations. For example, the first row specifies that the synthetic root clafer has neither source nor target classes, nor super-type.

4) Naming Discipline Equations specify the Clafer naming economy mechanism: given a clafer C, names of all elements in C's shape are derived from C's name, and can be used for navigation over the hierarchy. These constraints are also necessary for resolving targets of reference clafers and supertypes. Table B.4 presents the naming constraints. For example, the equation from the first row requires that the *head* map in a given LCS has the same label as the head class.

Compilation: From Clafer Model to Labeled Multi-Clafer Shape

A Clafer compiler consecutively traverses a Clafer model and builds a corresponding LMCS. It takes a Clafer declaration D, decodes into an Unlabeled Clafer Shape (UCS) $CS(D)$, and labels the elements of the latter using the name provided by D. After processing a declaration D, the compiler processes its children one by one. Decoding of each child D_i is regulated by the Cojoining Equations, so that shapes $CS(D)$ and $CS(D_i)$ are properly cojoined into LMCS. And so on until the entire Clafer model is traversed. Algebraically, compilation appears as an operation *Compile* described by the leftmost square in Fig. 4.10, as we discussed earlier.

Extracting: From Labeled Multi-Clafer Shape to Clafer Spec and Class Diagrams

A Clafer model provides: 1) a collection of classes and maps, and 2) a view on this collection, arranging classes into a hierarchy. The same class can play different roles in the hierarchy: being a parent of one class (source), a child of another class (head), or a reference class of another (target). Thus, elements of an LMCS are class and map *roles* rather than instantiatable (real) classes and maps. The latter are given by labels, and are instantiated by, respectively, objects and links. In contrast, roles are not instantiated, they impose a hierarchy on real classes and maps. This idea is captured by Clafer Spec. It considers labeling in an LMCS as a formal graph morphism, from the graph of class and map roles (UMCS) to the graph of real classes and maps (CD) formed by labels.

Importantly, the *labeling mapping* respects constraints: a predicate declaration in an LMCS is carried into the corresponding declaration in the CD. Thus, the *labeling mapping* is a formal CD morphism.

4.2.3 Formalizing Instantiation

Figure 4.8b presents a definition of Clafer instances and their main feature — derivability from the respective CD instances. Below we will see that derivability requires considering a

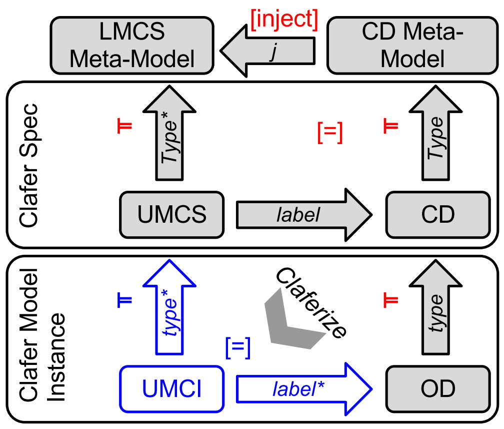


Figure 4.13: Architecture of Clafer instantiation

deeper schema that involves meta-models as shown in Fig. 4.13. The four upper models are formal CDs (DP-graphs), whereas the two bottom models are merely graphs (Clafer model instances, like ODs, do not carry constraints). All vertical arrows are typing mappings that respect the graph structure and satisfy the constraints (note the symbols $\models$). The right "column" of models conforms to the standard MOF architecture of instantiation; the left column is its counterpart for Clafer. All horizontal mappings are also graph morphisms; in addition, mappings *label* and j respect constraints (are DP-graphs morphisms). Chevron ***Claferize*** denotes an operation that completes a CD's instance, i.e., a pair (OD, *type*) to a Clafer instance (UMCI, *type**) (which is traced to the original instance via mapping *label**). Below we will often refer to instances by their source graphs, and say "instance OD" and "instance UMCI".

Let us consider an example showing how the operation ***Claferize*** works. Figure 4.15 elaborates the example from Fig. 4.5d by adding an intermediate preinstance UMCI_0, and slightly changes the notation to ease reading. Now we denote role classes and role maps by their class and map names, and shorten plaECU to ECU. Note that the class ECU plays two roles (formerly C2 and C4 in Fig. 4.5d): the source of the map *master* (ECUso) and the target of the map (ECUta).

As we have seen in Sect. 4.2.1, an instance of a CD can be seen as a mega-mapping $[\![..]\!]: CD \to \textsc{SetMap}$. In the example, we have the instance $[\![\underline{\text{Sing}}]\!]_{\text{OD}}=\{\underline{*}\}$, $[\![\text{ECU}]\!]_{\text{OD}}=\{\text{e1}, \text{e2}\}$, $[\![master^*]\!]_{\text{OD}}=\{m21, m11\}$, etc. Sequential mapping composition $label.[\![..]\!]_{\text{OD}}$ then yields an instance of LMCS. That is, for any element x of graph UMCS, we define $[\![x]\!]=[\![x.label]\!]_{\text{OD}}$, which gives us a mega-mapping $[\![..]\!]: UMCI \to \textsc{SetMap}$. The latter can be represented by a typed graph (due to the Grothendieck construction discussed at the end of Sect. 4.2.1), which we denote by UMCI_0 (read "Unlabeled Multi-Clafer Instance"$_0$, or *preinstance* for short). In other words, we set $[\![x]\!]_{\mathsf{UMCI}_0} = [\![x.\mathit{label}]\!]_{\mathsf{OD}} = x.\mathit{label}.[\![..]\!]_{\mathsf{OD}}$ for all elements x of graph UMCS.

Note that as the mapping *label* maps two different roles into one class ($[\![\mathsf{ECUso}]\!]_{\mathsf{UMCI}_0} = [\![\mathsf{ECUta}]\!]_{\mathsf{UMCI}_0} = \{\mathsf{e1},\ \mathsf{e2}\}$), each object ei, $i = 1, 2$ plays two roles of being potentially the source and the target of *master* links. Hence, in the carrier graph UMCI_0 we have two clones of e1 (e1:ECUso and e1:ECUta) and two clones of e2 (e2:ECUso and e2:ECUta). Other OD elements are not cloned because the mapping *label* only glues together classes ECUso and ECUta. Mapping $\mathit{label}^*_0 : UMCI_0 \to OD$ provides traceability of roles to objects, particularly, it glues together clones into their original objects.

Note that graph UMCI_0 is properly typed over graph UMCS. It also satisfies all multiplicities declared in UMCS because the mapping *label* carries these predicates into CD and the instance OD satisfies them.

Despite these good properties of the instance UMCI_0, it has two drawbacks. First, the clone e2:ECOta is not a master of any ECU, and does not actually fulfill its potential master role. Hence, its presence in the instance graph is unjustified. An example of this fact is that the node e2:ECUta is not reachable from the root node $\{\overset{*}{\_}\}$. Second, the graph UMCI_0 is not a correct multi-clafer shape: it has two different clafers (with heads m2:master and m1:master) with a common target node e1:ECUta, which is not allowed by the LMCS meta-model. Recall that the only two ways of cojoining two clafers are

$$\mathsf{this}.\mathit{source\_class} = \mathsf{this}.\mathit{parent\_clafer.head\_class}$$
$$\mathsf{this}.\mathit{target\_class} = \mathsf{this}.\mathit{target\_clafer.head\_class}.$$

To make the UMCI_0 a correct multi-clafer shape, the clone e1:ECUta must be cloned once again into $\mathsf{e1}_1$:ECUta and $\mathsf{e1}_2$:ECUta, as shown in the graph UMCI. Thus, the instance produced by mapping composition $\mathit{label}.[\![..]\!]_{\mathsf{OD}}$ needs some postprocessing to make it an actual multi-clafer shape: garbage removal to eliminate unreachable (unused) roles, and cloning to restore the correct clafer joining. Of course, each element in graph UMCI can be traced back to an element in the preinstance UMCI_0; this gives us a mapping *trace*. Mapping composition of *trace* and label^*_0 gives us a mapping $\mathit{label*}$ commuting with typing mappings.

An abstract schema of our work with the example is presented in Fig. 4.14. It shows three operations with models and mappings, ***Pullback, Recover***, and ***Compose***, consecutively performed. ***Pullback*** (i) converts an instance OD into a mapping $[\![..]\!]_{\mathsf{OD}} : CD \to \textsc{SetMap}$ by setting $[\![x]\!]_{\mathsf{OD}} = \mathit{type}^{-1}(x)$ for an element x in OD, then (ii) sequentially post-composes $[\![..]\!]_{\mathsf{OD}}$ with mapping *label*, and presents the result as a graph UMCI_0 (details can be found in

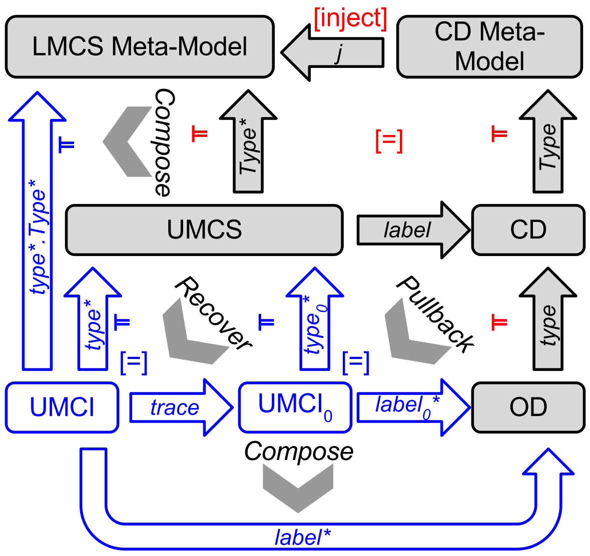


Figure 4.14: Clafer model instantiation

[15]). This operation is often read as "instance (OD, *type*) is pulled-back along the mapping *label*". It is proved in [55] that if the mapping *label* respects the constraints declared in LMCS and CD (which is always the case for Clafer compilation), and instance (OD, *type*) satisfies them, then the result of pulling it back also satisfies the constraints. Thus, the instance (UMCI_0, $type_0^*$) is correctly typed over LMCS, and satisfies all instance-regulating predicate constraints declared in UMCS.

However, it may happen that the corresponding deep instance (UMCI_0, $Type_0^*$) with typing

$Type_0^* = type_0^*.Type^*$: $\mathsf{UMCI_0} \to \mathsf{LMCSMetamodel}$

violates the constraints of clafer co-joining declared in the LMCS meta-model. Indeed, the mapping *label* cannot respect these constraints, because class diagram CD knows nothing about them. Hence, we need some operation to process the preinstance and fix its defects—this is done by our operation ***Recover***. It takes a preinstance (UMCI_0, $type_0^*$), removes objects unreachable from the root, and clones objects that glue clafers in an illegal way. The result is an instance UMCI, whose deep typing $type^*.Type^*$ conforms to the LMCS meta-model. This instance is traced back to the preinstance by mapping *trace*, whose composition with mapping $label_0^*$ gives us a traceability mapping $label^*$— a Clafer model instance is built.

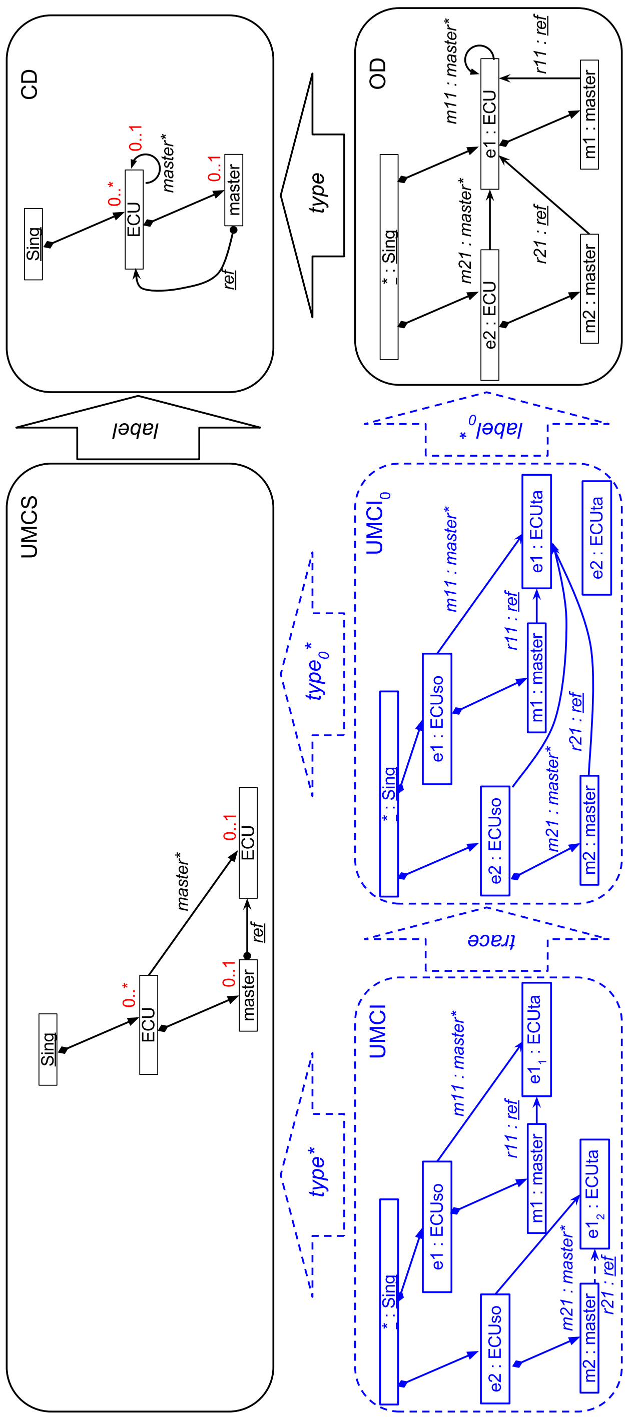


Figure 4.15: Clafer model instantiation: example

4.3 Concluding Remarks

In this chapter we precisely defined Clafer. Our formalization is done in a structure-explicit way. We have shown that the concept *clafer* indeed unifies classes, properties, and attributes. The main advantage of such a unification is the ability to arbitrarily nest properties, and to integrate inheritance into hierarchical modeling. These two key mechanisms allow Clafer to merge feature and class modeling in a single notation. Features are encoded as clafers with multiplicities 0..1 (for optional features) and 1..1 (for mandatory features). The use of singleton clafers is also important in our language for another reason. They allow to encode (partial) instances at the level of types, which we explain in the next chapter.

The work on formalization had a significant impact on Clafer design and implementation. The initial version of Clafer defined semantics of the language by translation to Alloy. Formalization of Clafer revealed several flaws in the language, helped to correct them, and introduced new language features. For example, initially,

- Clafer unified containment and reference clafers into a single kind of clafer which modeled inheritance and references via (multiple) inheritance. The work on semantics revealed that such a unification limited expressivity of the language. Later, it became clear that inheritance and references are separate mechanisms, but can be used simultaneously via hierarchical redefinition.
- Clafer did not permit hierarchical redefinition of clafers. It became possible only after precisely defining the concept of clafer.
- navigation over reference clafers was problematic due to potential name clashes. For example, if a clafer `C` had a child named `D` and the reference of `C` pointed to a clafer which also had a child named `D`, then there was no way of disambiguating the two cases. The work on formal semantics allowed us to clearly distinguish both cases in the abstract and concrete syntax.
- it was unclear whether clafers that point to types of primitive domains (e.g., integer) can nest other clafers in the containment hierarchy. Formalization of the concept of clafer showed that it is indeed possible.

Since we precisely defined Clafer, the toolchain development has been evolving simultaneously with the formal definition of the language.

Chapter 5

Partial Instances via Subclassing

The traditional notion of instantiation in OOM requires objects to be complete, i.e., be fully certain about their existence and attributes. In this chapter we explore the notion of *partial instantiation* of class diagrams, which allows the modeler to omit some details of objects depending on the modeler's intention. Partial instantiation allows modelers to express optional existence of some objects and slots (links) as well as uncertainty of values in some slots. We show that partial instantiation is useful and natural in domain modeling and requirements engineering. It is equally useful in architecture modeling with uncertainty (for design exploration) and with variability (for modeling software product lines).

Partial object diagrams can be (partially) completed by resolving (some of) the optional objects and replacing (some of) the unknown values with actual ones. Under the Closed World Assumption (CWA), completion reduces uncertainty of already existing objects, or deletes them if their existence is optional. Under the Open World Assumption (OWA), completion may additionally introduce new elements, perhaps uncertain. We present a simple theory of partial instantiation and completion under the CWA. We show that partial object diagrams can be modeled by subclassing and multiplicity constraints. As a result, class diagrams can implement partial instances with the well-known notions of subtyping and inheritance. Consequently, Clafer can use this encoding to specify models and (partial) instances in the same syntax.

5.1 Partial Instances and Object-Oriented Modeling

Instances play a major role in modeling. They represent real-world objects for which models provide abstractions. In OOM, an instance of a class diagram is an *object diagram*, i.e., a collection of objects and links instantiating, respectively, classes and associations. For a link $l\colon o \to v$, we will also say that object o owns *slot* l that holds value v.

Traditionally, objects are complete. Their types are known, and all slots have well-defined values. Such a notion of an instance, however, is restrictive when modeling involves uncertainty, variability, or simply underspecification. This is because classic (we will also say complete) instantiation requires that all slots are assigned values simultaneously. We discuss the notion of a *partial instance* that enriches the traditional instantiation. It allows object diagrams to have partiality, by which we assume that (a) existence of some objects and slots can be optional, and (b) there are slots with unknown values. By resolving optionality and replacing unknown values with actual ones, a partial object diagram becomes *complete*. There are many different completions of the same partial instance, and so the latter implicitly represents a set of instances. In this sense, a partial instance works like a class; in this dissertation we will make this observation precise.

Partial instances represent partial knowledge. They leave out knowledge that is unavailable at a given time, either due to uncertainty, variability, or underspecification. In uncertainty, the modeler captures several options but is unsure which one is the correct one (which one is correct is the missing knowledge). In variability, the modeler captures several options, each of which are correct and should be supported (the missing knowledge is the set of choices for a particular application). In underspecification, the modeler leaves out information that is irrelevant with respect to the modelers viewpoint. Thus, they differ in the intention. Partial instances, under various names, occur in:

- *Models with uncertainty.* Uncertainty captures possible choices that the modeler is unsure about ("don't know" semantics). An example would be a mobile device with hands-free input; this could be head gestures or voice input; the designer is uncertain about the choice, but the final solution will pick one of them. Partial instances of meta-models can represent uncertainty in models. They can treat uncertainty in requirements [28, 59] and in architectural models [60].

- *Models with variability.* Several choices are possible, each for a different product configuration (e.g., for a different customer). Partial instances of meta-models represent variability in models [46]. They are used to represent requirements models for product lines (including the product line scope), product line architectures (as demonstrated on the telematics product line example in Chapter 2), and product line tests. The variabilities in tests can be configured when the application is configured.

- *Models with underspecification.* Modelers focus on certain system aspects and can leave other aspects, which are outside their scope, underspecified ("don't care" semantics). Partial instances allow us to express partial specification of test cases as in Test-Driven Design (TDD) [79, 96].

- *Variability models* (e.g., feature models [82]). Instances of variability models represent system configurations; their partial instances represent partial configurations and support staged configuration [43], as shown in Chapter 3. Variability models are related to models with variability, but they do not consider further instantiations of the configurations (linguistically), because they are not meta-models.
- *Data with uncertainty.* Partial instances of data schemas represent uncertainty in application data. They are useful in databases [72], exchanging web data [14], and model finding [125].

The above applications of partial instances are difficult (if at all possible) to manage with complete instances. Partial instances allow one to delay design decisions and to construct instances incrementally. The missing parts of partial instances can be completed either by the modeler, or automatically by tools.

Despite the important applications, the traditional notion of instantiation in OOM offers limited support for partial instances. For example, UML object diagrams cannot express optionality of objects. One can use, however, UML class diagrams "as is" to encode partial instances. Our contribution makes this encoding precise and general. We show that partial object diagrams can be encoded by subclassing and strengthening multiplicity constraints. One of the implications is that **OOM languages with no direct support for partial instances can support them via class-based modeling.**

5.2 Requirements Elicitation with Partial Instances: An Example

Example-Driven Modeling (EDM) [28] systematically uses examples for eliciting, modeling, verifying, and validating complex business knowledge. During requirements elicitation a Subject Matter Expert (SME) transfers their knowledge to a Business Analyst (BA) who then explicates it as documents, models, and code. This section motivates the necessity of partial instances for eliciting and validating requirements, and in OOM in general. First, we consider partial instances under the CWA, where completion of partial instances means reduction of uncertainty, variability, or underspecification. Later, we discuss partial instances under the OWA, in which new objects and slots (perhaps, optional) can be added.

5.2.1 Completion under the Closed World Assumption

Alice is an SME and her organization needs a system for booking meeting rooms. She hires Charlie, a BA, to build such a system. Charlie's task is to implement room booking functionality. He is concerned with the timing aspect of scheduling meetings. Requirements elicitation is a complex task and, in practice, can only be done iteratively. The first session between Alice and Charlie goes as follows:

ALICE: We need to keep track of bookings to ensure that rooms and people are not double-booked. Recently, for example, Sue, the head of research, had scheduled two meetings at the same day at 10am.

CHARLIE: How did that happen?

ALICE: First, she organized a meeting at 10am. The other meeting was organized by Sam also at 10am. Sue somehow understood that Sam wanted to attend her meeting and confirmed her attendance of Sam's meeting. It wasn't the first time that miscommunication happened.

CHARLIE: I see. So how does Sue deal with conflicting meetings?

ALICE: In several ways. First, she may cancel one of the meetings. Alternatively, she confirms only one of the meetings while keeping the other one unconfirmed. She can also confirm the two meetings but they cannot overlap. Sometimes she combines the two meetings into one if the topics are similar. Each employee should have a daily agenda of meetings. Based on that they should be able to confirm or decline each meeting.

CHARLIE: That's quite complex. I think I understand...

[CHARLIE writes down the possible ways of scheduling meetings (Fig. 5.1).]

In Fig. 5.1, Conflict models the situation where Sue has two meetings scheduled at 10am. The object diagram shows her agenda with the meetings m1 and m2. It conforms to the class diagram in Fig. 5.1 CD, where time of the meeting is mandatory. The object diagram violates an important constraint that one person cannot have several meetings scheduled at the same time. To manage conflict resolution, Charlie creates a template for inserting information about the two meetings, in fact, a partial instance POD as shown in Fig. 5.1. The dashed arrow *part* indicates this activity. The initial partial instance must be as uncertain as possible. However, Sue cannot manage the time of Sam's meeting, hence, this attribute cannot be uncertain. By completing the partial instance incrementally, Charlie

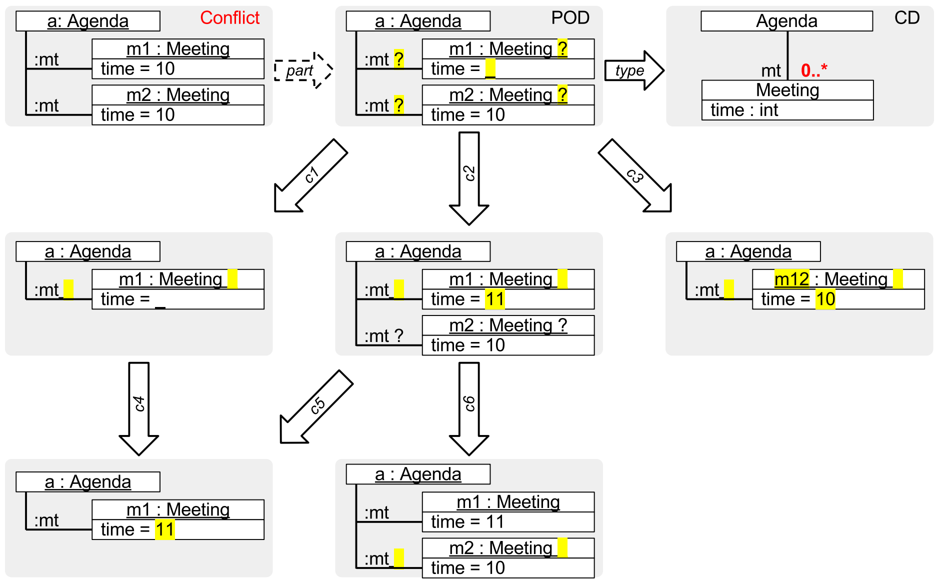


Figure 5.1: Several cases of completion of partial object diagrams. Changes between object diagrams are highlighted in yellow.

can arrive at a non-conflicting schedule. The partial instance conforms to the class diagram in Fig. 5.1 CD.

The partial instance POD has two types of partiality. First, the time of meeting m1 is unknown (has value _). As an organizer, Sue may pick the time later. The meaning of _ is that the concrete value exists but is unknown and it may be specified by a more complete instance. The second type of partiality is that the two meetings and corresponding slots are optional (labeled with ?). For example, it is unknown whether Sue confirms or declines the meeting m1 and/or m2. The meaning of ? is that an element may or may not exist.

Figure 5.1 depicts several cases of POD completion. The arrows *c1*. . . *c6* in Fig. 5.1 illustrate possible ways of scheduling meetings by Sue. They are partial or full completions of Fig. 5.1 POD. All these completions conform to the class diagram CD. Under the CWA, a partial instance is a (partial) completion of another partial instance if it removes some unknown value _ (by specifying the actual value) or label ? (by instantiating or deleting an element). A more complete instance can only reduce uncertainty, variability, or underspecification.

Completion *c1*. Sue cancels the meeting m2. Elements labeled with ? (m2 and its slot) have no instances in the more complete diagram. Additionally, she confirms the meeting m1, but may decide later when to schedule it. The object m1 and its slot are no longer labeled with ?. The slot time still has an unknown value, as Sue cannot pick the time unless she talks to her colleagues. The completion *c4* shows that Sue may decide to schedule the meeting at 11am. The more complete diagram replaces the unknown value _ with the actual value. The meeting is a fully complete instance without uncertainty, and is encoded as an object diagram.

COMPLETION *c2*. Sue confirms the meeting m1 and schedules it at 11am. In the partial object diagram, the label ? is removed from m1 and its slot. The value of time is specified as 11. Sue keeps the meeting m2 unconfirmed (still labeled as ?). The diagram can be further completed in two ways. First, Sue can cancel the meeting m2 as in the completion *c5*. Alternatively, as in the completion *c6*, she can confirm the meeting m2; its time does not overlap with m1. There may be several completion chains (e.g., *c1.c4* and *c2.c5*) leading to the same result.

COMPLETION *c3*. Sue decides to merge her meeting with Sam's one because the topics are similar. Formally, two objects are combined into one named m12 (we will also say that objects are glued together).

5.2.2 Completion under the Open World Assumption

Charlie works with his partner, Bob, to build the room booking system. The two BAs are interested in different aspects of the system. Charlie's task is to take care of scheduling; Bob needs to keep track of the available equipment. The session between Alice and Bob goes as follows:

ALICE: Each meeting is organized by a chair who is responsible for booking the room. The chair also notifies other participants about the meeting. Rooms have different equipment and, obviously, different numbers.

BOB: Let's understand a concrete meeting. Could you please give me an example of a room booking? What equipment is used?

ALICE: Sure. For example, Sue organizes meetings for her research group. They use an electronic whiteboard, as it simplifies sharing notes online.

[BOB writes down the example (see Fig. 5.2 Bob I).]

BOB: Perfect. Do all rooms have an electronic whiteboard?

ALICE: No. All rooms have a traditional whiteboard, but only some rooms offer the electronic one.

[BOB completes the example (see Fig. 5.2 Bob II).]

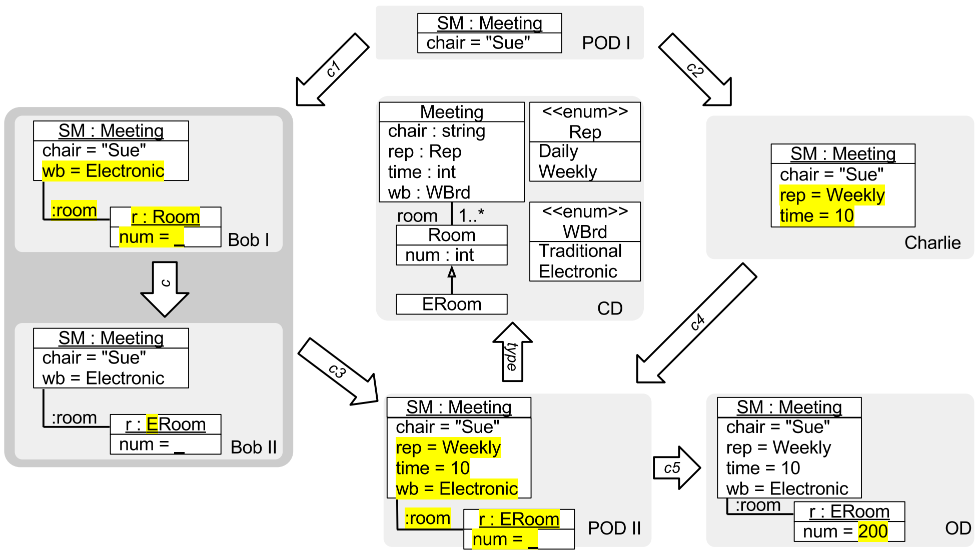


Figure 5.2: Abstraction and partial completion of examples. Changes between object diagrams are highlighted in yellow. Note that Bob II refines the type of room r.

In the next session Alice talks to Charlie again:

ALICE: As you may know, each meeting is organized by a chair.

BOB: Right, such as Sue. Alice, how often are the meetings scheduled? Can you give me a concrete example?

ALICE: For example, Sue organizes weekly meetings at 10am. They discuss progress done on research projects.

[CHARLIE writes down the example (see Fig. 5.2 Charlie).]

After the two sessions Bob and Charlie meet to consolidate their knowledge of different aspects of the system. Their goal is to come up with a consistent picture. Bob learned about rooms and equipment, whereas Charlie learned that meetings may repeat. Figure 5.2 shows the elicited examples and that the process of adding details can be modeled as instance completion.

Bob's first example (Fig. 5.2 Bob I) specifies that there is a meeting SM organized by Sue and that the meeting requires an Electronic whiteboard. He also specifies that the meeting takes place in some room r, but he does not know the room number num. After clarifying some details, Bob learned that only certain rooms provide the electronic equipment that Sue needs. He completes the previous example by refining the type of r to an assumed subtype ERoom (Fig. 5.2 Bob II). Charlie's example (Fig. 5.2 Charlie) shows that he learned that Sue schedules meetings at 10am and they repeat weekly.

Based on the partial examples Bob and Charlie create an example that merges their knowledge (Fig. 5.2 POD II). The partial object diagram is a combination of Charlie's example and Bob's refined example. Fortunately, there are no conflicts in the merged example. There is, however, still one unknown: the room number num where Sue meets her group. The two BAs propose a class diagram (Fig. 5.2 CD) that provides an abstraction for meetings. Abstractions generalize information to improve understanding of a set of examples. The BAs were able to construct the class diagram only after consolidating their partial knowledge.

Bob and Charlie decide to meet Alice again to validate the merged example and the proposed class diagram. Alice confirms that the example is valid. She also says that Sue uses room 200. Figure 5.2 OD shows a complete object diagram.

The completion in Fig. 5.2 works under the OWA. OWA allows completions to add new elements. For example, the completion POD II adds new elements to Bob's and Charlie's examples. Some slots do not exist in the examples by Bob (e.g., rep) or Charlie (e.g., wb). Also, Charlie's initial example had no uncertainty, but the partial instance POD II has uncertainty: the room number num is unknown. Clearly, completion based on OWA is more general than the one based on CWA.

Partial instances naturally express stakeholder's partial view of the world. When BAs focus on different aspects of the system, they construct partial examples. Modeling with partial instances has an important advantage over modeling with always complete instances. It explicates what is known and unknown given current knowledge. Our example showed that completion of partial examples may work under the CWA or the OWA. The former is useful for conflict resolution and exploring a set of configurations. The latter is adequate for requirements elicitation by various parties. OWA-completions subsume CWA-ones.

5.3 Modeling Partial Examples with Subclassing

This section shows that instantiation (partial and complete) of a class diagram can be encoded as extending the latter via subclassing. The main idea is that objects of class C are encoded as singleton subtypes of C; then links instantiating C's associations are naturally encoded as associations either inherited from C to the subclasses, or redefined in the subclasses.

5.3.1 Extension under the CWA

Figure 5.1 showed possible ways of resolving a conflict between two overlapping meetings. Let us now model all the solutions with subclassing as shown in Fig. 5.3. It parallels the structure of the previous figure. Instead of typing and completion, the diagrams are related by subclassing (arrows with hollow heads placed between class name and its superclass) and extension (hollow arrows between diagrams). Extension is a relation expressing that a more complete diagram includes the less complete one.

Figure 5.3 CD+ encodes Fig. 5.1 POD as a class diagram. The class diagram CD+ includes classes from Fig. 5.3 CD (the same as in Fig. 5.1 CD), but makes them abstract, and introduces subclasses. The class A is a singleton subclass of *Agenda*. Its class multiplicity is 1 (following class name and superclass), meaning that there is exactly one instance of this class. The two optional meetings are modeled by subclasses M1 and M2 with multiplicities 0..1. The two references from A to the meetings are also optional. As A is a subclass of *Agenda*, the two references redefine mt, i.e., they restrict the targets of mt to the two subclasses of *Meeting*. Both subclasses inherit the attribute time from *Meeting*. The class M1 says nothing about time and keeps its value unknown. The class M2 redefines the attribute time by specifying its value to be 10.

The extensions *e1* . . . *e6* parallel the completions *c1* . . . *c6* from Fig. 5.1. Informally, extension means that each element of the less complete diagram can be mapped to an element of the more complete one. Under the CWA the extensions reduce uncertainty. Class diagrams can do that by: introducing singleton subclasses, restricting multiplicities of classes/references/attributes, redefining targets of references, and assigning default values of attributes. All the extensions should include the class diagram from Fig. 5.3 CD, but with classes made abstract (similarly to CD+). We omit these classes to ease reading.

The extension *e1* models a situation when Sue confirms one of the meetings and cancels the other one. The multiplicity of M1 (and its slot) is redefined as 1. The multiplicity of M2 (and its slot) is redefined as 0. The diagram shows M2 to make it explicit that its multiplicity is 0. Removal of M2 from the diagram would have the same meaning. Furthermore, the value of time in M1 is kept unknown. The extension *e2* can be understood analogously. The extension *e3* describes a situation where Sue combines two meetings. It introduces a class M12 that merges information from classes M1 and M2 by subclassing them. Additionally, it refines class and slots multiplicities to be 1. In the case of diamond

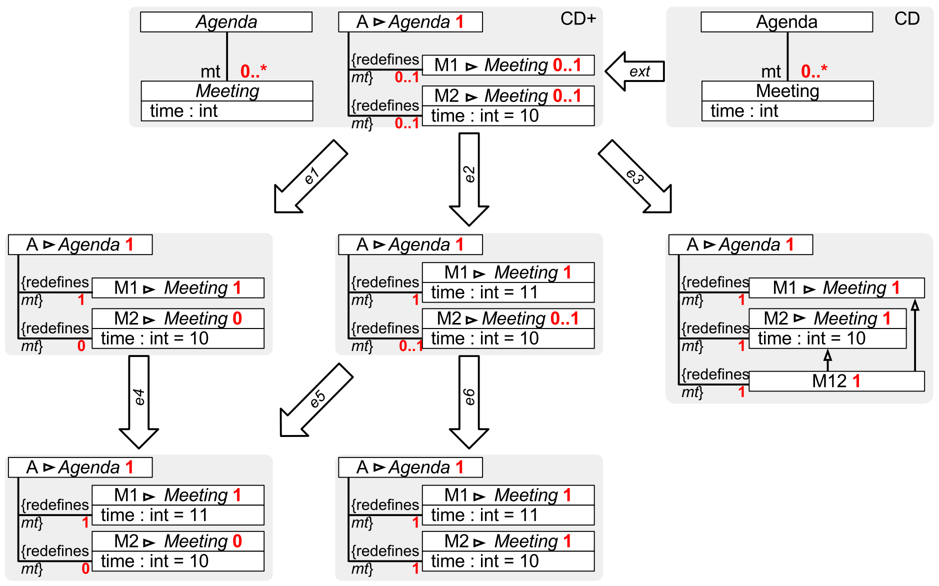


Figure 5.3: Several cases of extension of class diagrams (compare with Fig. 5.1)

inheritance, the properties from the common base are not duplicated. Thus M12 redefines the merge of the redefinitions of mt from M1 and M2.

5.3.2 Extension under the OWA

Bob & Charlie elicited examples of booking a meeting in Fig. 5.2. Figure 5.4 encodes the diagram with subclassing and extension. The class model in Fig. 5.4 CD is exactly the same as in Fig. 5.2. Other models are created as previously: objects are encoded as singletons, slots are encoded as redefined references/attributes, and each model includes classes from Fig. 5.4 CD but makes them abstract (omitted to avoid repetition). The mapping completion is replaced by extension.

Working under the OWA is natural when using subclassing and extension. For example, regardless of the definition of class Meeting, Charlie's class SM can easily add new attributes. They may have defined or undefined values. All the arrows *e1* ... *e6* could, in principle, be replaced by subclassing. The subclasses would need to be renamed to avoid name clashes.

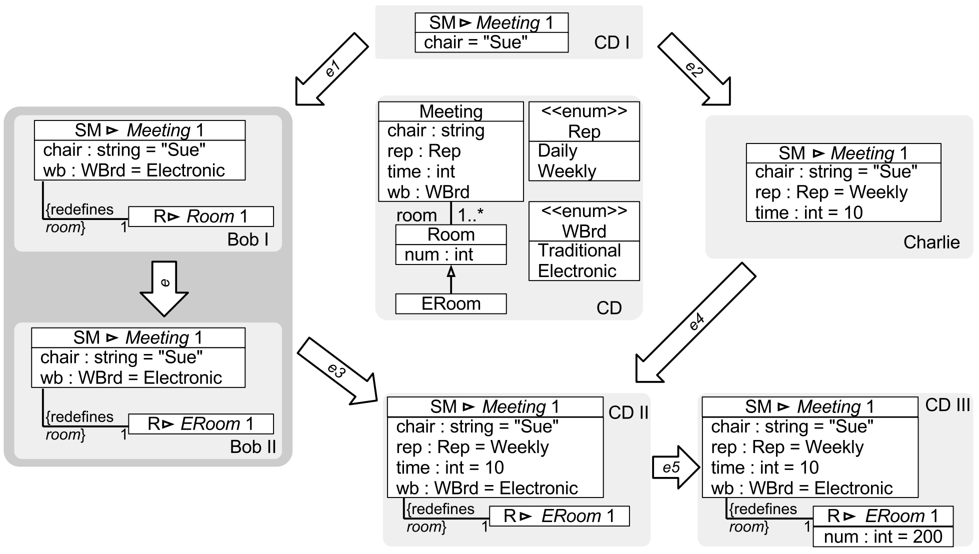


Figure 5.4: Partial examples as subclassing (compare with Fig. 5.2)

5.3.3 Encoding Partial Instances as Class Diagrams

We denote the encoding of partial object diagrams as class diagrams by a function *cdenc*. It takes a partial instance and encodes it as a class diagram that extends the class diagram that the partial instance conforms to. Figure 5.5 shows the previously defined class diagram from Fig. 5.1 CD and the partial instance from Fig. 5.1 POD that conforms to it. It also shows the completion *c6* of POD. All derived elements are shown as dashed and blue. The result of function *cdenc* is shown in the upper right corner of Fig. 5.5. The function *cdenc* takes POD, and extends CD with singleton classes (that encode objects) and references/attributes (that encode slots). It respects the labels **?** by placing multiplicities in the class diagram. If an attribute has undefined value, then it is skipped in the resulting class diagram, because it is inherited from one of the superclasses.

The derived class diagram (*cdenc*(CD, POD) in Fig. 5.5) has two important properties. First, it is an extension of CD that POD partially instantiates. Hence, the partial instance POD can be typed over the derived class diagram by *type+*. Second, all the completions of POD that are instances of CD must be isomorphic with instances of the derived class diagram. In the example, the completion *c6*(POD) is isomorphic with POD', i.e., an instance of *cdenc*(CD, POD). The partial object diagrams are not exactly the same due to different typing. The partial object diagram *c6*(POD) is typed over CD, whereas POD' is typed over *cdenc*(CD, POD). We formally show that the typing of *c6*(POD) over *cdenc*(CD,

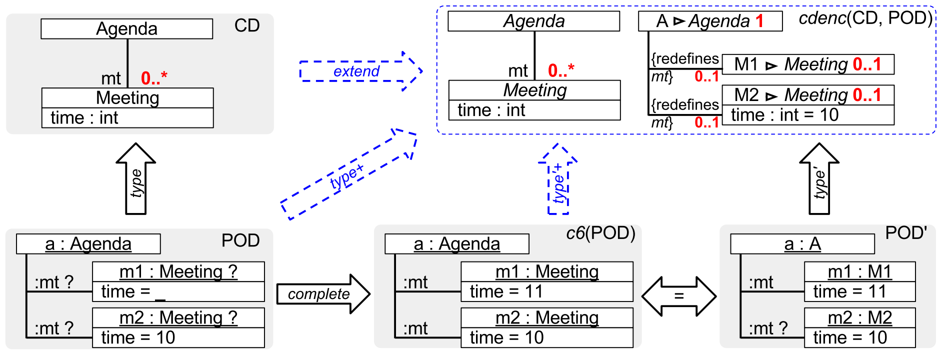


Figure 5.5: Example of partial instantiation via subclassing

POD) and the typing of POD' over CD can be derived under the CWA.

5.4 Partial Instantiation as Subclassing

This section formalizes class diagrams (CDs) and partial object diagrams (PODs) used in Sections 5.2 and 5.3 by building their meta-models. Although we already introduced formal CDs in Chapter 4, here we present a slightly modified version so that they more accurately fit in the context of partial instances. In particular, we extend the meta-model with explicit subclasses for attributes (with optional default values). We follow the same notation that is explained in Appendix A.1.

We also formalize the extension relations between CDs, and the completion relation between PODs, and prove a theorem stating that the latter can be encoded by the former (under the CWA for extension and completion).

5.4.1 Formal Class Diagrams and Their Extensions

The Meta-model: Classifiers.

Figure 5.6 specifies a meta-model of class diagrams. It is a graph whose nodes are meta-classes to be interpreted by sets; elements of those sets *instantiate* meta-classes. Node CLASS is instantiated by classes, for example, by Agenda, Meeting, int, string in Sample CD in Fig. 5.7; then we write $[\![\text{CLASS}]\!] = \{$Agenda, Meeting, int, string$\}$. Node REF is instantiated by references, for example, Sample CD instantiates REF by set $\{$person, mt, time$\}$.

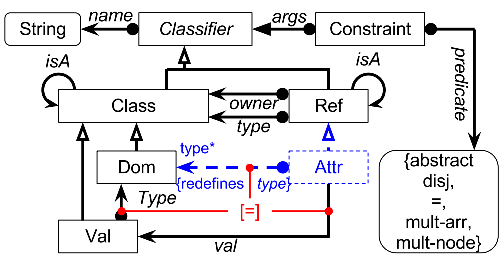


Figure 5.6: Meta-model of formal class diagrams

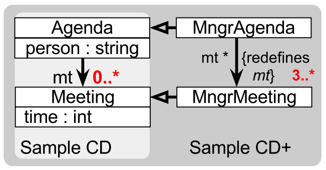


Figure 5.7: Sample instance: class diagram Sample CD and Sample CD+

Arrows in the meta-model are unidirectional meta-associations. Meta-associations are instantiated by sets of pairs of elements instantiating nodes; for example, for Sample CD, set $[\![$owner$]\!]$ consists of pairs (person, Agenda), (mt, Agenda), (time, Meeting).

The subclassing relation between classes is modeled by the meta-association *isA*. If this meta-association is instantiated — e.g., in the Sample CD+ , set $[\![isA]\!]$ has two elements (pairs of classes): (mngrAgenda, Agenda) and (mngrMeeting, Meeting) — it means that mngrAgenda is a subclass of Agenda, and mngrMeeting is a subclass of Meeting.

The keyword *redefines* means inclusion $[\![mt^*]\!] \subset [\![mt]\!]$ of the corresponding set of pairs. In such a case, UML says *mt* subsets mt*, and thus defines a meta-association loop *isA* for references too.

The *isA* (subset) mechanism is used in the meta-model itself (Fig. 5.6). The arrow from Dom to Class says that some of meta-classes are domains. For example, in Sample CD, $[\![\text{Dom}]\!]$={string, int} $\subset$ $[\![\text{Class}]\!]$ is the set of primitive domains used in the class diagram. Node Attr denotes the result of the query *"Select all references whose type is a domain"*; for Sample CD, $[\![\text{Attr}]\!]$={person, time}. We will say that it is a *derived* node (its frame is dashed and blue). The query also produces derived arrow *type**, which subsets (redefines) *type*.

As an attribute can be initialized with a concrete value (to be final in our context), the meta-model has a partially-defined meta-association *val*. Its target Val is instantiated by singleton classes that represent values of the primitive domains, $[\![\text{Val}]\!] = \{\{i\} : i \in [\![\text{int}]\!]\} \cup \{\{s\} : s \in [\![\text{string}]\!]\}$, and function $[\![Type]\!]$ provides their type: if $x \in [\![\text{Val}]\!]$ is in $[\![\text{int}]\!]$, then $[\![Type]\!](x)$=int. We require that for any object diagram instantiating the meta-model, and for any its attribute $a \in [\![\text{Attr}]\!]$, if a is initialized with a value, then the value has to be of the same type as the attribute, i.e., $a.[\![val]\!].[\![Type]\!] = a.[\![type^*]\!]$. We encode this constraint by labeling the three arrows with commutativity predicate [=].

Meta-model II: Constraints.

Constraints are an important part of formal class diagrams. Specification of constraints begins with a *signature Sign* of *predicate symbols (or labels)*, each one is supplied with its *arity*, i.e., a configuration (graph) of nodes and arrows for which the predicate can be declared (as explained in Sect. 4.2.1). In our examples, the signature is $Sign = \mathsf{mult\text{-}node} \sqcup \mathsf{mult\text{-}arr} \sqcup \{\mathsf{abstract}, \mathsf{disj}, =\}$. Set $\mathsf{mult\text{-}node}{=}\mathsf{int}{\times}\mathsf{int}^*$ consists of pairs of integers (including * for int^*), which can be declared for classes, i.e., the arity of each predicate in mult-node is some fixed single-node graph. Set $\mathsf{mult\text{-}arr}{=}\mathsf{int}{\times}\mathsf{int}^*$ consists of pairs of integers (including *), which can be declared for associations, i.e., the arity of each predicate in mult-arr is some fixed single-arrow graph (analogously to the signatures in Tab. A.2 in the Appendix). The arity of predicate abstract is also a singleton node graph. If a class is abstract, it can only be instantiated via its subclasses. In other words, there are no elements whose typing mapping points to the abstract class, but must point to one of the subclasses. UML's notation for declaring a class abstract is to display its name in *italic*.

Predicate = (commutativity) can be declared for any arrow diagram, in which there are two paths between the same source and target, like in the lower part of Fig. 5.6. The declaration ensures that for any element instantiating the source class, the two instantiated paths lead to the same element instantiating the target class. Note that having commutativity actually allows us to define subsetting (redefinition) of associations. For example, in Fig. 5.7 Sample CD+, declaring *mt* redefines mt* means commutativity: for any object diagram instantiating the diagram, and any object $a \in [\![\mathsf{MngrAgenda}]\!]$, we have $a.[\![mt^*]\!].[\![isA]\!] = a.[\![isA]\!].[\![mt]\!]$.

Extension Relation.

We first give a formal definition and then explain its meaning with special cases. Let CD be a consistent class diagram, i.e., $\textsc{Inst}(CD) \neq \varnothing$. (Note that an empty instance is legal if allowed by the constraints.) We say that a class diagram CD' *extends* CD (write $CD \leq CD'$), if

1. CD graph is a subgraph of CD' graph, particularly, they may coincide.
2. if a class A' belongs to $CD' - CD$, then
   (a) there exists a family of CD classes $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0, \mathsf{A}_1, ..., \mathsf{A}_n)$ with A_0 being the parent of $\mathsf{A}_1..\mathsf{A}_n$, which are all $(i = 0..n)$ declared abstract in CD$'$ and such

that A' is a child of all $\mathsf{A}_1..\mathsf{A}_n$ (and hence of A_0 too). The case n=0, hence, $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0)$, is not excluded.

   (b) if B' is another class (not equal to A') in $CD'-CD$ with $\mathsf{sup}(\mathsf{B'}) = (\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_m)$ and $\mathsf{B}_0 = \mathsf{A}_0$, then A' and B' are declared disjoint.

   (c) if a reference r' is owned by class A' in $CD' - CD$, then there is some A_i in the family $\mathsf{sup}(\mathsf{A}')$ such that r' is either inherited from A_i or redefines its reference r. In the latter case, if $\mathit{type}(r) = \mathsf{B}$ and $\mathit{type}(r') = \mathsf{B}'$, then B occurs into $\mathsf{sup}(\mathsf{B}')$.

3. all constraints in CD go into CD'. New constraints introduced in CD' must be consistent with constraints in CD so that CD' is also consistent.

Thus, $CD \leq CD'$ means that there is an embedding mapping $e : CD \to CD'$ satisfying the conditions above. There are several special cases of extension.

1. *Strengthening constraints.* CD is one class A with multiplicity 0..n and some attributes. CD' is composed of classes *A* and A', such that *A* is abstract and A' is a subclass of *A*, and the multiplicity of A' is $0 \leq m' \ldots n' \leq n$ with attributes inherited and/or redefined. Then because *A* is abstract in CD', CD' actually amounts to class A' with all its attributes inherited/redefined from A, that is, A' is A but with stronger multiplicity. For example, Fig. 5.3 shows that extension *e1* makes M1 a singleton.

2. *Deletion.* If in the first case the multiplicity is strengthened to be 0..0 for A', then the class A in CD will be effectively deleted. For example, Fig. 5.3 shows that extension *e1* deletes M2.

3. *Gluing.* CD consists of class A with two subclasses, A_1 and A_2, with multiplicities $\mathrm{m}_1..\mathrm{n}_1$ and $\mathrm{m}_2..\mathrm{n}_2$ respectively. CD' has in addition class A'_{12} subclassing both A_1 and A_2, which are declared abstract in CD', and its multiplicity is m'..n'. Because all (grand) parents of A'_{12} are abstract in CD', the latter, in fact, amounts to class $\mathsf{A'}_{12}$ with attributes inherited from A_1 and A_2. Thus, A_1 and A_2 have glued in CD' into $\mathsf{A'}_{12}$. For example, Fig. 5.3 shows that extension *e3* introduces M12 that subclasses M1 and M2. To prohibit extensions with gluing, it is enough to specialize the general definition by setting $n = 0$, i.e., $\mathsf{sup}(\mathsf{A}') = (\mathsf{A}_0)$: a class in the extension has exactly one superclass.

For a class diagram CD, we write $\textsc{Ext}(CD)$ for the set of all its extensions.

5.4.2 Partial Instances and Their Completion

Instantiation of Class Diagrams by Object Diagrams.

A class diagram is a pair $CD = (G_{CD}, C_{CD})$ with G_{CD} a graph with some additional structure specified in the previous section, and C_{CD} a set of constraints declared over the graph. An *object diagram* OD over CD is a graph G_{OD} equipped with a typing mapping $\mathit{type}_{OD} : G_{OD} \rightarrow G_{CD}$. Nodes in graph G_{OD} represent objects and values; arrows are links between them. As in UML, we also call links *slots*: for a link $\mathit{time} : \mathsf{M1} \rightarrow \mathsf{10}$, we say that object M1 owns slot *time* that holds value 10, and for a link $\mathit{room} : \mathsf{M1} \rightarrow \mathsf{R}$, we say that slot *room* holds a reference to object R. The typing mapping is a correct graph morphism compatible with partition into classes and domains. For example, if a node in G_{OD} is typed by int, then it must be an integer value.

We call an OD correctly typed over a CD's *preinstance*, and write $\mathrm{PI}\textsc{nst}(CD)$ for the set of CD's preinstances.

Inverting the typing mapping maps nodes of graph G_{CD} into sets, and arrows into mappings. For example, if C is a class in G_{CD}, then $\mathit{type}^{-1}_{OD}(\mathsf{C})$ is the set of objects typed by C. In Sect. 5.4.1 we denoted such sets by $[\![\mathsf{C}]\!]$. Similarly, if $r : \mathsf{C} \rightarrow \mathsf{C'}$ is a reference arrow in G_{OD}, then $\mathit{type}^{-1}_{OD}(r)$ is the set of links (i.e., pairs of objects) typed by r. In Sect. 5.4.1, we denoted such sets by $[\![r]\!]$, and noted that such a set defines a mapping $[\![r]\!] : [\![\mathsf{C}]\!] \rightarrow [\![\mathsf{C'}]\!]$. Hence, we can check whether multiplicities and other constraints declared in CD are satisfied.

We say that an OD over CD is its *correct (or legal) instance* if all constraints are satisfied. Let $\textsc{Inst}(CD)$ denote the set of all legal CD's instances. Clearly, $\textsc{Inst}(CD) \subset \mathrm{PI}\textsc{nst}(CD)$.

Instantiation of Class Diagrams by Partial Object Diagrams.

A *partial* object diagram is an object diagram, where some values in slots may be unknown, and some objects and slots may not exist (our examples marked such by label ?). To deal with unknown values, we add to every primitive domain a countable set of null values $\{\_{1}, \_{2}, \ldots\}$ called *indexed nulls.* (In the database literature, they are called labeled nulls.) For a given domain, say, int, we need many nulls (not just one), because different attributes of type integer may have (potentially different) unknown values. Making attributes certain means replacing nulls by actual (non-null) integer values, but having only one null value would force us to make all values equal. In our examples, we placed symbol _ into a slot

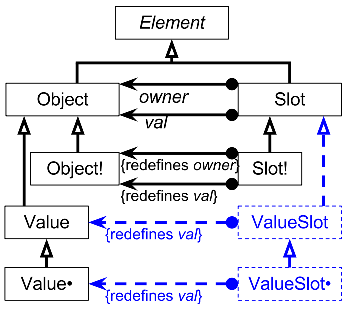


Figure 5.8: Meta-model of partial object diagrams

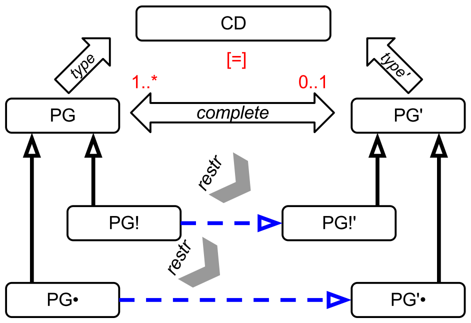


Figure 5.9: Rules of instance completion

with unknown value, but we assume that different slots (of the same type) hold different indexed nulls.

If existence of an object or slot is declared uncertain, we label it by **?** and say it is *optional*. Otherwise, an object or slot is considered certain and *mandatory*. If in concrete syntax slots belong to an optional object, then they are optional themselves. A mandatory object may have optional slots, but if a slot is mandatory (in the semantics), its owner is mandatory too (but the value may be unknown). Moreover, to avoid dangling references, a mandatory slot holding a reference must refer to a mandatory object. We admit optional slots with known values (for example, optional meeting M2 with certain time in Fig. 5.1).

The Meta-model.

Meta-model in Fig. 5.8 makes the discussion precise. The upper part (ELEMENT, OBJECT, SLOT) says that a partial object diagram is a graph. Meta-classes OBJECT! and SLOT! represent mandatory objects and slots; mandatory elements form a correct subgraph of the partial object diagram graph.

Metaclass VALUE represent values of primitive domains (e.g., integers and strings) together with the indexed nulls. For simplicity, values are assumed to be special objects (class VALUE is a subclass of OBJECT). Class VALUE$^\bullet$ represents actual values of primitive domains (nulls excluded). Derived class VALUESLOT is for slots holding values rather than references, and VALUESLOT$^\bullet$ is subclass of slots holding actual known values.

To be precise, instances of the meta-model in Fig. 5.8 are *partial graphs* rather than partial object diagrams: the latter are endowed with typing mapping into some class diagram. The meta-model states that a partial graph is a triple $PG = (G, G!, G^\bullet)$ with G

a graph, $G!$ its subgraph of *mandatory* elements, and $G^\bullet$ a subgraph of slots with *known values*.

Given a class diagram CD, a *partial object diagram* over it, POD, is a partial graph $PG_{POD} = (G_{POD}, G!_{POD}, G^\bullet{}_{POD})$ with a totally defined typing mapping (graph morphism) $type_{POD} : G_{POD} \to G_{CD}$, which maps proper objects to classes and values to value domains. The pair $(G_{POD}, type_{POD})$ is denoted by $|POD|$; it is the POD with all ?-labels removed.

Given a CD, we say that a POD is a *(partial) preinstance* if $type_{POD}$ is a correct graph morphism (thus, the set $\mathrm{PI}\textsc{nst}(CD)$ also includes well-typed graphs with unknown values). We call a preinstance POD an *(partial) instance* if $G_{POD} = G^\bullet{}_{POD}$ (i.e., all values are known) and all constraints are satisfied, i.e., $|POD| \in \textsc{Inst}(CD)$. We denote the set of (partial) preinstances by $\mathrm{pPI}\textsc{nst}(CD)$ and of (partial) instances by $\textsc{pInst}(CD)$.

Partial Object Diagram Completion.

Let $PG = (G, G!, G^\bullet)$ be a partial graph. Its *(partial) completion* comprises another partial graph $PG' = (G', G'!, G'^\bullet)$ and a partially defined graph mapping $c : G \to G'$, which is compatible with the extra partial graph structure. To wit: both restrictions of mapping c to the two subgraphs, $c! : G! \to G'$ and $c^\bullet : G^\bullet \to G'$, are actually inclusion mappings into the respective subgraphs of G', i.e., mapping c provides two inclusions $c! : G! \to G'!$ and $c^\bullet : G^\bullet \to G'^\bullet$ as shown in Fig. 5.9 (and so $G! \subset G'!$ and $G^\bullet \subset G'^\bullet$). Completion of partial object diagrams, i.e., typed partial graphs, requires, in addition, commutativity with typing mappings as shown in the upper part of the figure.

Let us see how this definition works. Given a CD, we say that a partial object diagram POD' is more complete than partial object diagram POD, if some unknown values _ in POD are replaced by actual values, and some of labels ? are removed by either removal of labels ? from objects/slots, or removal of objects/slots labeled by ?. The former removal means that an ?-element in POD certainly exists in POD', the latter removal means that a ?-element certainly does not exist in POD'. The multiplicities on the *complete* arrow in Fig. 5.9 are important. The multiplicity 0..1 means that an element of POD may have only one completion in POD'. The multiplicity 1..* means that a completion completes at least one element, i.e., it can reduce uncertainty by gluing elements (if the multiplicity was 1, gluing would be prohibited). Generally, we have a partially defined mapping $c : POD \to POD'$ commuting with typing of POD and POD'. We call this mapping *complete* (see Fig. 5.9), and write $c : POD \leq POD'$.

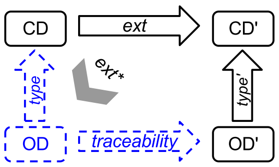


Figure 5.10: Projection of preinstances

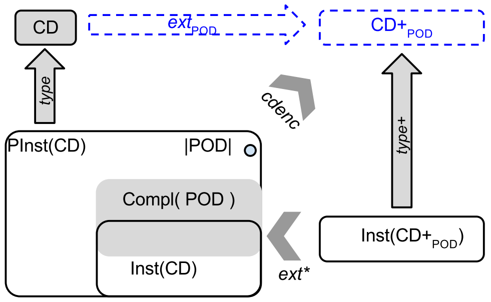


Figure 5.11: Instances of CD^+ are instances of CD and completions of POD

We write $\textsc{Compl}(POD) = \{|POD'| \in \textrm{P}\textsc{Inst}(CD) : POD \leq POD'\}$ for the set of all completions of POD.

5.4.3 Partial Object Diagrams via Class Diagrams

We first note that an extension $ext : CD \to CD'$ of diagram CD gives rise to a function $ext^* : \textrm{P}\textsc{Inst}(CD') \to \textrm{P}\textsc{Inst}(CD)$ that projects preinstances of CD' to preinstances of CD (see Fig. 5.10). Let OD' be a preinstance of CD', e' is its element, and $t' = type'(e)$ is its type in CD'. If $t' = ext(t)$ for some type $t \in CD$, then ext^* copies e into OD and gives it the type t. If $t' \in (CD' \setminus CD)$, then e is not copied into OD. In this way, by traversing all elements in OD', we build a CD's preinstance OD and traceability mappings from OD to OD'.

Theorem 1 *For any consistent class diagram CD and its partial preinstance POD there is a class diagram CD^+_{POD} and an extension $ext_{POD} : CD \to CD^+_{POD}$ such that the mapping*

$$ext^* : \textsc{Inst}(CD^+_{POD}) \to \textsc{Inst}(CD) \cap \textsc{Compl}(POD)$$

is a bijection. Moreover, if $POD \neq POD'$, then $CD^+_{POD} \neq CD^+_{POD'}$.

Figure 5.11 visualizes the theorem. Any correct instance of the class diagram CD^+_{POD}, projected onto preinstances of CD, is a correct instance of CD and is a completion of POD. All completions of POD, that are correct instances of CD, must also be correct instances of CD^+_{POD}. Note that there are completions of POD that are not correct instances of CD (they may violate its constraints).

We prove the theorem for the simpler case of completion without gluing, and correspondingly CD^+_{POD} without multiple inheritance.

Proof. The proof consists of two parts. In Part 1, we specify a function *cdenc*, which for a given pair (CD, POD), as above, produces CD^+_{POD} and an extension mapping $\mathit{ext}_{POD} \colon CD \to CD^+_{POD}$. In Part 2, we prove that ext^* is a bijection.

Part 1. (Below we will skip the index POD near CD^+ and *ext*)

Function *cdenc* encodes any partial object diagram POD as a class diagram CD^+, such that CD^+ is an extension of CD $(CD \le CD^+)$. For a given class diagram CD, any partial object diagram POD, such that $|POD| \in \text{PINST}(CD)$, the function $\mathit{cdenc}(CD, POD)$ constructs $CD^+ \in \text{EXT}(CD)$ as follows.

1. Copy all elements of CD to CD^+.
2. Label all classes of CD^+ that belong to CD as [abstract].
3. For each $o \in \text{OBJECT}$ belonging to POD, create a singleton class $c \in \text{CLASS}$ belonging to CD^+. The class subclasses o's class, i.e., $\mathit{isA}(c) = \mathit{type}(o)$. If $o \in \text{OBJECT}!$ then the multiplicity of c is $1..1$, otherwise it is $0..1$.
4. For each $s \in \text{SLOT}$ where $\mathit{owner}(s) = o$ and $\mathit{val}(s) = v$, such that $v \neq \_$, create a reference $r \in \text{REF}$ belonging to CD^+. Let us assume that the objects o and v are mapped to classes c and d, respectively, in CD^+. The reference r is defined so that $\mathit{owner}(r) = c$. Additionally, the reference redefines its type from CD, i.e., $\mathit{isA}(r) = \mathit{type}(s)$. If $s \in \text{VALUESLOT}$, then $\mathit{type}(r) = \mathit{Type}(\mathit{type}(v))$ and $\mathit{val}(r) = d$, otherwise $\mathit{type}(r) = d$. In the former case, the type of r is one of the primitive domains. If $s \in \text{SLOT}! \cup \text{VALUESLOT}^\bullet$ then the multiplicity of r is $1..1$, otherwise it is $0..1$.

Part 2. For the function *cdenc*, as defined above, the mapping ext^* defined at the very beginning of Sect. 5.4.3 is a bijection.

2.1) *Given a correct instance I in $\text{INST}(CD) \cap \text{COMPL}(POD)$, there is I^+ in $\text{INST}(CD^+)$ such that $\mathit{ext}^*(I^+) = I$.*

The partial graph of POD can be typed over CD^+, because of encoding by *cdenc*. Each element e_{POD} of POD can be typed over CD^+ by $\mathit{type}_{POD} : GP_{POD} \to CD^+$. If I completes POD, then for each element e belonging to I, we have $e = \mathit{complete}(e_{POD})$. The instance I^+ can be constructed by having the same partial graph as I and typing each e of I over CD^+ by $\mathit{type}^+(e) = \mathit{type}_{POD}(e_{POD})$. The instance I^+ is correct, as CD^+ preserves the constraints of CD and POD.

Furthermore, $\mathit{ext}^*(I^+) = I$ holds. The instance $\mathit{ext}^*(I^+)$ is a correct instance of CD, because extension is compatible with constraints. That is, we also have a function

$ext^* : \textsc{Inst}(CD^+) \to \textsc{Inst}(CD)$ (denoted again by ext^*). That way each correct instance of CD^+ can be projected onto a correct instance of CD.

2.2) *Given a correct instance $I^+ \in \textsc{Inst}(CD^+)$, the projection $ext^*(I)^+$ is in $\textsc{Inst}(CD) \cap \textsc{Compl}(POD)$.*

As shown previously, any correct instance I^+ of CD^+ can be projected onto a correct instance of CD, i.e., $ext^*(I^+) \in \textsc{Inst}(CD)$.

Furthermore, I^+ also belongs to $\textsc{Compl}(POD)$. It is because, POD can be typed over CD^+. Each element belonging to I^+ has exactly one type t^+ such that exactly one element of POD is mapped to t (it is established by *cdenc*). This correspondence establishes completion between elements of I^+ and $|POD|$, and from that follows that $ext^*(I^+) \in \textsc{Compl}(POD)$.

As it is seen from the proof, the constructions 2.1 and 2.2 are mutually inverse. The last statement of the theorem is also evident by construction. □

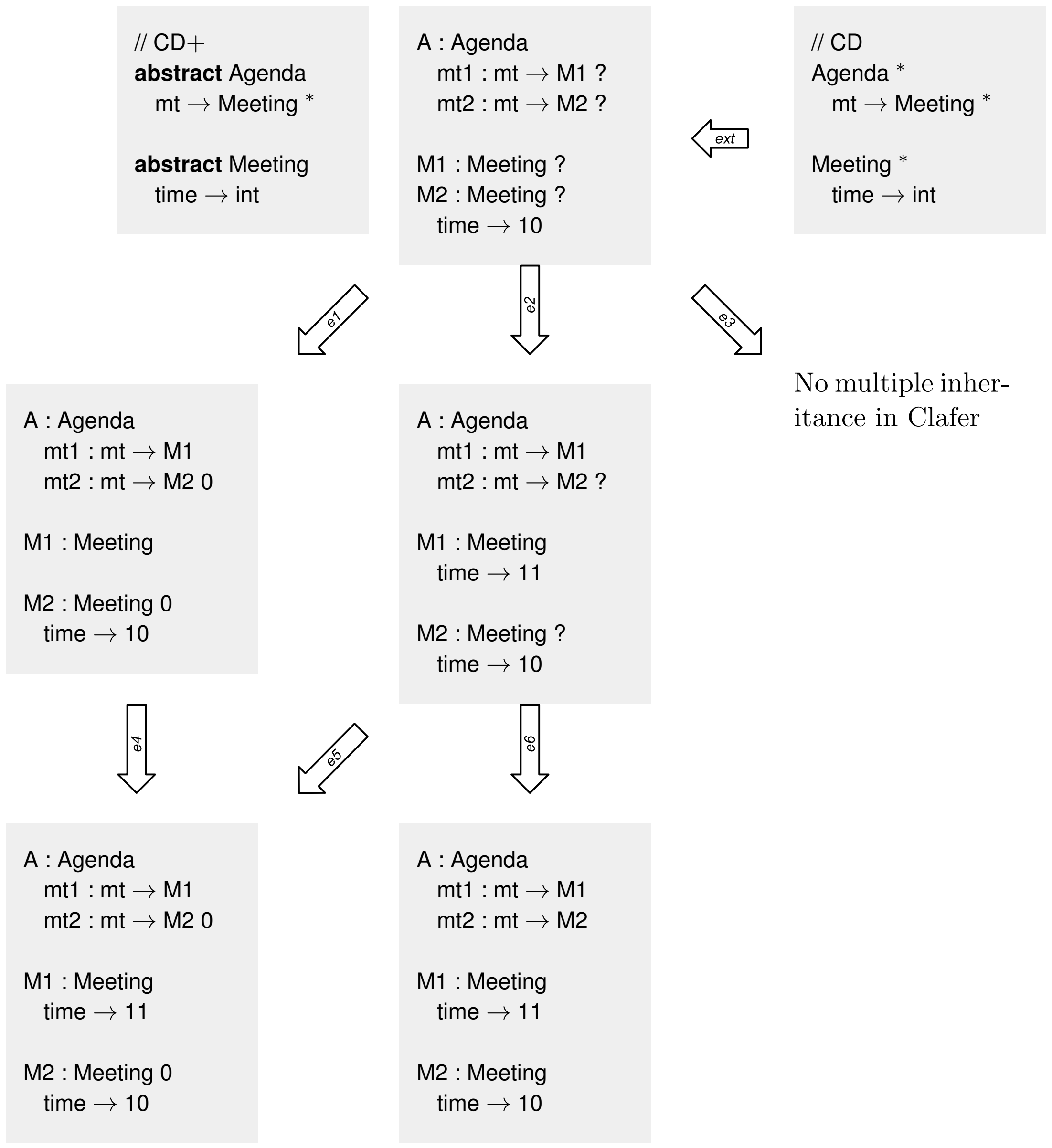
We conjecture that the theorem remains true for the general case of POD completions with gluing, but an accurate proof is our future work.

5.5 Partial Instances in Clafer

Having shown that the encoding of partial instances as class diagrams is sound and complete with respect to completions, we are ready to demonstrate its realization in Clafer. Figure 5.12 shows Clafer models corresponding to the class diagrams from Fig. 5.3 (both figures follow the same visual arrangement of models). The top right corner of the figure encodes the class diagram of Agendas and Meetings.

The diagram CD+ (composed of the two top adjacent boxes) encodes a partial instance representing the uncertainty in Sue's agenda. The clafer A is a singleton (by default). The agenda A contains two optional reference clafers (mt1 and mt2) that redefine mt and point to meetings M1 and M2, respectively. Further, the targets of references are redefined correctly, since M1 and M2 are subtypes of Meeting.

The meetings M1 and M2 are singletons with multiplicities 0..1 (written as ?). The clafer M1 keeps time uncertain, while M2 specifies that the value of time is 10. The meaning of time → 10 is the following: time : time → 10. The former time refers to an element of M2, whereas the latter time refers to the element of the same name inherited from Meeting. Finally, 10 can be considered a predefined singleton subtype of int representing number ten.



```
// CD+
abstract Agenda
  mt → Meeting *

abstract Meeting
  time → int
```

```
A : Agenda
  mt1 : mt → M1 ?
  mt2 : mt → M2 ?

M1 : Meeting ?
M2 : Meeting ?
  time → 10
```

```
// CD
Agenda *
  mt → Meeting *

Meeting *
  time → int
```

```
A : Agenda
  mt1 : mt → M1
  mt2 : mt → M2 0

M1 : Meeting

M2 : Meeting 0
  time → 10
```

```
A : Agenda
  mt1 : mt → M1
  mt2 : mt → M2 ?

M1 : Meeting
  time → 11

M2 : Meeting ?
  time → 10
```

```
A : Agenda
  mt1 : mt → M1
  mt2 : mt → M2 0

M1 : Meeting
  time → 11

M2 : Meeting 0
  time → 10
```

```
A : Agenda
  mt1 : mt → M1
  mt2 : mt → M2

M1 : Meeting
  time → 11

M2 : Meeting
  time → 10
```

Figure 5.12: Extension of class diagrams in Clafer

All extensions *e1*...*e6* besides *e3* follow the same pattern of subclassing and redefinition. The extension *e3* is currently not supported due to lack of multiple inheritance in Clafer. Extensions under OWA (Fig. 5.4) can be encoded analogically.

5.6 Concluding Remarks

Partial instances enable modeling with uncertainty, underspecification, and variability. As an example, we showed their use in OOM in requirements elicitation and validation. We considered partial instances and their completion under the CWA and the OWA. Despite many applications, support for partial instances in OOM languages is limited.

The chapter contributes to the design of modeling notations. It showed that under the CWA partial instances can be encoded as class diagrams by strengthening multiplicity constraints, redefinition, and subclassing. In other words, partial instantiation and subclassing+redefinition are formally equivalent for modeling uncertainty and variability within the presented scope. One of the implications is that any OOM language can offer support for partial instances as long as it offers the notion of subclassing for classes and properties (associations and attributes), and refinement of multiplicities. Our work makes this encoding generic and precise, therefore the presented concepts may be widely applicable.

We see several advantages of this encoding: 1) any OOM language without native support for partial instances can support them at the class level, simply by syntactical means; 2) encoding (partial) instances as class diagrams allows the modeler to specify constraints in the context of each such instance – in contrast, objects in object diagrams cannot contain constraints –, enables mixing abstractions and examples in the same notation – thus facilitating Example-Driven Modeling [28, 6] –, and enables cross-referencing between instances and types, e.g., an object (encoded as a singleton class) can be used as a type of a reference; 3) a user of such a language has fewer concepts to learn.

On the other hand, we also see two main drawbacks of this syntactical unification. A general disadvantage is that fundamental OOM concepts (instances and types) are not directly visible in the syntax, which may lead to confusion in case of singleton classes. Second, the class diagrams that encode partial instances are, arguably, bulky and convoluted. In the presented encoding, we abused class modeling by specifying "degenerated" class diagrams composed of singleton classes. It is unlikely that practitioners would work directly with such diagrams. A dedicated UML profile could address this problem. Clafer avoids this problem by a suitable syntax design.

Clafer natively encodes (partial) instances via classes. We argue that using the same syntax for expressing (partial) instances and class diagrams is a mechanism of syntactical unification of instances and types. Of course, instances still exist in the semantics, but they do not have to be exposed in the concrete syntax. This is another kind of unification in Clafer, besides the unification of classes, associations, and properties. When the two unifications are combined together, they allow to concisely specify: 1) feature models along with their partial configurations; 2) partial instances of meta-models; and 3) redefinition at any nesting level in the containment hierarchy and for clafers playing any role – something that is not possible, e.g., in UML class diagrams.

Chapter 6

Evaluation

In this chapter we evaluate Clafer analytically and experimentally. We argue that Clafer can encode feature and meta-models at least as concisely as state-of-the-art feature and meta-modeling languages. We then evaluate Clafer on realistic feature and meta-models, model templates, and domain models. We show that these rich structural models with complex constraints can be expressed in Clafer and analyzed within seconds. Finally, we discuss threats to validity.

6.1 Analytical Evaluation

We examine the extent to which Clafer meets its design goals from Sect. 2.2.

1. *Clafer provides a concise notation for feature modeling.* This can be seen by comparing Clafer to TVL, a state-of-the-art textual feature modeling language [34]. Figure 6.1 shows the TVL encoding of the feature model from Fig. 3.2. Conciseness can be measured as the number of elements used to encode a model. The Clafer model has slightly fewer concepts than the TVL model. Feature models in Clafer look very similar to feature models in TVL, except that TVL uses explicit keywords (e.g., to declare groups), braces for nesting, and feature names must be unique.

   Clafer's language design reveals several key ingredients allowing a class modeling language to provide a concise notation for feature modeling:

```
Options group allof {
  Size group oneof { Small, Large },
  opt Cache group allof {
    CacheSize group allof {
      SizeVal { int val; },
      opt Fixed
    }
  },
  Constraint { (Small && Cache) → Fixed; }
}
```

Figure 6.1: Options feature model in TVL

- *Concept unification*: The concept clafer unifies basic constructs of structural modeling, such as class, association, and property (which includes attribute, reference, and role). Such a unification enables arbitrary property nesting, which allows us to concisely specify feature models in a class modeling language. Neither UML nor Alloy provide this mechanism; there, associations and classes are declared separately, and properties cannot be arbitrarily nested. Although UML offers association classes, they cannot use primitive domains as association ends.
- *Instance composition and type nesting*: Clafer nesting accomplishes instance composition and type nesting in a single construct. UML provides composition, but type nesting is specified separately (cf. Fig. 2.5b). Alloy has no built-in support for composition and thus requires explicit parent-child constraints. It also has no signature nesting, so name clashes need to be avoided using prefixes or alike.
- *Default singleton multiplicity*: All clafers that have a parent in the containment hierarchy, are singletons by default. It allows one to specify mandatory features without declaring their multiplicity explicitly in the concrete syntax. In UML and Alloy, on the other hand, associations are multi-valued by default.
- *Group constraints*: Clafer's group constraints are expressed concisely as intervals. In UML groups can be specified in OCL, but using a lengthy encoding, explicitly listing features belonging to the group. The same applies to Alloy.
- *Constraints with default quantifiers*: Default quantifiers on relations allow writing constraints that look like propositional logic, even though their underlying semantics is first-order logic. For example, each clafer name in the last line in Fig. 3.2 would be preceded by the quantifier `some` (cf. lines 11-13 in Fig. B.1). Name resolution rules further contribute to the conciseness of constraints.

```
class Comp {
  reference version : integer
}

class ECU extends Comp {}

class Display extends Comp {
  reference server : ECU
  attribute options : Options
}
```

Figure 6.2: Component meta-model in KM3

- *Navigation over optional clafers*: Navigation expressions of the form $n_1.n_2 \ldots n_m$ encompass navigation along the clafer hierarchy and occur in constraints (e.g., in the last line of Fig. 3.3). Each of the names n_i may refer to a clafer of any multiplicity; in particular, the clafer n_1 may be optional, whereas n_2 mandatory. In Clafer and Alloy, all navigation expressions uniformly evaluate to a set (either empty or not). In OCL, however, one needs to explicitly check if navigation over an optional element evaluates to an empty set before proceeding to the next element. Otherwise, the navigation results in an *undefined* value indicating an error. Formally, unconditional mapping composition is defined in OCL for only total mappings, whereas in Clafer and Alloy one can compose partial mappings as well.

2. *Clafer provides a concise notation for meta-modeling.* Figure 6.2 shows the meta-model of Fig. 3.3 encoded in KM3 [80], a state-of-the-art textual meta-modeling language. The most visible syntactic difference between KM3 and Clafer is the use of explicit keywords introducing elements and mandatory braces establishing hierarchy. Both models have the same number of concepts. KM3, however, cannot express additional constraints in the model. They are specified separately, e.g., as OCL invariants.

3. *Clafer allows one to concisely mix feature and meta-models.* Clafer integrates subclassing into hierarchical modeling. Clafers at any nesting level in the containment hierarchy can subclass other clafers. Inheritance among clafers is a semantically rich operation. For example, inheritance among two reference clafers introduces two classes (head and target), four maps ($head$, $\underline{parent}$, $\underline{ref}$, and $head*$), and three inclusions ($head_s$, $head_h$, and $head_t$), with all the constraints regulating well-formedness

of LCS and inheritance. Using inheritance, one can reuse feature or class types in multiple locations; reference clafers allow reusing both types and instances. Feature and class models can be related via constraints.

4. *Clafer provides support for partial instances.* Clafer was designed to allow for modeling with uncertainty (e.g., as in partial models [59]) and variability (e.g., as in FBMTs [46]). Clafer encodes partial instances of models at the level of types. The partiality of models may be specified: 1) explicitly by using clafer multiplicity constraints, and 2) implicitly by omitting constraints when one clafer inherits partially specified elements from another clafer.

   Clafer covers the entire scope of FBMTs, as shown in Chapter 3, and most of the scope of partial models. Below we discuss the four kinds of partialities in partial models [115] and relate them to Clafer:

   - *May* – uncertainty about the existence of model elements. In Clafer it is modeled by mandatory and optional clafers, and constraints restricting their multiplicities.
   - *Abs* – uncertainty about how a set is refined into subsets. The *Abs* partiality can be modeled in Clafer by declaring an abstract clafer and then partitioning it via subclassing and redefinition.
   - *Var* – uncertainty about identity of model elements. The partiality *Var* corresponds to the gluing case of object completion presented in Chapter 5. The current Clafer implementation does not handle it due to lack of multiple inheritance. Multiple inheritance will be added to Clafer in the nearest future, however.
   - *OW* – uncertainty about model completeness. The partiality *OW* is about the distinction between the OWA and the CWA in the interpretation of completeness. The modeler can always extend Clafer models with new elements (as in the OWA). Our current tools, however, perform reasoning under the CWA. Reasoning under OWA is being implemented.

5. *Clafer tries to use a minimal number of concepts and has uniform semantics.* While integrating feature modeling into meta-modeling, our goal was to avoid creating a hybrid language with duplicate concepts. In Clafer, there is no distinction between class and feature types; they are all *clafers*. Features are relations and, besides their obvious role in feature modeling, they also play the role of attributes in meta-modeling.

We also contribute a simplification to feature modeling: Clafer has no explicit feature group construct; instead, every clafer has a group cardinality to constrain the number of children. This is a significant simplification; we no longer need to distinguish between *grouping features* (features used purely for grouping, such as menus) and feature groups. The grouping intention and grouping cardinalities are orthogonal: any clafer can be annotated as a grouping feature, and any clafer may choose to impose grouping constraints on children. This idea has also been adopted in the current draft of CVL [71].

Finally, both feature and class modeling have uniform semantics. Syntactic and semantic unification in Clafer keeps the language small, allows for uniform representation of models, and also simplifies development of tools for model analyses. Further, unification has the potential of simplifying model evolution as fewer special cases (concepts) need to be considered. In general, however, syntactic unification may also have some drawbacks, such as a lack of correspondence between the modeler's intent and native support for that intent in the language, worsened model comprehension, and less efficient model analyses. On the other hand, one could easily introduce two subclasses of clafer: class and feature, allowing the user to state an intention explicitly. All the tools, however, could still benefit from the semantic unification by looking at instances of features and classes as instances of clafer.

6.2 Experimental Evaluation

Our experiment aims to show that Clafer can express a variety of realistic variability models, meta-models, model templates, and domain models, and that useful analyses can be performed on these encodings in reasonable time. It follows that the richness of Clafer's applications does not come at a cost of lost analysis potential with respect to modeling in more specialized languages.

The experiment methodology is summarized in the following steps:

1. *Identify a set of models representative of the main use cases of Clafer: variability modeling, meta-modeling, mixed variability and meta-modeling, and domain modeling.* We chose models that can be encoded in the current implementation of Clafer. The set includes: FODA feature models, class-based meta-models that do not require multiple inheritance, Feature-Based Model Templates that combine feature and class models, and hierarchical domain models.

2. *Select representative analyses.* We studied the analyses in published literature and decided to focus on a popular class of analyses that reduce to model instance finding. These include inconsistency detection, element liveness analysis (detecting whether an element can never be instantiated), offline and interactive configuration, guided editing, etc. Since all these analyses share similar performance characteristics, we decided to use model instance finding, consistency and element liveness analysis as representative.

3. *Translate models into Clafer and record observations.* We created automatic translators for converting models to Clafer when applying simple rewriting rules sufficed. For all other cases translation was performed manually.

4. *Run the analyses and report performance results.* The analyses were performed by using our Clafer compiler, ClaferIG, and then employing the Alloy Analyzer (which is an instance finder) to perform the analysis.

The Clafer compiler is written in Haskell and comprises several chained modules: a lexer, layout resolver, parser, desugarer, semantic analyzer, optimizer, and code generator. The layout resolver makes braces grouping subfeatures optional. Clafer is composed of two languages: the core and the full language. The first one is a minimal language with well-defined semantics (as presented in Chapter 4). The latter is built on top of the core language and provides a large amount of syntactic sugar. The semantic analyzer resolves names and deals with inheritance. The code generator translates the core language into Alloy. The generator has benefited from the knowledge about the class of models, with which it is working, to optimize the translation. Similarly, analyzers for specialized languages have this knowledge.

The experiment was executed on a laptop with a Core Duo 2 @2.4GHz processor and 8GB of RAM, running Linux. The Alloy Analyzer was configured to use Minisat as a solver. All Clafer and generated Alloy models are available online[1]. In the subsequent paragraphs we present and discuss the results for the four subclasses of models.

Variability Models. In order to find representative models we first consulted SPLOT [94] — a popular repository of feature models. We succeeded in automatically translating all 341 models from SPLOT to Clafer (non-generated, human-made models; available as of July 18th, 2013). These included models with and without cross-tree constraints, ranging from a dozen to hundreds of features. Here, we report the most interesting cases together

[1] http://gsd.uwaterloo.ca/kbak/thesis

model name	nature	size [# features]	size [# constraints]	running time [s]
Digital Video System	Realistic	26	3	0.003
Dell Laptops	Realistic	46	110	0.007
Billing	Realistic	88	59	0.016
Android	Realistic	150	56	0.061
eShop	Realistic	287	21	0.043
FM-500-50-1	Generated	500	50	0.117
FM-1000-100-2	Generated	1000	100	0.320
FM-2000-200-3	Generated	2000	200	0.899
FM-5000-500-4	Generated	5000	500	5.000
The Linux Kernel	Realistic	6784	6058	8.935

Table 6.1: Results of consistency analysis for *feature models* expressed in Clafer.

with an additional four, which have been randomly generated; all are listed in Tab. 6.1. Digital Video Systems is a small example with few cross-tree constraints. Dell Laptops models a set of laptops offered by Dell in 2009. This is one of the few models that contains more constraints than features. Billing describes a product line of billing methods; it contains tens of features and constraints. EShop [88] is the largest realistic model that we found on SPLOT. It is a domain model of online stores. The remaining models are randomly generated using SPLOT, with a fixed 10% constraint/variable ratio. Besides SPLOT, we also translated two other models: Android and The Linux Kernel. The former is a realistic variability model of the product line of Android phones. It was created based on publicly available documents. The model goes far beyond standard feature modeling due to the presence of integer and string attributes, abstract clafers, and inheritance. The latter model is a Boolean feature model extracted from KConfig model of the Linux kernel [119]. We included it in the benchmark because it is the largest realistic and publicly-available feature model we are aware of.

We checked the consistency (i.e., lack of contradicting constraints) of each model through instance finding. Tab. 6.1 presents a summary of the results. The analysis time was less than a second for up to two thousand features and less than ten seconds for up to several thousand features. We observed that the biggest bottleneck was translation of Alloy models into CNF formulas by the Alloy Analyzer. Reasoning about CNF formulas in a SAT-solver takes no more than hundreds of milliseconds even for the largest models.

Meta-Models In order to identify representative meta-models, we first turned to the Ecore Meta-model Zoo[2]. From there, we selected the following meta-models: AWK Programs, ATL, ANT, BibTex, UML2, ranging from tens to hundreds of elements. We extracted OCL constraints from the UML specification [103] and manually added them to the Clafer encoding of UML2. We also explored the Repository for Model Driven Development (ReMoDD)[3], from where we selected the following meta-models with OCL constraints: ER2RE, CPFSTool, OMGDD, Workflow, and RBAC.

We translated all the Ecore meta-models into Clafer automatically. One interesting mapping was the translation of EReference elements with eOpposite attribute (symmetric reference), as there is no first-class support for associations in Clafer yet. The same deficiency has been reported in a preliminary study that compared Clafer with UML class diagrams [134]. We modeled the association as two reference clafers related by constraints. Moreover, although for the meta-model of UML2 we expressed multiple inheritance in Clafer syntax, we only performed reasoning for slices that required single inheritance.

Besides meta-models, we also manually encoded OCL constraints from an OCL benchmark [63] (benchmarks B1 and B3) to show that Clafer can express many non-trivial OCL constraints over class models. We observed certain patterns during the translation of OCL constraints to Clafer and believe that this task can be automated for a large class of constraints. Tab. 6.2 presents sample OCL constraints from the UML2 meta-model translated into Clafer. Each constraint, but the last one, was written in the context of some class. Their intuitive meanings are as follows: 1) ownedReception is empty if there is no isActive; 2) endType aggregates all types of memberEnds; 3) if memberEnd's aggregation is different from none, then there are two instances of memberEnd; 4) there are no two types with the same names.

There are several reasons why Clafer constraints are more concise and uniform compared to OCL invariants. Similar to Alloy, every Clafer definition is a relation. This approach eliminates extra constructions such as OCL's collect, allInstances, checking for empty collections, and conversions between scalars and singletons. Finally, assuming the default some quantifier before set expressions (e.g. memberEnd.aggregation - none), we can treat the result of an operation as if it were a propositional formula, thus eliminating extra exists quantifiers. Although more concise, the Clafer constraint language lacks some features of OCL, such as support for higher-order associations, transitive closure, and ordered sets.

We applied automated analyses to: 1) slices of the UML2 meta-model: Class Diagram

[2]http://www.emn.fr/z-info/atlanmod/index.php/Ecore

[3]http://www.cs.colostate.edu/remodd/

Context	OCL	Clafer
Class	(not self.isActive) implies self.ownedReception→isEmpty()	!isActive $\Longrightarrow$ no ownedReception
Association	self.endType = self.memberEnd→ collect(e | e.type)	endType = memberEnd.type
Association	self.memberEnd→exists(aggregation <> Aggregation::none) implies self.memberEnd→size() = 2	memberEnd.aggregation - none $\Longrightarrow$ #memberEnd = 2
–	Type.allInstances() → forAll (t1, t2 | t1 <> t2 implies t1.name <> t2.name)	all disj t1;t2 : Type | t1.name != t2.name

Table 6.2: Constraints in OCL and Clafer.

	size		
meta-model/instance	[#classes]	[#constraints]	running time [s]
State Machines	11	28	0.029
Class Diagram	19	17	0.037
Behaviors	20	13	0.057
CPFSTool	64	62	0.033
ER2RE	89	183	0.151
RBAC	96	128	0.058
Workflow	104	76	0.067
OMGDD	163	34	0.070

Table 6.3: Results of strong consistency analysis for UML2 *meta-model* slices and ReMoDD meta-models in Clafer

from [32], State Machines, and Behaviors; and 2) ReMoDD meta-models with OCL constraints (Tab. 6.3). Each meta-model had tens of classes and our goal was to include a wide range of OCL constraints. We checked the *strong consistency* property [31] for these meta-models. To verify this property, we instantiated the meta-models' elements that were at the bottom of inheritance hierarchy by restricting their multiplicity to be at least one. The same constraints were imposed on containment references within all meta-model elements. The analysis confirmed that none of the meta-models had dead elements. Our results thus showed that element liveness analysis can be done efficiently for realistic meta-models of moderate size.

Feature-Based Model Templates. The next class of models were Feature-Based Model Templates akin to our telematics example from Chapter 2. A FBMT consists of a feature

FBMT	#features/#classes/#constraints	instantiation [s]	liveness [s]
Telematics (8)	8/7/17	0.007	0.049
FindProduct (16)	13/29/10	0.041	0.080
TaxRules (7)	16/24/62	0.069	0.086
Checkout (41)	18/78/314	0.734	2.520

Table 6.4: Analyses for *Feature-Based Model Templates* expressed in Clafer. Parentheses by the model names indicate the number of optional elements in each template.

model (cf. lines 1–10 in Fig. 3.5), a meta-model (cf. Fig. 3.3), a template (cf. Fig. 3.4), and a set of mapping constraints (cf. lines 12–16 in Fig. 3.5). To the best of our knowledge, Electronic Shopping [88] is the largest example of a model template found in the literature. We used its templates, listed in Tab. 6.4, for evaluation: FindProduct and Checkout are activity-diagram templates, and TaxRule is a class-diagram template. Each template has substantial variability in it. All templates have between 10 and 20 features, tens of classes, and from tens to hundreds of constraints. For comparison, we also include our telematics example.

We manually encoded the above FBMTs in Clafer. For each of the diagrams in [88], we took a slice of the UML2 meta-model and created a template that conforms to the meta-model, using mandatory and optional singleton classes as described in Sect. 3.1.4. To create useful and simple slices of UML diagrams, we removed unused attributes and flattened the inheritance hierarchy, since many superclasses were left without any attributes. Thus, the slice preserved the core semantics. Furthermore, we sliced the full feature model, so that it contained only features that appear in diagram. Finally, we added mappings to express dependencies between features and model elements, as described in Sect. 3.1.4.

We performed two types of analyses on FBMTs. First, we created sample feature configurations (like in Fig. 3.6) and instantiated templates in the Alloy Analyzer. We inspected each instance and verified that it was the expected one.

Second, we performed element liveness analysis for the templates. The analysis is similar to element liveness for meta-models [31], but now applied to template elements. We performed the analysis through repeated instance finding; in each iteration we required the presence of groups of non-exclusive model elements. Tab. 6.4 presents a summary of the inspected models and times of analyses.

We consider our results promising since we obtained acceptable timings for slices of realistic models without fully exploiting the potential of reasoners. The results can be further improved by better encoding of slices (for example, representing activity diagram

	size		
domain model	[#clafers]	[#constraints]	running time [s]
Electronic Stability Control	78	10	0.046
Merchandise Financial Planning	142	118	0.044
Business Motivation Model	155	54	0.085

Table 6.5: Results of consistency analysis for domain models in Clafer

edges as relations instead of as sets in Alloy) and using more intelligent slicing methods; e.g. some constraints are redundant, such as setting source and target edges in ActivityNodes. Removing these constraints would speed up reasoning process. However, we can already see that Clafer is a suitable vehicle for specifying FBMTs and analyzing them automatically.

Domain Models. Domain models capture complex business knowledge about the whole domain. They express the knowledge in terms of concepts and constraints. The concepts form a hierarchy where more complex concepts are decomposed into simpler ones. Furthermore, such models contain a fair number of constraints. We encoded several realistic models. Electronic Stability Control[4] is an initial model based on US Electronic Stability Control standard. Merchandise Financial Planning[5] is a domain model extracted from Oracle's documentation. It handles three types of hierarchies: Calendar, Product, and Location. Business Motivation Model[6] is based on documentation by the Business Rules Group. As with the variability models, we performed consistency analysis for each domain model. Table 6.5 reports the results.

6.3 Threats to Validity

External Validity Our evaluation is based on the assumption that we chose representative models, and useful and representative analyses.

All models, except the four randomly generated feature models, were created by humans to model real-word artifacts. Some of the models come from academia, while others from the industry, thus they should share characteristics with other industrial models. The majority of practical models have fewer than a thousand features [83], so reasoning about

[4]http://t3-necsis.cs.uwaterloo.ca:8091/ESC/FMCSA571.126-model

[5]http://t3-necsis.cs.uwaterloo.ca:8091/Merchandise%20Financial%20Planning

[6]http://t3-necsis.cs.uwaterloo.ca:8091/BMM/model

corresponding Clafer models is feasible and efficient. Perhaps the biggest real-world feature model to date is the Linux Kernel model [119]. Our tools handled a Boolean version of this model. Analysis of the full model (with non-Boolean attributes and constraints) may require reasoning with CSP and SMT-solvers instead of SAT-solvers. Working with models of this size requires proper engineering of analyses. Our objective here was to demonstrate feasibility of analyses. Robust tools for Clafer are under development.

We believe that the slices of the UML2 that were selected for the experiment are representative of the entire meta-model, because we picked the parts with more complex constraints. While there are not many existing FBMTs to choose from, the e-commerce example [88] was reverse-engineered from the documentation of an IBM e-commerce platform, making the model quite realistic. Similarly, domain models are also based on existing industrial specifications.

Not all model analyses can be reduced to instance finding performed using combinatorial solvers (i.e., the relational model finder in case of Alloy [125]). Combinatorial analyses, however, are most widely recognized and most effective [18].

Instance finding for models has similar uses to testing and debugging for programs [76]—it helps to uncover flaws in models and assists in evolution and configuration [28]. For example, it helped us to debug the initial telematics model. Some software platforms already provide configuration tools using reasoners. For example, Eclipse uses a SAT-solver to help users select valid sets of plug-ins [89].

Liveness analysis for model elements has been previously exploited, for instance in [124, 31]. Tartler et al. [124] analyze liveness of features in the Linux kernel code, reporting about 60 previously unreported dead features in the released kernel versions. Linux is not strictly a feature-based model template, but its build architecture, which relies on (a form of) feature models and presence conditions in code (i.e., conditional compilation), highly resembles our model templates.

Analyzers based on instance finding solve an NP-hard problem, thus, no hard guarantees can be given for their running times. Although progress in solver technologies has placed these problems in the range of practically tractable, there do exist instances of models and meta-models that cannot be effectively analyzed with our tools. Our experiments aim to show that this does not happen for realistic models.

There exist more sophisticated analyzes (and classes of models) that cannot be addressed with Clafer infrastructure, and are not reflected in our experiment. For example, instance finding based on SAT-solving is limited to instances of bounded size. It is possible to build sophisticated meta-models that only have very large instances. This problem is irrelevant for feature models and model templates as they allow no classes that can be instantiated without bounds. SMT-solvers, however, are able to reason about infinite models.

Moreover, special-purpose languages may require more sophisticated analysis techniques such as behavioral refinement checking, model checking, model equivalence checking, etc. These properties typically go beyond static semantics expressed in structural models and thus are out of scope of generic Clafer tools.

Internal Validity Translating models from one language to another can introduce errors and change the semantics of the resulting model.

We used our own tools to convert SPLOT and Ecore models to Clafer and then to translate Clafer to Alloy. We translated FBMTs and OCL constraints manually. The former is rather straightforward; the latter is more involved. We published all the models so that their correctness can be reviewed independently.

Another threat to correctness is the slice extraction for the UML2 and e-commerce models. Meta-model slicing is a common technique used to speed-up model analyses, where the reasoner processes only relevant parts of the meta-model. We performed it manually, while making sure that all parts relevant to the selected constraints were included; however, the technique can be automated [117].

The correctness of the analyses relies on the correctness of the Clafer compiler and the Alloy analyzer. The Alloy analyzer is a mature tool. We have been testing the Clafer compiler for years by translating hundreds of models to Alloy and inspecting the results.

6.4 Concluding Remarks

We demonstrated Clafer's potential in feature, meta-, and domain modeling. The analytical evaluation showed that Clafer can concisely express feature and meta-models via uniform syntax and unified semantics. The experimental evaluation showed that Clafer can express and analyze a range of structural models augmented with complex constraints: domain, variability, class, and meta-models. Besides the mentioned applications, Clafer has also been used for modeling architectural framework concepts [8] and for specifying the structure of documents in Intelligent ET [109], a tool for extracting knowledge from office documents.

Chapter 7

Related Work

In this chapter we compare Clafer with variability, OO, data modeling, and knowledge representation languages. We also discuss tool support for model analyses and the notion of concept unification.

7.1 Variability Modeling

Clafer builds on several previous works, including encoding feature models as UML class models with OCL [44]; a Clafer-like graphical profile for Ecore, having a bidirectional translation between an annotated Ecore model and its rendering in the graphical syntax [121]; and the Clafer-like notation used to specify Framework-Specific Modeling Languages (FSMLs) [8]. Moreover, feature models have been characterized as views on class diagrams (referred to as ontologies) in [45]. None of these works provided a proper language definition that unifies feature and class modeling, and did not provide an implementation like Clafer's; also, they lacked Clafer's concise constraint notation. Although we introduced Clafer earlier [24], our previous work lacked precise semantics because semantic unification posed a major challenge. In our current work, we precisely specify the unification of feature and class modeling constructs and present Clafer's semantics.

Cardinality-based feature models have been formalized in [42] as context-free grammars, and in [95] using set theory. None of the works covers references, attributes, and a constraint language that can deal with multiply-instantiated features. Our work is more complete in that sense and, in fact, subsumes cardinality-based feature modeling, equipping it with inheritance. Also, none of the works considers unification of feature and class models, whereas we precisely show how it can be done.

Text-based Variability Language (TVL) is a textual feature modeling language [34]. It favors the use of explicit keywords, which some software developers may prefer; Clafer avoids them. TVL covers feature models with attributes, i.e., Boolean features and features of other primitive types such as integer. The key difference is that Clafer is also a class modeling language with multiple instantiation, references, and inheritance. It would be interesting to provide a translation from TVL to Clafer. The opposite translation is only partially possible. Currently TVL models can be analyzed with a SAT-solver.

Common Variability Language (CVL) is an Object Management Group (OMG) proposal for a standard for specifying and resolving variability [71]. CVL is being designed by a working group whose members include variability modeling tool vendors, industrial practitioners, and academics. In contrast with other works, CVL models are not self-contained. CVL allows for introducing variability into any existing model that conforms to the MOF meta-model. In particular, it can be used to create families of UML models. Similar to UML, the language has explicit syntactical constructions for different concepts. For example, depending on feature type it is either *choice* (Boolean feature), *classifier* (feature with multiplicity), or *parameter* (feature with attribute). In contrast with CVL, Clafer is unified and focuses on analyses. It may be used as a CVL's backend for model analyses.

Kconfig and Component Description Language (CDL) are domain-specific languages that model variabilities in operating systems [20]. Kconfig models describe configuration options of the Linux kernel; CDL is used by eCos. Both languages express feature models with attributes. The languages have domain-specific concepts, e.g., Kconfig has tristate features (*yes*, *no*, *module*), while CDL groups features into packages. Both languages mix variability with mechanisms to control how models are presented to users in a configurator. Clafer, on the other hand, is general-purpose, has one construct, and is user-interface agnostic. It can express Kconfig and CDL models, except for the UI aspects.

Asikainen and Männistö present Forfamel, a unified conceptual foundation for feature modeling [10]. The basic concepts of Forfamel and Clafer are similar; both include subfeature, attribute, and subtype relations. The main difference is that Clafer's focus is to provide a concise concrete syntax, such as being able to define feature, feature type, and nesting by stating an indented feature name. Also, the conceptual foundations of Forfamel and Clafer differ; e.g., features in Forfamel correspond to Clafer's instances, but features in Clafer are both types and instances. Also, a feature instance in Forfamel can have several parents; in Clafer, an instance has at most one parent. These differences likely stem from the difference in perspective: Forfamel takes a feature modeling perspective and aims at providing a foundation unifying the many existing extensions to feature modeling; Clafer limits feature modeling to its original Feature-Oriented Domain Analysis (FODA) scope [82], but integrates it into class modeling. Finally, Forfamel considers a constraint language as out of scope, hinting at Object Constraint Language (OCL). Clafer comes with a concise constraint notation.

Decision models group decisions and focus on product derivation from a product family [41]. Notations for specifying decision models include a tabular notation [116] and Synthesis [120]. Clafer models are more expressive, and therefore subsume decision models (modulo language features controlling UI aspects such as visibility).

7.2 Object-Oriented Modeling

UML [103] class and object diagrams are two standardized structural modeling notations. UML class diagrams is a syntactically much richer notation than Clafer. While Clafer reduces structural modeling concepts to *clafer*, UML class diagrams make each concept explicit both in the syntax and semantics (e.g., different types of associations). UML class diagrams offer limited support for property nesting. Clafer, on the other hand, was designed as a language for hierarchical modeling. Both UML class diagrams and Clafer offer support for subclassing of classes and associations, and for partial instances. The latter can be encoded via singleton classes and subclassing.

OCL [101] is a language that augments UML models with constraints. The Clafer constraint language is heavily inspired by Alloy's: sets are always flattened. OCL, on the other hand, allows nested sets (i.e., sets of sets), provides first-class support for ordered sets and bags, and is based on the three-valued logic. The language is very expressive and can also be used for specifying queries and model transformations. OCL, however, has been criticized for complexity, verbosity, and poor concrete syntax [127].

UML object diagrams [103] offer partial support for partial instances. Slots may have unknown values, called nulls, that correspond to our _. Objects and slots, however, cannot be labeled as optional. Our work provides syntax for both partialities and supplies it with a formal semantics. UML class diagrams, on the other hand, can support partial instances "as is" via subclassing of classes, attributes, and associations. Our work makes this encoding precise; it assumes, however, that the typing mapping from object diagrams to class diagrams is total. UML object diagrams allow partial typing for objects, i.e., objects may have a missing classifier. Partial typing is also supported by subclassing, as new attributes and associations can be introduced in subclasses (as in Fig. 5.4) but the presented theory needs to be extended to cover that case (extension for OWA).

UML class and object diagrams are two separate diagrams that exist independently, i.e., have separate meta-classes and cross-referencing in certain ways is not allowed (e.g., an object cannot be used as a type of a reference and links cannot be made between objects and classes). In practice, relating UML object to class diagrams is troublesome, because it requires the modeler to constantly switch attention between separate diagrams. In Clafer, classes can be naturally mixed and cross-referenced with objects (encoded as singleton classes). Also, due to syntactic unification, singleton classes encoding instances can be easily evolved into traditional classes by updating their multiplicities.

Modal Object Diagrams (MODs) [93] extend UML object diagrams with positive/negative and example/invariant modalities. We focused on positive examples; the conflicting example in Sect. 5.2 would be a negative example in MODs. MODs have two further extensions: partial and parametrized object diagrams. The former are related to our labels **?** and extension relation. The latter are related to unknown values _. We provide concrete syntax and semantics for both. MODs were encoded in Alloy as partial instances via existentially quantified formulas, whereas we encode them generically via singletons. Existentially quantified formulas do not reflect explicitly the structure of diagrams.

UML models are specified mostly in a graphical notation, but there are also textual notations. UML-based Specification Environment (USE) [62] is a language and toolkit for analyzing class models. UML Human-Usable Textual Notation (HUTN) [100] is an OMG standard for specifying object models. TextUML Toolkit [104] visualizes textually encoded models. The textual notations are incomplete in many respects, e.g., they lack constraints, associations, or cannot model classes, etc. Clafer is a single language that provides all the above and subsumes the listed textual notations.

In essence, Alloy [76] is a modeling language that reduces OOM to two primitives: signatures (typed sets) and relations. The former correspond to classes; the latter to associations. In the context of variability modeling Alloy has the same shortcomings as UML class diagrams. In contrast with Clafer, Alloy provides no first-class support for redefinition and for encoding hierarchical models (in Clafer, clafers can be nested arbitrarily deeply). Both can be realized indirectly by specifying model constraints. Alloy can encode and analyze more conceptually complex UML class diagrams [4]. Alloy is supported by a reasoner; Kodkod [125] is Alloy's relational model finder. Internally, it uses a SAT-solver for instance finding and model checking. Clafer can additionally use a CSP-solver and has a preliminary implementation using an SMT-solver. Although Kodkod has direct support for partial instances, Alloy does not expose it in the concrete syntax. One way of encoding partial instances is through singletons. We make this encoding precise. AlloyPI [96] extends Alloy with special syntax for partial instances; i.e., types and partial instances have distinct notations.

MOF [102] is an OMG standard for class-based meta-modeling. The language is stratified into an essential subset (EMOF) and a complete one (CMOF). Kernel Meta Meta Model (KM3) [80] is a state-of-the-art textual notation for class-based meta-modeling. As mentioned earlier, class-based meta-modeling languages, such as MOF and KM3 and MOF cannot express feature models as concisely as Clafer. Furthermore, Clafer covers the same scope as MOF, but is based on fewer concepts. In MOF, similarly to UML object diagrams, properties may have unknown values. They are specified as a question mark ? (we use _ for the same purpose). MOF does not consider the second type of partiality, i.e., optional existence of elements (that we label as ?). Clafer models can express both.

Nivel is a meta-modeling language, which was applied to define feature and class modeling languages [9]. It supports deep instantiation, enabling concise definitions of languages with class-like instantiation semantics. Clafer's purpose is different: to provide a concise notation for combining feature and class models within a single model. Nivel could be used to define the abstract syntax of Clafer, but it would not be able to naturally support our concise concrete syntax.

Partial instances occur in the contexts of uncertainty, variability, or underspecification. Modal Transition Systems express partiality of behavioral models by introducing transitions with optional existence [87]. Salay et. al showed an application of partial instances to model uncertainty in software requirements [114] and presented a technique for refinement verification between partial models [113]. Partial models [59] express uncertainty about a concrete model variant. Model templates [46] express variability and model multiple variants simultaneously. These works use annotations (similar to our labels ?) to indicate optional elements. Our work, on the other hand, shows how a language can natively support partial instances by encoding them at the type level. The annotations go beyond the semantics of assumed base languages.

Furthermore, as we have noted in Sect. 2.2.3, certain variants of model templates (partial models, respectively) may be syntactically incorrect and dedicated analyses are needed to detect such variants in advance [46]. The subclassing approach, on the other hand, may encode labeled models at the level of types to make them compatible with the base languages, as shown in this dissertation. Each valid instance of such a model encoded at the meta-level represents a syntactically correct variant. Consequently, the problem of syntactically incorrect variants does not occur. It is possible, however, to define dedicated verification procedures for model templates in Clafer if needed.

Under the OWA, our theory of partial completion assumes that two partial instances, that are completed by another (partial) instance, are consistent with each other. For example, we exclude the case of two partial object diagrams POD_1 and POD_2 that contain an object o with a slot s and the slot has different values in POD_1 and POD_2. The problem of merging inconsistent graph-based structures is discussed in [112]. That work, however, is not concerned with encoding partial instances via types.

Architectural languages, such as AADL [60] and AUTOSAR [106], support subclassing of classes and associations. They are used to define partial architectures and refine subcomponents. Clafer can specify architectural models with variability, and perform consistency analysis and configuration optimization on them.

7.3 Data Modeling

Entity-relationship (ER) model [118] is a classical representation of data in relational database systems. ER models are based on entities, relationships, and attributes. Original ER models are very simple and a number of extensions was proposed, e.g., role names. From the viewpoint of ER, the concept *clafer* unifies entities with relationships and attributes, as the relationship is always reified. Navigation from the head class to the target class in *clafer* is given directly, while in ER diagrams it would have to be derived, e.g., by executing a query. Relational databases typically require models to be in the first normal form (no nested tables). Clafer allows arbitrary nesting of clafers. Clafer, however, some limitations with respect to ER, such as no primary and foreign keys, and no queries for view computation.

Structured Query Language (SQL) [75] is a textual domain-specific language for managing data in relational databases. Originally it was based on the theory of relational algebra and had the same expressivity as First-Order Logic (FOL). Relational algebra applies set operators over finitary relations (database tables). SQL can express non-trivial constraints and queries. In fact, current SQL is Turing complete; Clafer is less expressive.

Partial instances, under the name of incomplete information, is a classical topic in databases, from a seminal (and still influential) paper [72] to lattice-theoretic models [92] to semistructured data [14]. However, this work is based on the *value-oriented* relational data model; optionality of objects and slots is not considered.

Object-Role Modeling (ORM) [67] is a conceptual modeling method for querying and expressing structure, business rules, ontologies, and data. It uses diagrams and provides a Controlled Natural Language (CNL) verbalization so that specification of model elements may resemble English sentences. Although ORM is a complete language, its concrete syntax is quite complex. Similarly to Clafer, ORM treats attributes and more complex concepts uniformly.

Extensible Markup Language (XML) [131] is a standardized markup language for exchanging data over the web and among applications. Conceptually, XML distinguishes elements, attributes, and references. XML Schema (XSD) [130] is a language for describing model of XML documents. Similarly to UML, it has a number of overlapping concepts and types. As XML and XSD, Clafer can naturally express hierarchical data models. Although all three languages support references and attributes, Clafer additionally supports complex constraints and integrates all of that into a concise framework. XML has been criticized for verbosity, complexity, and being hardly human-readable [13, 29]. Clafer's concrete syntax is more concise and is meant to be human-readable.

7.4 Knowledge Representation

Knowledge Representation languages capture the structure of data together with its meaning. Hierarchical models play an important role in Knowledge Representation (KR). They are used for building a common vocabulary, e.g., in the form of ontologies specified in RDF [128] or OWL [132]. The former is a set of triples *subject-predicate-object* and does not have an explicit structure. In contrast, Clafer models recursively decompose concepts. RDF Schema (RDFS) [129] is a semantic extension of Resource Description Framework (RDF) and provides a meta-model for RDF documents.

Web Ontology Language (OWL) [132] is a family of standardized languages based on Description Logic (DL) for modeling ontologies. The idea behind DL was to define the largest decidable fragment of FOL. OWL defines three sublanguages (Lite, DL, and Full) that have different expressive power. OWL DL is decidable, OWL Full is not [135]. OWL focuses on the inheritance hierarchy, whereas Clafer focuses both on the inheritance and the containment hierarchy. In contrast to RDF and OWL, Clafer is based on FOL. Clafer and OWL are supported by reasoners. While OWL tool support focuses on classification, Clafer instantiates models.

Prolog [37] is a declarative logic programming language based on FOL. It has been used in artificial intelligence, natural language processing, and databases. Prolog programs consist of rules and facts. Due to Prolog's expressiveness programs are not guaranteed to terminate. Datalog [33] is a syntactic subset of Prolog that guarantees termination. Prolog is a more general language than Clafer.

OWL relies on the Open World Assumption, while Prolog and Clafer on the Closed World Assumption. Hierarchical models are typically more concise in Clafer than in RDF, OWL, and Prolog.

7.5 Analyses

Many analyses rely on the fundamental problem of checking satisfiability of a propositional formula (SAT problem). SAT-solvers, such as MiniSat [58], Sat4j [21], Chaff [97], are programs that automatically check satisfiability of Boolean formulas. Although, in the worst case the problem is exponential, modern solvers effectively deal with complex formulas.

Working with propositional formulas directly is inconvenient. Kodkod [125] is a relational model finder that translates relational logic models to SAT. CrocoPat [22] is a tool for imperative relational programming. It internally uses Binary Decision Diagrams (BDDs).

Satisfiability Modulo Theory (SMT) extends SAT with theories that are typically within FOL, such as functions, integers, strings, arrays. In contrast with SAT-solvers, SMT-solvers can work with infinite models. Examples of SMT-solvers include Z3 [48], Yices [57], and Hampi [84]. Many solvers implement SMT-LIB [16], a common input language and a set of benchmarks.

SAT and SMT-solvers enable push-button verification, i.e., they work completely automatically. Theorem provers, on the other hand, work interactively. Their reasoning is based on axioms and inference rules. Higher Order Logic (HOL) [2], Isabelle [107], and Prototype Verification System (PVS) [105] are some prominent theorem provers.

Some structural modeling languages have strong mathematical foundations and are meant to be analyzed. Z [74] is a set-based notation for describing models in terms of pre- and post-conditions. The language is supported by theorem provers. Alloy [76] is a state-of-the-art formal modeling language supported by SAT-solvers. Clafer uses the Alloy Analyzer and a CSP-solver to perform model analyses. FORMULA [77] is a general-purpose rich modeling language supported by an SMT-solver. Clafer has a preliminary implementation that uses an SMT-solver as a reasoner.

Other reasoning techniques include Answer Set Programming (ASP) and Rewriting Logic (RL). ASP is a form of purely declarative logic programming. It can express more constructs than SAT, but fewer than SMT. Disjunctive Logic Programming (DLV) system [90] and Smodels [98] are some of ASP-solvers. RL substitutes terms of a formula with other terms. Maude system [36] is a state-of-the-art implementation of RL. Clafer models may contain defaults. Reasoning over such models is easy in Maude or DLV, while it is non-trivial using SAT and SMT-solvers.

7.6 Unification

A rigorous approach to unification of different types of associations based on mathematical operations with mappings, particularly, tabulation, was proposed in [53]. Clafer develops this idea further by introducing: 1) inheritance among clafers, 2) the ability to arbitrarily nest properties, 3) a naming discipline that compacts syntax, and 4) the notion of multi-clafer shape and the corresponding hierarchical view on Class Diagrams.

Formalization and unification of different views on relationships and properties has been done in conceptual modeling, e.g., [133, 47]. In contrast to our work, they typically focus on complex semantic aspects rather than seemingly simple tabular and navigational aspects, and use first-order rather than diagrammatic logic and algebra.

In OOM, unification often appears within the problem of discontinuity in transition from the design to implementation. The attempts to bridge the gap ranged from designing a new programming language (or extending an existing one) [85, 56] to defining relationships at implementation level in terms of an existing language constructs [30, 17, 66] to concrete work on implementing UML associations in Java [123, 3, 61]. These works rarely discuss implementation of reified associations, which are basic for Clafer. UML 2 has both, navigable and reified associations, but the corresponding fragment of the meta-model is inconsistent [52].

Refactoring of UML class diagrams is a practical activity performed during model evolution [122, 12]. We acknowledge the importance of refactorings, but our idea with Clafer is more radical: we conjecture that unification of modeling concepts reduces the number of required model refactorings.

Formalization of conceptual modeling constructs within a framework based on diagram predicates and operations over sets and mappings was proposed in [51, 49]. The subsequent idea to interpret various diagrammatic notation used in structural modeling as different visualizations of the same format of DP-graphs is developed in [54] (where DP-graphs are called *sketches*). Particularly, considering UML class diagrams as visualizations of formal class diagrams is elaborated in [50, 53]. Several applications of DP-graphs to model-driven software engineering are developed in [110, 111]

Relating problem-space feature models and solution-space models has a long tradition. For example, feature models have been used to configure model templates before [39, 68]. That work considered model templates as superimposed instances of a meta-model and presence conditions attached to individual elements of the instances; however, Clafer implements model templates as specializations of a meta-model. Such a solution allows us treating the feature model, the meta-model, and the template at the same metalevel, simply as parts of a single Clafer model. This design allows us to elegantly reuse a single constraint language at all these levels. As another example, Janota and Botterweck show how to relate feature and architectural models using constraints [78]. Again, our work differs from this work in that our goal is to provide such integration within a single language. Such integration is given in Kumbang [11], which is a language that supports both feature and architectural models, related via constraints. Kumbang models are translated to Weight Constraint Rule Language (WCRL), which has a reasoner supporting model analysis and instantiation. Kumbang provides a rich domain-specific vocabulary, including features, components, interfaces, and ports; however, Clafer's goal is a minimal clean language covering both feature and class modeling, and serving as a platform to derive such domain specific languages, as needed.

7.7 Concluding remarks

In this chapter we placed Clafer in the context of structural modeling languages, partial instantiation, model analyses, and concept unification. Clafer has similar expressivity as the discussed languages and subsumes some of them. Similarly to Alloy and FORMULA, it offers push-button verification. Clafer reduces structural modeling to single, well-defined concept: *clafer* that has the characteristics of class, association, and property. As such, it may be used in the design of future hierarchical modeling languages that integrate feature and class modeling.

Chapter 8

Conclusion

In this dissertation we presented our research on modeling and analysis of variability in SPLs. The premise for our work are usage scenarios mixing feature and class models together, such as representing components as classes and their configuration options as feature hierarchies, and relating feature models and component models using constraints. In Chapter 2, we showed that modeling problem and solution space using either cardinality-based feature modeling or class modeling brings extra complexity, and cardinality-based feature models lack precise semantics. Moreover, we also argued that representing partial instances (of meta-models) and relating feature to component configurations is non-trivial.

In Chapter 3, we introduced Clafer. We integrated feature modeling into class modeling, rather than trying to extend feature modeling as previously done [42]. The design of Clafer revealed that a class modeling language can provide a concise notation for feature modeling if it unifies concepts, supports clafer nesting, group cardinalities, and constraints with default quantifiers. Also, hierarchical redefinition and encoding partial instances via subclassing allow to concisely mix feature and class models. This work also has implications for programming language design. Current OO programming languages suffer from the same problems as UML class diagrams. The need for arbitrary object nesting (declaring containment and the type of contained object) has been recognized in, for example, JavaScript Object Notation (JSON) [38] and Protocol Buffers [65].

We formalized Clafer in Chapter 4. The language design and precise definition were a major effort. It took us a considerable amount of time to understand the basic concepts of structural modeling, discuss various design choices, evaluate them on examples, and to implement them in tools. Our design contributes a precise characterization of the relationship between feature and class modeling. Representing both types of models in single language allows us to use a common infrastructure for model analysis and instantiation, as shown in Chapter 6.

In Chapter 5 we demonstrated how to encode partial instances as class models. While several previous works encode partial instances as singletons (singleton idiom in Alloy; in AADL and in AUTOSAR, the components are nested and they have cardinalities—AUTOSAR calls them prototypes), we are not aware of a formalization of this idea. The presented theory makes the concept of partial instances via subclassing and its relation to explicit partial instances precise, improving the understanding of both approaches to language design and their tradeoffs. The encoding allows any OOM language without native support for partial instances to support them at the class level. Clafer natively uses this encoding to specify class models and to encode object models in the same syntax.

8.1 Limitations and Future Work

The work on Clafer presented in this dissertation is limited to structural modeling. As of now, Clafer provides no first-class support for behavioral modeling and analysis. Behavioral extensions, however, are under investigation and will become a part of Clafer in the future.

Clafer lacks some features of prominent structural modeling languages, such as UML class diagrams and Alloy. Due to asymmetry of textual syntax, Clafer has no first-class support for bidirectional associations. The semantics, however, fully support them. We have proposed several syntactic variants for encoding associations. Clafer also lacks support for multiple inheritance. Moreover, in contrast with OCL and Alloy, Clafer constraint language does not support functions, transitive closure, and operations on relations. They remain future work.

Clafer is intended for abstract modeling, therefore Clafer models typically do not contain all the necessary implementational details. For example, the language does not distinguish unordered and ordered sets, as OCL does. Unlike Kconfig and CDL, Clafer is not concerned with the way of presenting models in user interface. It is unclear whether a structural modeling language should be able to define that.

Clafer has a compact, yet non-customizable syntax. Although it can express a variety of models, the current concrete syntax may be unnatural for some of them. Providing customizable concrete syntax (on top of the unified semantics) remains future work.

Clafer strives for minimality while providing expressiveness of concept-rich languages. There are tradeoffs between notations that are unified and that explicate concepts in concrete syntax. Unification allows keeping the language small and simplifies implementation of tools, as fewer special cases need to be considered; however, users may prefer an explicit notation to better express their intents, for better comprehension, and, potentially, for more efficient tools. Once two concepts are unified in semantics, differentiating them in syntax is easy (e.g., using annotations or syntactic extensions as in DSLs). The benefit is that a common infrastructure can still be used owing to the semantic unification. Unification also eases composition in the language (e.g., polymorphism in OO eases objects composition). In a sense, unification is of interest also for languages that are targets of DSL extensions. The benefits and disadvantages of concept unification in practical modeling should be investigated further in user studies. Also, we are not aware of any objective and precise guidelines that specify which language construct should be unified and which should express domain concepts explicitly. Such guidelines would be helpful for designing more usable notations.

We have done preliminary tests with real-world users to evaluate Clafer's usability, i.e., the ease of learning, writing, and understanding. A small test with students, BAs, and consultants showed that they are able to specify correct models after reading a short tutorial [5]. We also created an exercise [27] for the SE464 (Software Design and Architecture) course students where they modeled, analyzed, and optimized a Software Product Line of mobile phones. The students expressed positive comments on language usability and usefulness of analyses, but complained about the complex process of setting up tools. Proper user studies are needed to objectively evaluate the usability of Clafer.

The current Clafer toolchain supports model analyses, collaborative model development, and configuration. All these activities are performed with no connection to source code. We acknowledge that models have limited use on their own; they should be related with the source code. Tools for generating code from Clafer models remain future work.

In the context of partial instances, the formal part of our work focused on completion and extension under the CWA. It omitted the case of completion with gluing instances. The latter case and formalization under the OWA remain future work.

References

[1] choco: an Open Source Java Constraint Programming Library. http://www.emn.fr/z-info/choco-solver/. (online; accessed August 2013).

[2] The HOL system description. http://hol.sourceforge.net. (online; accessed August 2013).

[3] D. Akehurst, G. Howells, and K. Mcdonald-Maier. Implementing associations: UML 2.0 to Java 5. *SOSYM*, 6, 2007.

[4] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. 2007.

[5] Michał Antkiewicz. Concept modeling using Clafer: Tutorial. Technical report, GSD Lab, University of Waterloo, 2011.

[6] Michał Antkiewicz, Kacper Bąk, Krzysztof Czarnecki, Dina Zayan, Andrzej Wąsowski, and Zinovy Diskin. Example-Driven Modeling Using Clafer. In *MDEBE*, 2013.

[7] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Jimmy Liang, Rafael Olaechea, and Krzysztof Czarnecki. Clafer Tools for Product Line Engineering. In *SPLC*, 2013.

[8] Michał Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *IEEE TSE*, 35(6), 2009.

[9] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4), 2009.

[10] Timo Asikainen, Tomi Männistö, and Timo Soininen. A unified conceptual foundation for feature modelling. In *SPLC*, 2006.

[11] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Adv. Eng. Inform.*, 21(1), 2007.

[12] Dave Astels. Refactoring with UML. In *XP*, 2002.

[13] Jeff Atwood. XML: The Angle Bracket Tax. http://www.codinghorror.com/blog/2008/05/xml-the-angle-bracket-tax.html. (online; accessed August 2013).

[14] Pablo Barceló, Leonid Libkin, Antonella Poggi, and Cristina Sirangelo. XML with incomplete information: models, properties, and query answering. In *PODS*, 2009.

[15] Michael Barr and Charles Wells. *Category theory for computing science*, volume 10. Prentice Hall New York, 1990.

[16] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0 Reference Manual. http://www.smt-lib.org. (online; accessed August 2013).

[17] W. Harrison C. Barton and M. Raghavachari. *Mapping UML designs to Java*, volume 35. ACM, 2000.

[18] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[19] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.

[20] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *ASE*, 2012.

[21] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7, 2010.

[22] Dirk Beyer. Relational Programming with CrocoPat. In *ICSE*, 2006.

[23] Kacper Bąk. Optimized translation of Clafer models to Alloy. Technical report, GSD Lab, University of Waterloo, 2011.

[24] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and meta-models in Clafer: mixed, specialized, and coupled. SLE, 2010.

[25] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: Unifying Class and Feature Modeling. In *Submitted to SOSYM*, 2013.

[26] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Partial Instances via Subclassing. In *SLE*, 2013.

[27] Kacper Bąk, Rafael Olaechea, Michał Antkiewicz, and Krzysztof Czarnecki. Clafer Exercise. http://goo.gl/cEgZ4D. (online; accessed August 2013).

[28] Kacper Bąk, Dina Zayan, Krzysztof Czarnecki, Michał Antkiewicz, Zinovy Diskin, Andrzej Wąsowski, and Derek Rayside. Example-Driven Modeling. Model = Abstractions + Examples. In *ICSE*, 2013.

[29] Jeff Bone. Does XML Suck? Revisited. http://www.oreillynet.com/xml/blog/2002/08/does_xml_suck_revisited.html. (online; accessed August 2013).

[30] C. Bunse and C. Atkinson. The normal object form: bridging the gap from models to code. 1999.

[31] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams Using Constraint Programming. In *MoDeVVA*, 2008.

[32] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL contracts for the verification of model transformations. In *OCL workshop of MoDELS*, 2009.

[33] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989.

[34] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12), 2011.

[35] Matthias Clauß. Modeling variability with UML. In *GCSE*, 2001.

[36] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2001.

[37] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, 2003.

[38] Douglas Crockford. Request for Comments: 4627. The application/json Media Type for JavaScript Object Notation (JSON). https://tools.ietf.org/html/rfc4627. (online; accessed August 2013).

[39] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

[40] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In *GPCE*, 2002.

[41] Krzysztof Czarnecki, Paul Grüenbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *VaMoS*, 2012.

[42] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *SPIP*, 10(1), 2005.

[43] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2), 2005.

[44] Krzysztof Czarnecki and Chang H. Kim. Cardinality-based feature modeling and constraints: A progress report. In *OOPSLA*, 2005.

[45] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC*, 2006.

[46] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, 2006.

[47] M. Dahchour, A. Pirotte, and E. Zimányi. Generic relationships in information modeling. *JDSIV*, 3730, 2005.

[48] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963, 2008.

[49] Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *DKE*, 47, 2003.

[50] Zinovy Diskin. Visualization vs. specification in diagrammatic notations: A case study with the UML. In *Diagrams*, 2002.

[51] Zinovy Diskin and Boris Cadish. Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In *OOER*, 1995.

[52] Zinovy Diskin and Jürgen Dingel. Mappings, maps and tables: Towards formal semantics for associations in uml2. In *MoDELS*, 2006.

[53] Zinovy Diskin, Steve Easterbrook, and Juergen Dingel. Engineering Associations: From Models to Code and Back through Semantics. In *TOOLS*. 2008.

[54] Zinovy Diskin, Boris Kadish, Frank Piessens, and Michael Johnson. Universal arrow foundations for visual modeling. In *Diagrams*, 2000.

[55] Zinovy Diskin and Uwe Wolter. A diagrammatic logic for object-oriented visual modeling. *Electronic Notes in Theoretical Computer Science*, 203(6), 2008.

[56] S. Ducasse, M. Blay-Fornarino, and A. M. Pinna-Dery. A reflective model for first class dependencies. In *OOPSLA*, 1995.

[57] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, SRI Computer Science Laboratory, 2006.

[58] Niklas Een and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919, 2004.

[59] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*, 2012.

[60] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[61] D. Gessenharter. Mapping the UML2 semantics of associations to a Java code generation model. 2008.

[62] Martin Gogolla, Fabian Büttner, and mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69(1-3), 2007.

[63] Martin Gogolla, Mirco Kuhlmann, and Fabian Büttner. A benchmark for ocl engine accuracy, determinateness, and efficiency. In *MODELS*. 2008.

[64] Hassan Gomaa. *Designing software product lines with UML*. Addison-Wesley Boston, USA;, 2004.

[65] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/. (online; accessed August 2013).

[66] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *OOPSLA*, 2004.

[67] Terry Halpin. ORM2. In *OTM*, 2005.

[68] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *ICSE*, 2008.

[69] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies. SLE, 2010.

[70] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in linux and ecos. In *VaMoS*, 2012.

[71] IBM, Thales, Fraunhofer FOKUS, and TCS. *Proposal for Common Variability Language (CVL) Revised Submission*, 2012.

[72] Tomasz Imieliński and Witold Lipski. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[73] Software Enginnering Institute. Catalog of software product lines. http://www.sei.cmu.edu/productlines/. (online; accessed August 2013).

[74] ISO/IEC. *ISO/IEC13568:2002: Information technology – Z formal specification notation – Syntax, type system and semantics*. 2002.

[75] ISO/IEC. *ISO/IEC 9075-1:2008: Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. 2008.

[76] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2011.

[77] Ethan K. Jackson and Janos Sztipanovits. Towards A Formal Foundation For Domain Specific Modeling Languages. In *EMSOFT*, 2006.

[78] Mikolás Janota and Goetz Botterweck. Formal approach to integrating feature and architecture models. In *FASE*, 2008.

[79] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9), 2005.

[80] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *IFIP*, 2006.

[81] Kyo C. Kang. FODA: Twenty years of perspective on feature modeling. In *VaMoS*, 2010.

[82] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Nowak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, CMU, 1990.

[83] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

[84] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver For String Constraints. In *ISSTA*, 2009.

[85] B. B. Kristensen. Complex associations: abstractions in object-oriented modeling. In *OOPSLA*, 1994.

[86] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[87] Kim G Larsen and Bent Thomsen. A modal process logic. In *LICS*, 1988.

[88] Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, 2006.

[89] Daniel Le Berre and Pascal Rapicault. Dependency management for the Eclipse ecosystem: Eclipse p2, metadata and resolution. In *IWOCE*, 2009.

[90] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3), 2006.

[91] Jimmy Liang. Solving Clafer Models with Choco. Technical Report GSDLab-TR 2012-12-30, GSD Lab, University of Waterloo, 2012.

[92] Leonid Libkin. Approximation in databases. In *ICDT*, 1995.

[93] Shahar Maoz, Jan Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *ECOOP*, 2011.

[94] Marcílio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T. - Software Product Lines Online Tools. In *OOPSLA*, 2009.

[95] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *VaMoS*, 2011.

[96] Vajih Montaghami and Derek Rayside. Extending Alloy with partial instances. In *ABZ*, 2012.

[97] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and harad. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.

[98] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In *NMR*, 2000.

[99] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. Modeling and multi-objective optimization of quality attributes in variability-rich software. In *NFPinDSML*.

[100] OMG. *UML Human-Usable Textual Notation*, 2004.

[101] OMG. *Object Constraint Language*, 2010.

[102] OMG. *Meta Object Facility (MOF) Core Specification*, 2011.

[103] OMG. *OMG Unified Modeling Language*, 2011.

[104] Open-source. TextUML Toolkit. http://abstratt.com/textuml/. (online; accessed August 2013).

[105] Sam Owre, Sree Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. Springer-Verlag, 1996.

[106] AUTOSAR Partnership. Release 4.1. http://www.autosar.org/. (online; accessed August 2013).

[107] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Springer – Berlin, 1994.

[108] Klaus Pohl, Gunter Bockle, and Frank van der Linden. *Software product line engineering*, volume 10. Springer, 2005.

[109] Rehan Rauf, Michał Antkiewicz, and Krzysztof Czarnecki. Logical Structure Extraction from Software Requirements Documents. In *RE*, 2011.

[110] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *JLAP*, 79(7), 2010.

[111] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 81(4), 2012.

[112] Mehrdad Sabetzadeh and Steve Easterbrook. Analysis of inconsistency in graph-based viewpoints: a category-theoretical approach. In *ASE*, 2003.

[113] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In *ICST*, 2012.

[114] Rick Salay, Marsha Chechik, and Jennifer Horkoff. Managing requirements uncertainty with partial models. In *RE*, 2012.

[115] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In *FASE*. 2012.

[116] K. Schmid and I. John. A customizable approach to full lifecycle variability management. *SCP*, 53, 2004.

[117] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-Driven Slicing of UML/OCL Models. In *ASE*, 2010.

[118] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1, 1976.

[119] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Variability model of the linux kernel. In *VaMoS*, 2010.

[120] Software Productivity Consortium Services Corporation. Reuse-driven software processes guidebook, version 02.00.03. Technical Report SPC-92019-CMC, 1993.

[121] Matthew Stephan and Michał Antkiewicz. Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical Report 2008-08, Univeristy of Waterloo, 2008.

[122] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *UML*, 2001.

[123] C. Suscheck and B. Sandén. A construct for effectively implementing semantic associations. *JOT*, 2(3), 2003.

[124] Reinhard Tartler, Julio Sincero, and Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *FOSD*, 2009.

[125] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[126] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer, 2007.

[127] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the object constraint language of UML. Technical report, MIT Laboratory for Computer Science, 1999.

[128] W3C. *RDF Primer*, 2004.

[129] W3C. *RDF Schema*, 2004.

[130] W3C. *XML Schema*, 2004.

[131] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008.

[132] W3C. *OWL 2 Web Ontology Language Document Overview*, 2009.

[133] Y. Wand, V. Storey, and R. Weber. An ontological analysis of the relationship construct in conceptual modeling. *TODS*, 24(4), 1999.

[134] Dina O. Zayan. Model Evolution: Comparative Study between Clafer and Textual UML. Technical report, GSD Lab, University of Waterloo, 2012.

[135] Evgeny Zolin. Complexity of reasoning in Description Logics. http://www.cs.man.ac.uk/~ezolin/dl/. (online; accessed August 2013).

APPENDICES

Appendix A

Formal Class Diagrams

A.1 Notation and Terminology

A.1.1 Mappings

By a *mapping f from a source set A to a target set B* we understand a function that sends each element of A to a collection (perhaps, empty) $f(a)$ of elements of B. We say that f is **multi-valued**. The most general collection we consider is a bag (or a family, or an indexed set) of elements – precise definitions are in Sect. A.2.1. We call a mapping **set-valued**, if all bags $f(a)$ are actually sets (UML then annotates the mapping with marker "unique"). A set-valued mapping is **single-valued**, if all non-empty sets $f(a)$ are singletons.

A mapping is **total** if all bags $f(a)$ are not empty; otherwise it is **strictly partial**. An underspecified mapping, which may be total but not necessarily, is called **partial**. Thus, totality and strict partiality are constraints that a general (partial) mapping may satisfy. Following a common practice, we often say partial instead of strictly partial. Note that according to the definition above, a single-valued mapping can be partial.

A set-valued mapping is **inclusion** if its source is a subset of the target, $A \subset B$, and for all $A \in A$, $f(a) = a$ (but a on the right of equality is considered as an element of B). An inclusion of A into itself is the identity mapping $\mathit{id}_A \colon A \to A$.

A set-valued mapping is **containment** if its inverse is total and single-valued.

Table A.1 presents the mappings and their notation. We mark a non-constrained bag-valued mapping with the label [bag] while being set-valued is assumed by default and we

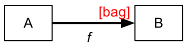
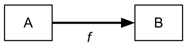
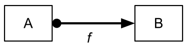
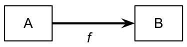
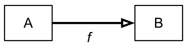
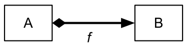
Mapping	Arrow	Intended semantics
partial bag-valued		No constraints.
partial set-valued		$f(a)$ is a set for all $a \in A$
total		for any $a \in A$ there is $b \in f(a)$.
single-valued		for any $a \in A$ there is at most one $b \in f(a)$.
inclusion		$A \subset B$ and $f(a) = a$.
containment		for any $b \in B$ there is exactly one $a \in A$ s.t. $f(a) = b$.

Table A.1: Notational conventions for mappings

hide the predicate [set]. Note the difference between the arrow heads for a general multi-valued mapping (a black triangle) and a single-valued mapping (an open arrow-head).

A.1.2 Shapes and Fonts

We use the following terminology and conventions to formally specify models and meta-models. Boxes (called classes) represent sets, boxes with rounded edges represent primitive domains (e.g., Integer), arrows (called maps) represent mappings between sets. Names of classes in models are written in Serif font, whereas names of classes in meta-models are written in Small Capital font. Names of maps are in *italic* font. If a class or map name is predefined then it is underlined. Diagram predicates are [red and enclosed in brackets]. For multiplicities we skip the braces and write numbers (e.g., 1), or number ranges (e.g., 1..1). Derived elements are shown as blue dashed.

A.2 Semantics and syntax of DP-graphs (formal CDs)

In the OO modeling view, the world consists of *objects* and *links* between them. We typically collect objects into *sets*, and links into *mappings* between these sets. Taken together, objects, links, sets, and mappings, constitute a huge universe denoted by SetMap. A particular OO model (class diagram) specifies a small fragment of the universe; usually, by describing a diagram of sets and mappings involved in the fragment, and properties they must satisfy. Here, we outline the basics of a mathematical framework, in which such OO-modeling can be formally specified. We first consider semantics, i.e., the universe SetMap as such, in Sect. A.2.1–A.2.3, and then proceed with syntactical means for specifying fragments of SetMap (Sect. A.2.4). We assume that the basic notions of naive set-theory (set, subset, an ordered pair etc., and (single-valued) function, injection, bijection, etc.) are known.

A.2.1 Semantic universe: Mappings

We give two definitions: *relational* (a mapping is a span of functions), and *navigational* or *functional* (a mapping is a multi-valued function as in Sect. A.1.1). Then we show that both essentially define the same construct called a *mapping*.

Multi-relations or Spans.

Let A and B be sets. By a mapping from A to B we can understand a set of *labeled links*, i.e., triples (a, b, ℓ) with $a \in A$, $b \in B$, and $\ell \in L$ a label taken from some predefined set L of link IDs, so that multiple links between the same a and b are possible. The following definition makes the idea precise.

Definition 1 (span) *A* multi-relation *or a* span, *$r\colon A \nrightarrow B$, is a triple (L_r, s_r, t_r) with L_r a set, and s_r, t_r two totally defined single-valued functions from L_r as shown in the following diagram:*

(span) $$A \xleftarrow{s_r} L_r \xrightarrow{t_r} B.$$

Set L_r is the head*, functions s_r, t_r are the* source *and* target legs*, and sets A, B are the* source *and* target feet *of the span.*

We will denote the set of all spans from A to B by $\mathsf{Span}(A, B)$.

To ease reading formulas, we will align them with the geometry of diagram (span) and write $a = s_r.\ell$ to denote application of function s_r to $\ell \in L_r$, which results in $a \in A$, and similarly $\ell.t_r = b$ denotes t_r applied to ℓ with result $b \in B$. The triple (a, ℓ, b) is then called an *r-link* ℓ from a to b.

It is easy to see that a span $r : A \nrightarrow B$ gives rise to a total single-valued function $\hat{r}\colon L_r \to A \times B$ (even if r is strictly partial), and conversely, any such function gives us a span with $s_r = \hat{r}.p$ and $t_r = \hat{r}.q$ where $p\colon A \leftarrow A \times B$ and $q\colon A \times B \to B$ are projection functions. (Totality of r is equivalent to *left-surjectivity* of $\hat{r}$, i.e., surjectivity of s_r.) Any relation $r \subset A \times B$ is a span, whose head is r and legs are projections restricted to r. Hence, the set of all relations $\mathsf{Rel}(A, B)$ is included into $\mathsf{Span}(A, B)$.

A span $r \in \mathsf{Span}(A, B)$ can be seen as a *multi-relation*, i.e., a relation with possible repetitive pairs of elements (links). We can eliminate repetitive links by considering the image of L_r, i.e., set $L_r! \overset{\text{def}}{=} \hat{r}(L_r) \subset A \times B$, which consists of pairs of elements, i.e., is a binary relation. It gives rise to a *reduct* span $r!$, whose head is $L_r!$, and legs are restrictions of projections p, q above to set $L_r!$. Thus, we have a function $!_{A,B}\colon \mathsf{Span}(A, B) \to \mathsf{Rel}(A, B)$; the subindex will be omitted if it is clear from the context.

We will also need the notion of span isomorphism, in which the number of links matters, not their IDs.

Definition 2 *Two spans $r1, r2\colon A \nrightarrow B$ are considered* isomorphic*, $r1 \cong r2$, iff there is a bijection between their heads commuting with legs.*

Multi-valued functions.

By a mapping from A to B we can also understand a function that sends each element of A to a collection (perhaps, empty) $f(a)$ of elements of B. Hence, we first need to define collections.

Definition 3 (families or bags) *Let X be a set. A* family *of X's elements is given by an* indexed set *I and a function $x\colon I \to X$, for which we prefer to write x_i for the value $x(i)$, $i \in I$. Correspondingly, the graph of function x, i.e., the set $\{(i, x_i)|\ i \in I\}$ is denoted by exprression $\{\{x_i|\ i \in I\}\}$, where double-brackets indicate that repetitions (say, $x_i = x_j$ for $i \neq j$) are possible. In the UML parlance, such double-bracketed expressions are often called* bags*, and we also use this term. However, formally, a bag is a family, i.e., the graph of the indexing function of the family.*

The set of all bags of X's elements is denoted $\mathsf{Bag}(X)$.

Note that an ordinary subset A of X can be seen as a bag $\{\{a_a|\ a \in A\}\}$=$\{(a, a)|\ a \in A\}$, which is the graph of inclusion A into X. Thus, the powerset of X, $\mathsf{Set}(X)$, is included into $\mathsf{Bag}(X)$.

Any bag/family $x \in \mathsf{Bag}(X)$ can be compressed to its *carrier* set by eliminating repetitions, i.e., by taking the image $\{x_i |\ i \in I\}$ of the indexing function. We denote the resulting set by $x! \subset X$. We thus have a function $!_X : \mathsf{Bag}(X) \to \mathsf{Set}(X)$; the subindex will be omitted if it is clear from the context.

Definition 4 *A* multi-valued (mv) *function, $f : A \twoheadrightarrow B$, is a single-valued total function $f : A \to \mathsf{Bag}(X)$. Given $a \in A$, we will denote the indexing function of family $f(a)$ as $f^a : I^a_f \to B$.*

By composing f with $!_X$, we obtain the set-valued reduct of f, function $f! : A \to \mathsf{Set}(X)$.

Like for span isomorphism, what maters is the number of indices rather than their IDs.

Definition 5 *Two mv-functions $f1, f2 : A \twoheadrightarrow B$ are considered* isomorphic*, $f1 \cong f2$, if for each $a \in A$, there is a bijection between the indexing sets $\iota_a : I^a_{f1} \to I^a_{f2}$ commuting with the indexing functions fi^a, that is, $\iota_a.f2^a = f1^a$.*

Spans and functions together: Mappings

Given a span $r : A \nrightarrow B$, we can build a multi-valued function $r^* : A \twoheadrightarrow B$ by defining for a given $a \in A$,

1) the index set $I^a_{r^*} = s_r^{-1}(a) \subset L_r$, and

2) for a link $\ell \in I^a_{r^*}$ considered as an index, $r^{*a}(\ell) = \ell.t_r$ (see Def.4 for the notation used).

In a slightly different notation,

$$\begin{aligned} r^*(a) &= \quad \{\{b_\ell |\ \ell \in I^a_{r^*} \text{ and } \ell.t_r = b_\ell)\}\} \\ &= \{(\ell, b_\ell) |\ \ell \in I^a_{r^*} \text{ and } \ell.t_r = b_\ell\}, \end{aligned} \tag{A.1}$$

Given any mv-function $f : A \twoheadrightarrow B$, we build a span $f^* : A \nrightarrow B$ by defining

1) the set of links $L_{f^*} = \uplus\{I^a_f |\ a \in A\}$ (where $\uplus$ denotes *disjoint* union), and

2) for any link $\ell \in L_{f^*}$, $s_{f^*}(\ell) = a$ iff $\ell \in I^a_f$, and $t_{f^*}(\ell) = f^a(\ell)$.

Theorem 1 *For any span $r : A \nrightarrow B$, $r^{**} \cong r$ and $r!^* = r^*!$.*

For any mv-function $f : A \twoheadrightarrow B$, $f^{**} \cong f$ and $f!^* = f^*!$.

Proof.. Straightforward checking

Thus, spans and mv-functions are two equivalent ways of specifying a unidirectional association between two sets. We can use either of them to make technicalities easier. We thus use a loose term "mapping" as a reference to either a span, or an equivalent mv-function. For example, working with set-valued functions is convenient, and this is how we have interpreted non-bag arrows in our formal CDs. However, if we need to consider instantiation in the classical UML sense via typed graphs, direct linking and, hence, spans, may be a better choice. As for bag-valued functions, working with them is technically much simpler in the span representation.

For a puristically oriented reader, we can define a mapping in the Clafer spirit as a pair (r, f) with $r\colon A \nrightarrow B$ a span and $f\colon A \twoheadrightarrow B$ an mv-function with $r^* \cong f$ (or, equivalently, $f^* \cong r$).

Set-valued mappings.

Given an mv-function $f\colon A \twoheadrightarrow B$ and an element $a \in A$, suppose that the indexing function $f^a\colon I_f^a \to B$ (see Def. 4) is injective. Then the bag $f(a)$ does not have repetitions, indexes can be forgotten, and the bag can be seen as a subset of B. If all indexing functions are injections, then all bags $f(a)$ can be seen as subsets, and $f \cong f!\colon A \to \mathsf{Set}(B)$. We then say that f is a *set-valued* function.

It is easy to see that in the span representation, the counterpart of set-valued functions are relations. A mapping/span $r : A \nrightarrow B$ is a *relation* iff each element/link $\ell \in L_r$ is completely identified by the pair $(s_f.\ell, \ell.t_f)$.

Theorem 2 *A span $r\colon A \nrightarrow B$ is a relation iff its navigational counterpart, mv-function $r^*\colon A \twoheadrightarrow B$, is set-valued.*

A.2.2 Semantic universe: Operations on mappings

(Sequential) Mapping Composition.

For set-valued mappings, $f\colon A \twoheadrightarrow B$, $g\colon B \twoheadrightarrow C$, their composition is an ordinary functional composition: for any $a \in A$, $a.f.g = (a.f).g$, where for a set $X \subset B$, $X.g \overset{\text{def}}{=} \bigcup_{x\in X} x.g$.

For the general case of bag-valued mappings, it is much easier to define composition for the span representation. Given two consecutive spans $q : A \nrightarrow B$, $r : B \nrightarrow C$, their *composition* $q.r\colon A \nrightarrow C$ is defined as follows. The head

$$L_{q.r} \overset{\text{def}}{=} \{(\ell, \ell') \in L_q \times L_r : \ell_1.t_q = s_r.\ell_2\},$$

and the legs are defined by setting

$$s_{q.r}.(\ell, \ell') = s_q.\ell, \text{ and } (\ell, \ell').t_{q.r} = \ell'.t_r,$$

which is a straightforward generalization of the ordinary binary relation composition for the general span case. It is easy to see that if spans are relations and hence functions q^*, r^* are set-valued, two definitions of composition coincide (up to isomorphism).

Note that composition of set-valued mappings can be bag-valued. For example, suppose that $A = \{a\}$, $B = \{b_1, b_2\}$, $C = \{c\}$, and mappings are defined functionally: $f^*(a) = \{b_1, b_2\}$, and $g^*(b_1) = g^*(b_2) = \{c\}$. Then $f.g$ consists of two links, $\ell_1 = (ab_1, b_1c)$ and $\ell_2 = (ab_2, b_2c)$, so that $(f.g)^*(a)$ is a bag $\{\{c_{\ell_1}, c_{\ell_2}\}\}$.

With so defined mapping composition, we can check that given a span $f : A \nrightarrow B$, its navigational counterpart f^* is actually the composition $s_f^{-1}.t_f$. Although functions s_f and t_f are always set-valued, their composition can be bag-valued as demonstrated by the example above. Think of B as the head L_β of some span β with $s_\beta = (f^*)^{-1}$, and $t_\beta = g^*$. Then $\beta^* = s_\beta^{-1}.t_\beta = f.g$.

Inversion.

Given a set-valued function $f\colon A \twoheadrightarrow B$, its *inverse* is a set-valued function $g\colon A \twoheadleftarrow B$ such that the equivalence

$$a \in g.b \Leftrightarrow a.f \ni b$$

holds for any $a \in A$ and $b \in B$. For the general case of bag-valued functions, we again resort to spans.

Given a span $r : A \nrightarrow B$, its *inverse* $r^{-1} : A \nleftarrow B$ is defined as follows: $L_{r^{-1}} = L_r$, $s_{r^{-1}} = t_r$ and $t_{r^{-1}} = s_r$. That is, the inverse of mappings uses the same span but swaps the roles of its legs. If spans are relations, both definitions coincide. It is evident that $r^{-1^{-1}} = r$.

A.2.3 Semantic universe: Configurations of mappings and their properties

Table A.2 presents several important properties of mapping configurations, which we call *diagram predicates*. The left column gives their names, the middle one specifies their *arities*,

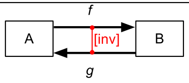
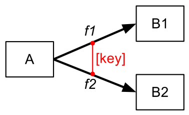
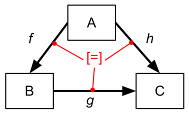
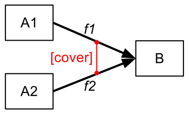
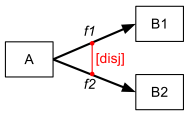
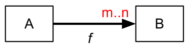
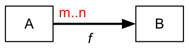
Predicate name/symbol	**Shape**	**Intended semantics** (elements a, a', b, b' range over A, B resp.)
inv		maps f, g are mutually inverse iff their spans f^*, g^* are such.
key		$f_i(a) = f_i(a')$ for both $i = 1, 2$ implies $a = a'$.
=		$f^*.g^* = h^*$
cover		for any $b \in B$ there is $a \in A_i$ s.t. $b \in f_i!(a)$ for i=1 or 2, or both.
disj		$f_1!(a) \cap f_2!(a') = \varnothing$ for all $a \neq a'$.
mult-trg		$m \leq \|f(a)\| \leq n$.
mult-src		$m \leq \|g(b)\| \leq n$ where g is the inverse of f.

Table A.2: A signature of diagram predicates (the labels [bag] are omitted)

i.e., configurations of mappings that may have the property, and the right column provides semantics.

A.2.4 Syntax: DP-graphs

As mentioned, an OO model specifies a fragment of the universe by describing a diagram of sets and mappings involved in the fragment, and properties they must satisfy. That is, a model appears as a graph with diagram predicate declarations (a DP-graph) that describe properties. Formal CDs used in the paper are DP-graphs, whose nodes are called *classes*, arrows are *maps (= unidirectional assications)*, and predicate declarations are *constraints* imposed on classes and maps. Hence, semantics of a formal CDs is given by interpreting its classes as sets, and maps as mappings such that the constraints are satisfied. In this section we specify syntax and semantics of DP-graphs/formal CDs formally.

Graphs

Definition 6 (Graphs and their moprhisms.) *A* (directed multi)graph *G consists of a set of nodes G_N, a set of arrows G_A, and two total single-valued functions $src, trg\colon G_A \to G_N$ giving each arrow its source and target.*

A graph morphism (mapping) *$f\colon G_1 \to G_2$ is a pair of functions $f_N\colon G_{1N} \to G_{2N}$ and $f_A\colon G_{1A} \to G_{2A}$ such that the incidence of nodes and arrows is preserved: for any arrow $a \in G1_A$, $src(f_A(a)) = f_N(src(a))$ and $trg(f_A(a)) = f_N(trg(a))$.*

DP-Graphs

Definition 7 (Signature.) *A* (diagram predicate) signature *is a set Σ of* predicate symbols *together with assignment to each label $P \in \Sigma$ its* arity shape *– a graph $G_{\rm ar}(P)$.*

Definition 8 ((Diagram) formulas.) *Given a diagram predicate signature Σ and a graph G, a* (diagram) formula *over G is a pair $(P, args)$ with $P \in \Sigma$ a predicate symbol, and $args\colon G_{\rm ar}(P) \to G$ a graph morphism binding formal parameters in the arity graph by the actual arguments — elements of graph G. Assume the arity graph $G_{\rm ar}(P)$ has a finite set of arrows $\alpha_1 ... \alpha_n$, and does not have isolated nodes. Then a formula can be encoded by an expression $P(a_1 ... a_n)$ where $a_i = args(\alpha_i)$, $i = 1..n$. In other words, a formula is a pair $(P, \vec{a})$ with P a predicate symbol and $\vec{a}$ a bag of arrows of the carrier graph, whose indexing set is the arity graph $G_{\rm ar}(P)$ (note that the indexing function must be a correct graph morphism – this constraint was called [ad], arity discipline, in Sect. 4.2.1.*

Definition 9 (DP-Graph) *A* DP-graph *is a pair $S = (G, \Phi)$ with G a* carrier *graph, and Φ a set of formulas over G (here S stands for* specification*, or* sketch *— a family of categorical constructs similar to DP-graphs).*

Definition 10 (DP-Graph Morphisms.) *A* DP-graph morphism (mapping) *$f : S_1 \to S_2$ is a morphism of the carrier graphs, $f : G_1 \to G_2$, compatible with formulas in the following way.*

We first note that any graph morphism $f : G_1 \to G_2$ translates formulas over G_1 into formulas over G_2: any formula $\phi = P(a_1..a_n)$ in Φ_1 (with $a_i = \alpha_i.args$) is translated into a formula over G_2, $f(\phi) = P(a_1.f, ..., a_n.f)$—indeed, $args.f : G_{\mathrm{ar}}(P) \to G_2$ is a graph morphism. Then we require that all translated formulas were declared in Φ_2: $f(\phi) \in \Phi_2$ for all $\phi \in \Phi_1$.

A.2.5 Syntax and semantics together

A major idea of categorical logic [15] is to treat semantic universes syntactically, that is, in our case, as DP-graphs. Indeed, the universe SetMap can be seen as a huge (categoricians would say *big*) DP-graph: its nodes are sets, arrows are mappings, and formulas are true statements about sets and mappings. For example, if A and B are sets (nodes in graph $\Gamma[\textsc{SetMap}]$), and f, g are mappings between them (i.e., arrows in $\Gamma[\textsc{SetMap}]$) going in the opposite direction, then, if mappings f and g are mutually inverse, i.e., $(f, g) \models$ [inv], then we add formula [inv](f, g) to set $\Phi[\textsc{SetMap}]$. Thus, the big set of formulas $\Phi[\textsc{SetMap}]$ consists of all valid statements about all possible configurations of sets and mappings matching predicate arities. Then an instance of a DP-graph S can be seen as a DP-graph morphism $[\![..]\!] : S \to \textsc{SetMap}$. An immediate consequence of such an arrangement is the following important result:

Theorem 3 *Any DP-graph morphism $f : S_1 \to S_2$ gives rise to a function between the respective sets of instances, $[\![f]\!] : [\![S_1]\!] \leftarrow [\![S_2]\!]$, where $[\![S_i]\!]$ denotes the (big) set of all instances of DP-graph S_i.*

Proof.. As a correct instance of S_2 is a correct graph morphism, $[\![..]\!] : S_2 \to \textsc{SetMap}$, its composition with f gives us a correct instance of S_1.

Appendix B

Details of Clafer

B.1 Clafer Concrete Syntax

Conciseness is an important goal for Clafer; therefore, it provides syntactic sugar for common constructions. Figure B.1 shows the model from Fig. 3.2 in a desugared notation, in which the defaults (e.g., multiplicities) are inserted into clafer declarations. Each declaration starts with group cardinality, followed by name, optional supertype, then by optional clafer's target, and ends with multiplicity (see Fig. B.2). The desugared notation shows that all clafers nested in an abstract clafer are abstract by default. There are also different kinds of clafers: *basic* (have no reference target), *reference set* (name followed by '$\rightarrow$' symbol, and *reference bag* (name followed by '$\twoheadrightarrow$' symbol); not shown.

Clafer multiplicity is given by an interval $m..n$. Clafer provides syntactic sugar similar to syntax of regular expressions: `?` (optional) denotes 0..1; `*` denotes 0..$*$; and `+` denotes 1..$*$. By default, clafers have multiplicity 1..1.

Group cardinality is given by an interval $m..n$, with the same restrictions on m and n as for multiplicities, or by a keyword: `xor` denotes 1..1; `or` denotes 1..$*$; `opt` denotes 0..$*$; and `mux` denotes 0..1; further, each of the keywords makes children optional by default. For example, `xor` on `size` (line 2) states that only one child instance of either `small` or `large` is allowed. No explicit group cardinality stands for 0..$*$, except when it is inherited from clafers supertype.

```
abstract 0..* options 0..* {
  abstract 1..1 size 1..1 {
    abstract 0..* small 0..1 {}
    abstract 0..* large 0..1 {}
  }
  abstract 0..* cache 0..1 {
    abstract 0..* size → int 1..1 {
      abstract 0..* fixed 0..1 {}
    }
  }
  [ some this.size.small &&
    some this.cache  ⟹
    some this.cache.size.fixed ]
}
```

Figure B.1: Desugared Clafer model.

$\langle$Clafer$\rangle \Rightarrow \langle$Abs$\rangle$ $\langle$GCard$\rangle$ string $\langle$Super$\rangle$ $\langle$Target$\rangle$ $\langle$Card$\rangle$ $\langle$Elements$\rangle$
$\langle$Abs$\rangle \Rightarrow$ | **abstract**
$\langle$Elements$\rangle \Rightarrow$ **{**$\langle$ElList$\rangle$**}** $\langle$ElList$\rangle \Rightarrow$ | $\langle$Element$\rangle$ $\langle$ElList$\rangle$
$\langle$Element$\rangle \Rightarrow \langle$Clafer$\rangle$ | $\langle$Constraint$\rangle$
$\langle$Super$\rangle \Rightarrow$ | **:** string
$\langle$Target$\rangle \Rightarrow$ | $\langle$Kind$\rangle$ string
$\langle$Kind$\rangle \Rightarrow \rightarrow$ | $\twoheadrightarrow$
$\langle$GCard$\rangle \Rightarrow$ | **xor** | **or** | **mux** | **opt** | $\langle$NCard$\rangle$
$\langle$Card$\rangle \Rightarrow$ | **?** | **+** | ***** | $\langle$NCard$\rangle$
$\langle$NCard$\rangle \Rightarrow$ integer **..** ExInteger
$\langle$ExInteger$\rangle \Rightarrow$ ***** | integer

Figure B.2: BNF grammar of Clafer (no constraints).

B.2 Clafer Constraint Language

The Clafer constraint language is essentially borrowed from Alloy [76]. The two most significant differences are name resolution rules and the default some quantifier before clafer names. Both developments contribute to conciseness of the constraints defined over hierarchical models. Constraints are logical expressions composed of terms and logical operators. Terms either relate values (integers, strings) or are navigational expressions. The value of navigational expression is always a set, therefore each expression must be preceded by a *quantifier*, such as no (requires set to be empty), one (requires set to have one element), lone (requires set to have at most one element), or some (requires the set to be non-empty). Lack of explicit quantifier (Fig. 3.2) stands for some (Fig. B.1).

Although the constraints are specified over Clafer models, we define their semantics over Class Diagrams (our semantic domain). A formal class diagram of Clafer model is composed of classes, maps, and constraints. An instance of class diagram is an object diagram that is composed of objects and links; it must satisfy constraints defined over CD. Each constraint is defined in context of a class. The context corresponds to defining constraints nested under clafers, because in an LCS the head class represents the clafer. If in a Clafer model constraint is defined at top-level, then in the corresponding LMCS and CD it is defined in the context of synthetic root. The constraint language used in Clafer allows one to define new diagram predicates of shapes spanning several Clafer shapes and whose semantics is expressible in first-order logic.

B.2.1 Grammar

Figure B.3 shows grammar of the core constraint language. The full constraint language has additional syntactic sugar, but any constraint may be desugared to the core constraint language. In the first production in Fig. B.3 *var* represents variables bound by quantifiers. In the production with binary operators, $\oplus$ is one of $<, =, +, -$ (logical comparison, equality, addition, and subtraction, respectively). In the production with set operators, $\otimes$ is one of $++, --, \&, in$ (set union, difference, intersection, and subsetting, respectively). The last production *Name* represents names of *head* maps that correspond to clafer names, or is one of reserved names. When *Name*s form a sequence $n_1.n_2 \ldots n_m$, we call such as an expression a *navigation*. The dot between names indicates *relational join*.

⟨Exp⟩ ⇒	
all var **:** ⟨SetExp⟩ | ⟨Exp⟩	*universal quantification*
| ⟨Exp⟩ **&&** ⟨Exp⟩	*conjunction*
| **!** ⟨Exp⟩	*negation*
| ⟨Exp⟩ $\oplus$ ⟨Exp⟩	*binary operators*
| # ⟨Exp⟩	*set cardinality*
| ⟨SetExp⟩	*set expression*
⟨SetExp⟩ ⇒	
⟨SetExp⟩ $\otimes$ ⟨SetExp⟩	*set operators*
| ⟨SetExp⟩ **.** ⟨SetExp⟩	*relational join*
| Name	*reserved/map name*

Figure B.3: BNF grammar of core Clafer constraints

B.2.2 Name Resolution Rules

Name resolution rules disambiguate names of clafers used in constraints. The rules are needed as clafer names may repeat in Clafer model. The rules are applied during compilation of Clafer model to LMCS; thus LMCS and CD have all names properly resolved. The rules are similar to CVL rules [71], as the latter were inspired by Clafer. A name is resolved in the context of a clafer (top-level constraints are defined in the context of synthetic root) as follows:

1. *Reserved names.* Check if it is a special name: such as *parent*, *ref*, and this. The latter indicates object for which the constraint is evaluated. Further, primitive domains also use reserved names, int for integers, and string for strings.

2. *Binding.* Check if name is introduced by a local variable (used in constraints with quantifiers).

3. *Descendants.* Look up the name in descendant clafers of the context clafer in breadth-first search manner. If a clafer has supertype, take into account inherited clafers.

4. *Targets.* Similar to the previous step but additionally take into account clafers reachable via references.

5. *Ancestors.* Search in the ancestors clafers starting from the parent clafer of the context and up. For each ancestor, look up the name using the rules *Descendants* and, if necessary, *Targets.*

6. *Top level.* Search in other top-level clafers. For each clafer apply rules *Descendants* and, if necessary, *Targets.*

7. *Error.* If the name cannot be resolved or is ambiguous within a single step, the constraint is not well-formed and an error is reported.

For navigations (expressions of the form $n_1.n_2 \ldots n_m$) the name resolution rules are applied to resolve n_1 first. Once it is resolved, subsequent clafers ($n_2.n_3 \ldots n_m$) are resolved by applying only rules *Reserved names*, *Descendants*, and *Error*. Note that n_1 becomes the context clafer for resolving n_2, and n_2 becomes the context for n_3, etc. A fully resolved name is a navigation that starts with this, i.e., is of the form $\mathsf{this}.c_2 \ldots c_m$.

B.2.3 Type Rules

The type system is specified in a series of formal rules.

$$\frac{statementA}{statementB}$$

The above rule says that if A holds, then B follows.

Expressions

Universal quantification. In the rule below the environment *env* is extended by specifying that the type of var is SetExp.

$$\frac{env, \mathsf{var} :: \mathsf{SetExp} \vdash \mathsf{Exp} :: Boolean}{env \vdash \mathsf{all\ var : SetExp} \quad | \quad \mathsf{Exp} :: Boolean}$$

Conjunction.

$$\frac{env \vdash \mathsf{Exp1} :: Boolean \quad env \vdash \mathsf{Exp2} :: Boolean}{env \vdash \mathsf{Exp1\ \&\&\ Exp2} :: Boolean}$$

Negation. $\dfrac{env \vdash \mathsf{Exp} :: Boolean}{env \vdash \mathsf{!Exp} :: Boolean}$

Comparison.

$$\frac{env \vdash \mathsf{Exp1} :: \tau \quad env \vdash \mathsf{Exp2} :: \tau}{env \vdash \mathsf{Exp1} \oplus \mathsf{Exp2} :: Boolean},$$

where $\oplus \in \{<, =\}$.

Arithmetic operator.

$$\frac{env \vdash \mathsf{Exp1} :: \tau \quad env \vdash \mathsf{Exp2} :: \tau}{env \vdash \mathsf{Exp1} + \mathsf{Exp2} :: \tau}\text{, where } \oplus \in \{+, -\}.$$

Set cardinality. $$\frac{env \vdash \mathsf{Exp} :: \tau}{env \vdash \mathsf{\#Exp} :: \mathit{Integer}}$$

Set expression. $env \vdash \mathsf{Exp} :: \tau$

Set Expressions

Set operators.

$$\frac{env \vdash \mathsf{Exp1} :: \tau \quad env \vdash \mathsf{Exp2} :: \tau}{env \vdash \mathsf{Exp1} \mathbin{\mathsf{++}} \mathsf{Exp2} :: \tau},$$

where $\oplus \in \{++, --, \&\}$.

Subsetting. $$\frac{env \vdash \mathsf{Exp1} :: \tau \quad env \vdash \mathsf{Exp2} :: \tau}{env \vdash \mathsf{Exp1\ in\ Exp2} :: \mathit{Boolean}}$$

Relational join.

$$\frac{env \vdash \mathsf{Exp1} :: \tau \times \upsilon \quad env \vdash \mathsf{Exp2} :: \upsilon \times \phi}{env \vdash \mathsf{Exp1.Exp2} :: \tau \times \phi}$$

$$\frac{env \vdash \mathsf{Exp1} :: \tau \quad env \vdash \mathsf{Exp2} :: \tau \times \upsilon}{env \vdash \mathsf{Exp1.Exp2} :: \upsilon}$$

Reserved/map name.

$env \vdash \mathsf{this} :: \tau$

$env \vdash \mathsf{Name} :: \tau \times \upsilon$

B.2.4 Semantics

The semantics assumes that: 1) navigation paths have already been resolved to specific clafers (head classes), and 2) all expressions are correctly typed. A constraint is specified in the context of a class, and is evaluated in the context of each instance (object) of that class. We call the latter context an *environment*. For an object o we initialize environment to be $env = \{\mathsf{this} \mapsto o\}$

$Env = Var \rightarrow Value$

$Value = \mathcal{P}(Object) \cup \mathcal{P}(Link)$

Environment maps variables to values, which are either sets of objects or links. Note that a single object would be represented as a singleton set.

A constraint is a Boolean-valued expression. The semantics uses two interpretation functions:

$[\![]\!]_E : Exp \rightarrow Env \rightarrow Boolean \cup \mathcal{P}(Object)$

$[\![]\!]_S : SetExp \rightarrow Env \rightarrow Value$

The former function interprets abstract syntax elements of expressions for a given environment and evaluates to a Boolean value or a set of objects. A set of objects is always a singleton. In particular, values of primitive domains (e.g., integer) are encoded as singletons. Analogically, the latter function interprets set expressions, which are either sets of objects or links.

Semantics of Expressions

Universal quantification. For universal quantification the expression Exp has to hold for each instance of SetExp. It is done by extending the environment with a mapping from from var to an instance.

$[\![\mathsf{all\ var : SetExp\ |\ Exp}]\!]_E\ env = \bigwedge\{[\![\mathsf{Exp}]\!]_E\ (env \oplus \mathsf{var} \mapsto v) | v \in [\![\mathsf{SetExp}]\!]_S\ env\}$

Conjunction.

$[\![\mathsf{Exp1\ \&\&\ Exp2}]\!]_E\ env = [\![\mathsf{Exp1}]\!]_E\ env \wedge [\![\mathsf{Exp2}]\!]_E\ env$

Negation. $[\![\mathsf{!Exp}]\!]_E\ env = \neg\ [\![\mathsf{Exp}]\!]_E\ env$

Less than. $[\![\mathsf{Exp1 < Exp2}]\!]_E\ env = [\![\mathsf{Exp1}]\!]_E\ env < [\![\mathsf{Exp2}]\!]_E\ env$

Equality. $[\![\mathsf{Exp1 = Exp2}]\!]_E\ env = ([\![\mathsf{Exp1}]\!]_E\ env = [\![\mathsf{Exp2}]\!]_E\ env)$

Subsetting.

$[\![\mathsf{Exp1\ in\ Exp2}]\!]_E\ env = [\![\mathsf{Exp1}]\!]_E\ env \subseteq [\![\mathsf{Exp2}]\!]_E\ env$

Addition. $[\![\mathsf{Exp1 + Exp2}]\!]_E\ env = [\![\mathsf{Exp1}]\!]_E\ env + [\![\mathsf{Exp2}]\!]_E\ env$

Subtraction. $[\![\mathsf{Exp1 - Exp2}]\!]_E\ env = [\![\mathsf{Exp1}]\!]_E\ env - [\![\mathsf{Exp2}]\!]_E\ env$

Set cardinality. $[\![\mathsf{\#Exp}]\!]_E\ env = |\ [\![\mathsf{Exp}]\!]_E\ env|$

Set expression. Although set expressions are also expressions, they must be quantified to evaluate to a Boolean value. $[\![\mathsf{SetExp}]\!]_E\ env = [\![\mathsf{SetExp}]\!]_S\ env$

Semantics of Set Expressions

Union. $[\![\mathsf{Exp1 ++ Exp2}]\!]_S\ env = [\![\mathsf{Exp1}]\!]_S\ env \cup [\![\mathsf{Exp2}]\!]_E\ env$

Difference. $[\![\mathsf{Exp1 -- Exp2}]\!]_S\ env = [\![\mathsf{Exp1}]\!]_S\ env \setminus [\![\mathsf{Exp2}]\!]_E\ env$

Intersection. $[\![\mathsf{Exp1\ \&\ Exp2}]\!]_S\ env = [\![\mathsf{Exp1}]\!]_S\ env \cap [\![\mathsf{Exp2}]\!]_E\ env$

Relational join. Relational join (the dot operator) joins two relations. In our formal CDs all navigational expressions start with the this keyword, which is then followed by names of maps (clafer names): *Name*. The this keyword indicates an object; it can be viewed as a unary relation. Furthermore, all other relations are binary, thus the final result of each navigation expression is always a set of objects. If any of the components of the navigational expression evaluates to an empty set, the final result is also an empty set.

$[\![\mathsf{Exp1.Exp2}]\!]_S\ env =$
$\{(x,z) | \exists y((x,y) \in [\![\mathsf{Exp1}]\!]_S env \wedge (y,z) \in [\![\mathsf{Exp2}]\!]_S env)\}$

Reserved/map name. It refers to names of maps in formal CD. $[\![\mathsf{Name}]\!]_S\ env = env(\mathsf{Name})$

B.3 LMCS Constraints

Each Labeled Multi-Clafer Shape is only valid if it satisfies incidence constraints (defined in Tab. B.1), clafer kind/shape discipline constraints (defined in Tab. B.2), clafer cojoining constraints (defined in Tab. B.3), and naming discipline constraints (defined in Tab. B.4).

Description	**Constraints**
The map *head_map* goes from class (role) *source_class* to class *head_class*. Analogical constraints hold for the maps *parent_map*, *ref_map*, and *target_map*.	$\text{this}.head\_map.so = \text{this}.source\_class$ $\text{this}.head\_map.ta = \text{this}.head\_class$

Table B.1: Incidence constraints in the context of CLAFER, which the LMCS meta-model in Fig. 4.12 must satisfy

Description	**Constraints**
The LCS of $\underline{\text{Sing}}$ has only the class $\underline{\text{Sing}}$ as head and does not participate in inheritance.	$\text{this} \in \underline{\textsc{Sing}} \implies$ $\text{this}.source\_class = \bot$ $\text{this}.target\_class = \bot$ $\text{this}.super = \bot$
The LCS of Dom is a basic clafer whose parent is $\underline{\text{Sing}}$.	$\text{this} \in \textsc{Dom} \implies$ $\text{this}.source\_class = \underline{\text{Sing}}$ $\text{this}.target\_class = \bot$ $\text{this}.super = \bot$
Basic clafers have a *source_class* but no *target_class*. Analogical constraints apply to reference clafers.	$\text{this} \in \textsc{BasicClafer} \implies$ $\text{this}.source\_class \neq \bot$ $\text{this}.target\_class = \bot$
Top-level clafers, that are not a synthetic root, are abstract.	$\text{this}.parent = \bot \wedge \text{this} \neq \underline{\textsc{Sing}} \implies$ $\text{this}.abstract = true$

Table B.2: Clafer kind constraints in the context of CLAFER, which the LMCS meta-model in Fig. 4.12 must satisfy

Description	**Constraints**
The class *source_ class* of given LCS is a class *head_ class* of the parent LCS. Analogical constraints hold for cojoining LCS with the LCS of its target.	$\text{this}.source\_\ class = \text{this}.parent.head\_\ class$ $\text{this}.target\_\ class = \text{this}.target.head\_\ class$
In case of subclassing between two clafers, there exists an inclusion between the head classes of the two LCSs. Analogical constraints hold for maps $super_s$ and $super_t$ if the corresponding source and target classes exist in both LCSs.	$\text{this}.super \neq \bot \implies$ $\text{this}.super_h.so = \text{this}.head\_\ class$ $\text{this}.super_h.ta = \text{this}.super.head\_\ class$

Table B.3: Clafer cojoining LMCS constraints in the context of CLAFER, which the LMCS meta-model in Fig. 4.12 must satisfy

Description	**Constraints**
The map *head* has the same name as the class *head*.	$\text{this}.head\_\ map.label = \text{this}.head\_\ class.label$
The map *parent* is named "<u>parent</u>". Analogical constraints holds for the map *ref*.	$\text{this}.parent\_\ map.label = \text{"}\underline{\text{parent}}\text{"}$
The map *target* (if defined) has name derived from the name of the class *head* by concatenating the name with *.	$\text{this}.target\_\ map.label =$ $concat(\text{this}.head\_\ map.label, *)$
There is one distinguished element of STRING named "<u>Sing</u>". Analogical constraints hold for other predefined clafers, such as "<u>int</u>" and "<u>string</u>".	$\text{this} \in \underline{\text{SING}} \iff \text{this}.head\_\ class.label = \text{"}\underline{\text{Sing}}\text{"}$

Table B.4: Naming constraints in the context of CLAFER, which the operation *Compile* must satisfy

Appendix C

Modeling in Clafer

C.1 Full Telematics Model

Below is the running example of telematics system modeled in Clafer.

```
abstract options
  xor size
    small ?
    large ?
  cache ?
    size → int
      fixed ?
  [ small && cache ⟹ fixed ]

abstract comp
  version → int

abstract ECU : comp

abstract display : comp
  server → ECU
  ‘options // shorthand for options : options
  [ version ≥ server.version ]

abstract plaECU : ECU
  plaDisplay : display 1..2
    [ !cache
      server = parent ]
```

```
ECU1 : plaECU

ECU2 : plaECU ?
  master → ECU1

// feature model for the specific PL

telematics
  xor channel
    single ?
    dual ?

  extraDisplay ?

  xor size
    small ?
    large ?

  [ dual ⇔ ECU2
    extraDisplay ⇔ #ECU1.plaDisplay = 2
    extraDisplay ⇔ (ECU2  ⟹  #ECU2.plaDisplay = 2)
    large ⇔ !plaECU.plaDisplay.options.size.small
    small ⇔ !plaECU.plaDisplay.options.size.large ]

// concrete product
[ dual && extraDisplay && telematics.size.large ]
[ all c : comp | c.version = 1]
```