Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Clafer: Unifying Class and Feature Modeling

Kacper Bąk · Zinovy Diskin · Michał Antkiewicz · Krzysztof Czarnecki · Andrzej Wąsowski

Received: date / Accepted: date

Abstract We present *Clafer* (class, feature, reference), a class modeling language with first-class support for feature modeling. We designed Clafer as a concise notation for meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models and meta-models via constraints (such as mapping feature configurations to component configurations or model templates). Clafer allows arranging models into multiple specialization and extension layers via constraints and inheritance. We identify several key mechanisms allowing a meta-modeling language to express feature models concisely. Clafer unifies basic modeling constructs, such as class, association, and property, into a single construct, called *clafer*. We provide the language with a formal semantics built in a structurally-explicit way. The resulting semantics explains the meaning of hierarchical models whereby properties can be arbitrarily nested in the presence of inheritance and feature modeling constructs. The semantics also enables building consistent automated reasoning support for the language: to date, we implemented three reasoners for Clafer based on Alloy, Z3 SMT, and Choco3 CSP solvers. We show that Clafer meets its design objectives using examples and by comparing to other languages.

Keywords Language Design · Feature Modeling · OOM · Semantics · Unification

CR Subject Classification D.2.1 [*Software Engineering*]: Requirements/Specifications—languages

Kacper Bąk
GSD Lab, University of Waterloo, Canada
E-mail: kbak@gsd.uwaterloo.ca

Zinovy Diskin
GSD Lab, University of Waterloo, Canada
E-mail: zdiskin@gsd.uwaterloo.ca

Michał Antkiewicz
GSD Lab, University of Waterloo, Canada
E-mail: mantkiew@gsd.uwaterloo.ca

Krzysztof Czarnecki
GSD Lab, University of Waterloo, Canada
E-mail: kczarnec@gsd.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen, Denmark
E-mail: wasowski@itu.dk

1 Introduction

Both feature and meta-modeling have been used in Software Product Line (SPL) engineering to model variability. Feature models can be seen as tree-like menus of mostly Boolean, sometimes also numerical and textual, configuration options, augmented with cross-tree constraints [45]. These models are typically used to show the variation of *user-relevant* characteristics of product variants within a product line. In contrast, meta-models, usually represented as class models as expressed using Meta Object Facility (MOF) [54], specify concepts representing more detailed aspects of products; including behavioral and architectural aspects. For example, meta-models are often used to specify the component and connector types of *product line architectures* and the valid ways of connecting 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* [21,6], (ii) proposals to unify feature modeling notations into a useful

and widely applicable subset [58], and (iii) attempts to express feature models as class models in the Unified Modeling Language (UML) [16,23]. In fact, a number of practitioners use UML-based representations to model variability [11]. 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* [19,37,38,32]. Cardinality-based feature modeling achieves this by extending feature models toward class modeling by introducing multiple instantiation and references. It is incomplete, however, because it does not offer inheritance. Class modeling, which natively supports multiple instantiation, references, and inheritance, enables feature modeling by a stylized use of containment (UML's composition) and the profiling mechanisms of MOF or UML (e.g., as in [34]).

Both developments have notable drawbacks, however. An important advantage of feature modeling as originally defined by Kang et al. [45] is its simplicity; several respondents to a recent survey confirmed this view [46]. 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 be hardly called feature models in the original sense; they are more of class models rather than easy-to-use menus guiding configuration decisions. On the other hand, whereas the model parts requiring multiple instantiation and references are naturally expressed as class models, the parts that have feature-modeling nature cannot be expressed simply in class models, but rather clumsily simulated using composition hierarchy and certain modeling patterns. Even worse, such a solution requires inconvenient model refactorings, while according to a recent survey [11], *evolvability of variability models is one of the main challenges faced by practitioners.*

We present *Clafer* (class, feature, reference), a language that unifies feature and class modeling. It can naturally express feature models, while allowing for full class modeling and relating both types of models. Clafer supports: (i) class-based meta-models, (ii) object models, (iii) partial object models (with uncertainty), (iv) feature models with attributes and multiple instantiation, (v) full and partial configurations of feature models, (vi) mixtures of meta- and feature models and model templates [18], (vii) first-order logic constraints. Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. On the other hand, by designing Clafer we wanted to create a language that builds upon as few concepts as possible and that is easy to learn. The main principle guiding our design was that "simple things (feature models and simple constraints) should be natural and simple while complex things (meta-models and complex constraints) should be possible".

In this paper, we present several contributions.

1. We identified several key mechanisms allowing a meta-modeling language to express feature models concisely; particularly, concept unification, instance composition and type nesting, and default singleton multiplicity.
2. We unify basic constructs of structural modeling: *class*, *association*, and *property* (which includes *attribute*, *reference*, and *role*), into a single construct, called *clafer*. Such a construct has the characteristics of a class (ability to nest and inherit properties and other classes in it), an association (ability to navigate over it), and an attribute (ability to store single or multiple (set, bag) values of primitive types in it).
3. Rich semantic capabilities are packed into very compact syntax that makes class models concise. If necessary, concrete uses of Clafer could introduce syntax to distinguish among the various interpretations of clafers, such as features, classes, components, as well as, any domain-specific concepts.
4. We provide a mathematical model of Clafer's syntactical mechanism (meta-models and the overall architecture) using *formal class diagrams* [29], which is a structural modeling formalism based on category theory and diagrammatic logic [31]. The formalism has allowed us to define semantics concisely by specifying mappings between artifacts and operations over the artifacts and mappings. The structures of the syntactic and semantic domains are aligned and made explicit rather than flattened and hidden in a multitude of first-order logic formulas.

The main benefit of semantic unification is simplification of analyses and tools that operate both on feature and class models, and mappings between them. The language is supported by tools for model analyses including consistency checking, instantiation and instance completion [48,49], and single- and multi-objective optimization [53]. Analyzes are performed by translating Clafer models into the input language of the underlying backend solvers (Alloy relational logic solver [40], Z3 Satisfiability Modulo Theories (SMT) solver [26], and Choco 3 Constraint Satisfaction Problems (CSP) and Constraint Programming (CP) solver) [44]. The results of the analysis are translated back to Clafer. We also provide a web-based suite of tools for working with Clafer models and the backend solvers [3,52].

In the industrial context, support for variability is needed when modeling product-line architectures using such standards as AUTOSAR [56], EAST-ADL [17], and SysML [35]. These standards do not require the

unification of feature and class modeling; however, they point to the need of having both feature and architectural models in a single, integrated system specification. In fact, the first level of an EAST-ADL specification is the *technical feature model* and AUTOSAR defines a feature model interchange format [57] allowing for integration of external feature modeling tools and support for adding variation points into AUTOSAR models. The thesis by Padilla Gaeta demonstrates techniques for adding variability to SysML models [33]. Furthermore, formal reasoning, such as constraint checking and propagation, over such integrated specifications is required, which in turn, necessitates the integration of the modeling languages and the supporting reasoners. In contrast, by unifying feature and class models in Clafer, we provide unified reasoning support for such integrated system specifications. For example, the thesis by Murashkin demonstrates that Clafer can express and optimize complex automotive electronic/electric architectural models that span three abstraction levels of an architecture modeling standard EAST-ADL [51].

The paper is organized as follows. We introduce our running example in Sect. 2. We define (i) the challenges of representing the example using either only class modeling or only feature modeling, (ii) the challenges of mapping feature configurations to component and option configurations, and (iii) a set of design objectives for Clafer in Sect. 3. We then present Clafer in Sect. 4 and show that it naturally supports unified feature and class modeling. Section 5 explains the semantic foundations of Clafer, and shows how it implements the unification idea. Section 6 describes Clafer syntax design in more detail, and shows that a Clafer model is a hierarchical view on a formal class diagram. We evaluate the language analytically in Sect. 7 and demonstrate that it satisfies the design objectives. We conclude in Sect. 9, after having compared Clafer with related work in Sect. 8. Our work is supported with a set of appendices following the main body of the paper.

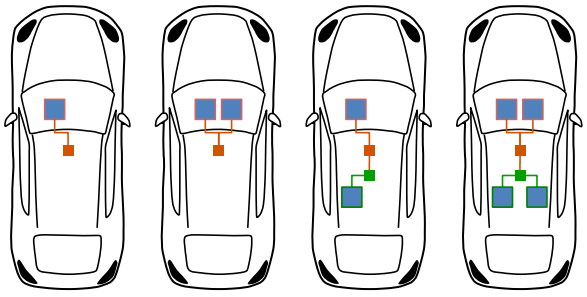
2 Modeling Variability in SPLs: An Example

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. 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. 1a has only one display for the driver. The car in Fig. 1b has two displays: one for the driver and one for the front-seat passenger; both displays are controlled by the same ECU. The car in Fig. 1c has one display in the front and one in the back; the displays are controlled by separate ECUs. The car in Fig. 1d has a separate display for each person in the car.

Figure 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. 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 numerical or textual attribute) that can be either selected or excluded when configuring a concrete product. 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. 1c); however, we can add one extra display per channel (as in Fig. 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) 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. 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. 2c). We can also specify the cache size and decide whether it is fixed or can be updated dynamically. Thus, the solution space should in-



(a) (b) (c) (d)

Fig. 1: Sample configurations of a telematics system

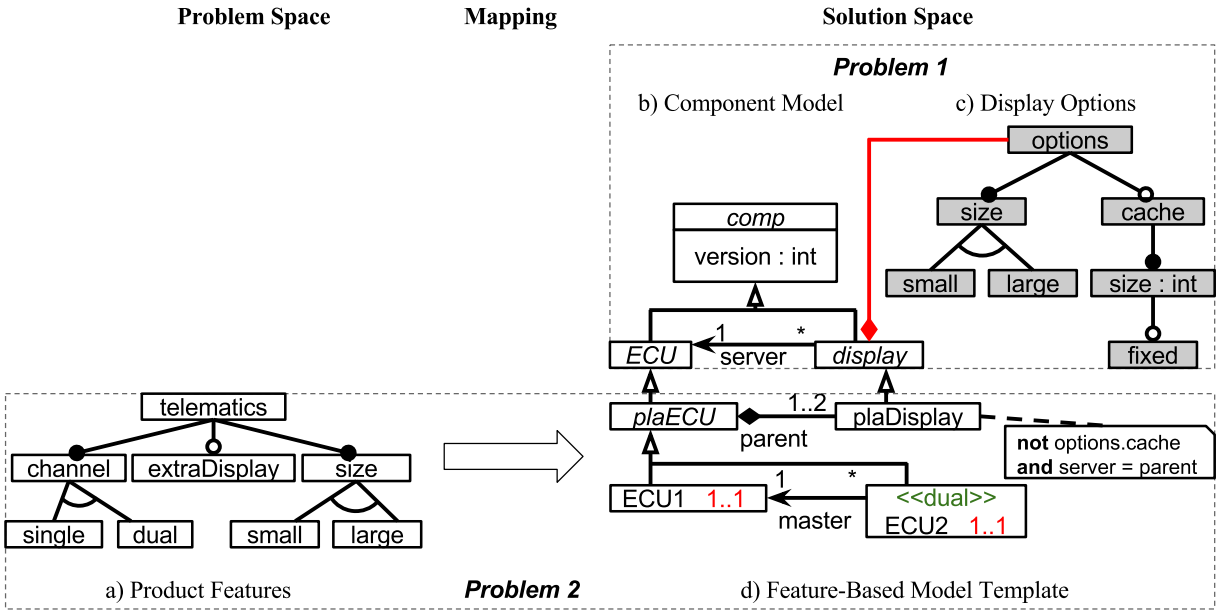


Fig. 2: Telematics product line

clude a class model of component types and a feature model of component options.

Moreover, the component meta-model in Fig. 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 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 [14]).

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) [18]). In Fig. 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. 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. Thus, FBMTs relate the problem-space feature configurations to the solution-space component and option configurations. A block-arrow in Fig. 2 represents this mapping. We will provide a precise specification of the complete mapping later in this paper.

The example motivates two issues of modeling variability in SPLs: 1) the necessity to merge feature and class models in a single solution space, and 2) 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* and *Problem 2*.

3 Two Problems

3.1 Problem 1: Merging Feature and Class Models

The solution space in Fig. 2 contains a class-based meta-model and a feature model. To capture our intention, the models are connected via UML composition. As

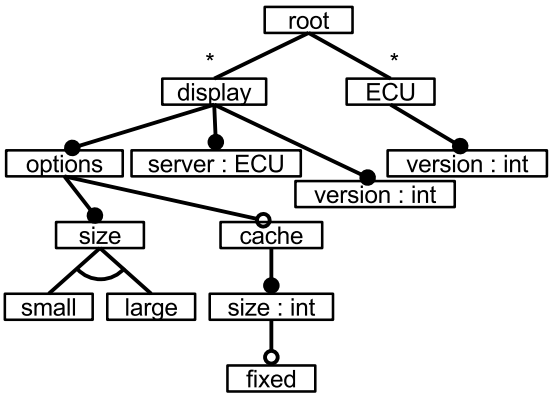


Fig. 3: Cardinality-based feature model of components

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.

3.1.1 Class Modeling via Feature Modeling

Figure 3 shows the part of the model which represents components. The model introduces a synthetic root feature; display and ECU can be multiply instantiated (as indicated by the multiplicity *), but they cannot be *abstract*; and display has a server subfeature representing a reference to instances of ECU. Two subfeatures version are added to display and ECU to match the meta-model in Fig. 2b since feature models do not support inheritance. Extending cardinality-based feature modeling with inheritance would bring the 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 [45].

3.1.2 Feature Modeling via Class Modeling

Figure 4 shows only the display option model, as the component model remains unchanged (as in Fig. 2b). A

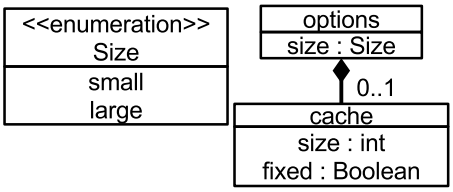
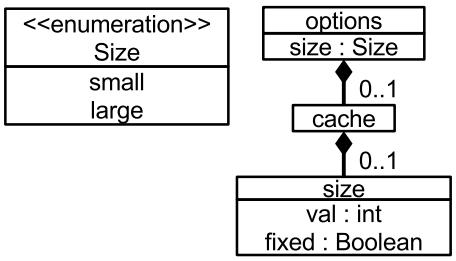
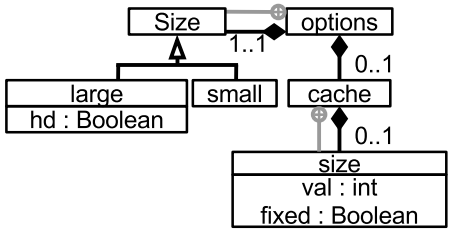


Fig. 4: Meta-model of display options



(a) Iteration 1



(b) Iteration 2

Fig. 5: Evolved meta-model of display options

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.

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

1) 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 Object Constraint Language (OCL) [55] constraint expressing the dependency but hiding an important structural information within constraints is not generally advisable.

A better solution would be to *reify* the attribute size as a class contained in cache as shown in Fig. 5a, in which the structural dependency is explicit. Note also that in some cases, reification of attributes would natu-

rally be modeled using 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. 5b, which is again a substantial refactoring of the initial class model from Fig. 4. In contrast, adding the property hd to the feature model in Fig. 2a amounts to plain nesting the feature hd under the feature large.

These examples show a general drawback of ordinary class modeling in the context of gradual model development. Modeling properties by attributes is compact, but disallows 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 be a problem for an inexperienced modeler.

2) 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. 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. 5b), which would further complicate the model.

3) 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. 2a) may become 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. 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, as well as, to most object-oriented programming languages, such as Java and C++.

3.2 Problem 2: Mapping Features to Component Configurations

Relating heterogeneous models by a mapping is a non-trivial task. For example, a FBMT in Fig. 2 relates a feature model, a class model (of components), and (implicitly) their meta-models. 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 [24].

3.3 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:

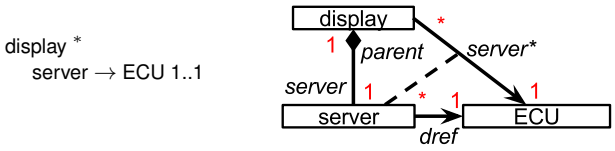
1. *Provide a concise notation for feature modeling*
2. *Provide a concise notation for class modeling*
3. *Allow mixing of feature models and class models*
4. *Use a minimal number of concepts and have a uniform semantics*

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. 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, as well as, easy evolving of feature models towards class models, if needed; 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. The next section presents Clafer—the language designed to meet these requirements.

4 Clafer vs. the Two Problems

4.1 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 construct called *clafer*. Throughout the paper, if the word Clafer begins with upper case, then it refers to the language, otherwise it refers to the unifying concept or the corresponding construct, or to a syntactical unit—a model expressed using Clafer is built from clafers. For example, Fig. 6a 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 (display, implicitly, is nested under the *synthetic root* clafer, i.e., ancestor of all clafers). A



(a) Clafer model (b) Rendering as a UML Class Diagram

Fig. 6: Clafer model and its meaning

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

Fig. 7: Feature model of component options in Clafer

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

The clafer declaration (server) specifies a relationship between its parent class (display) and its target class (ECU), and it declares the following (cf. Fig. 6b):

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 *dref* (dereference) 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 server). By default, we assume that for any instance this of class display, the equality this.*server*.*dref*=this.*server** holds. In fact, this condition means that the class server can be considered as a *reification* of the association *server** in the sense of UML's association classes. Figure 6b uses the UML syntax for representing an association class (the dashed line from the association server* to the class server) to indicate this fact.

If a clafer has no reference (i.e., has neither class ECU nor maps *dref* 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*. We make these concepts more precise in Sect. 5.

4.2 Solving Problem 1: Merging Feature and Class Models

Let us model the running example using Clafer. In general, a Clafer model consists of three types of constructs: *clafers*, *constraints*, and *objectives*. In this paper, we are only concerned with *clafers* and *constraints*; *objectives* are used in multi-objective optimization and they are discussed elsewhere [53,51]. Each line in Fig. 7 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: the clafer options is at the top level; the clafers size (line 2), cache, and the constraint are nested under options; the clafers small and large are nested under size; etc. Below we discuss the Clafer model step by step. We provide the full example in Appendix F for reference.

4.2.1 Feature Modeling

The Clafer model in Fig. 7 corresponds to the model of display options in Fig. 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 inheriting from 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: an instance of a *child* clafer cannot exist unless an instance of its *parent* also exists. The clafer size → int corresponds to a feature with an attribute of type integer; it also nests another clafer

```
1  abstract comp
2    version → integer
3
4  abstract ECU : comp
5
6  abstract display : comp
7    server → ECU
8    'options // shorthand for options : options
9    [ version ≥ server.version ]
```

Fig. 8: Component meta-model in Clafer

fixed (cf. Fig. 2b). If cache is eliminated, then its children size → int and fixed are eliminated too.

4.2.2 Constraints

Constraints express dependencies among clafers or restrict numerical and textual values. For example, the constraint in line 8 requires that a small display with cache must have the cache of fixed size. The clafer small is found within size; the full path inserted by the compiler will be this.size.small. Clafer constraints were inspired by Alloy. The latter notation is elegant, concise, and expressive enough to restrict both feature and class models. Similarly to Alloy and OCL, constraints in Clafer can be either declared at the top-level or nested under a clafer. Top-level constraints are global, in a sense that they must hold for every instance of the model. Nested constraints must only hold for every instance of the context clafer (i.e., the clafer they are nested under). We define Clafer constraints in Appendix D.

Each clafer introduces a namespace. For example, the two different clafers named size exist in different namespaces (one within options, and one within cache). 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 or parent. 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 it is ambiguous within one rule, an error is reported.

4.2.3 Class-Based Meta-Modeling

Figure 8 shows a component meta-model (from Fig. 2b) encoded in Clafer. Clafer version (line 2) corresponds to the attribute of the class *comp* in Fig. 2b; and clafer server (line 7) corresponds to the unidirectional association pointing to the class ECU in Fig. 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*, 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 1 summarizes Object-Oriented Modeling (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. 8, 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 *non-overlapping* (disjoint) and *covering* by default.

Quotation (cf. 'options in Fig. 8) 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 effectively include options from Fig. 7 as a part of display in Fig. 8.

While inheritance is about sharing a supertype, reference clafers enable sharing instances. An instance of the reference clafer server will point to some instance in the clafer ECU. In general, there may be several instances of the clafer display that will reference the same instance of ECU as their 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. 3.

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

Table 1: Corresponding constructs in OOM and Clafer

```
abstract plaECU : ECU
  plaDisplay : display 1..2
    [ no cache ]
    [ server = parent ]

ECU1 : plaECU

ECU2 : plaECU ?
  master → ECU1
```

Fig. 9: Architectural template in Clafer

4.3 Solving Problem 2: Mapping Feature to Component Configurations

Clafer can encode model templates in a way that configurations are always syntactically correct. Figure 9 encodes the template from Fig. 2d in Clafer. The predefined keyword parent points to one of the instances of *plaECU*, which is either ECU1 or ECU2. Instead of making existence of a class optional, it is assumed that the class exists, but it has the 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.

Having defined an architectural template, we can expose the variability points present in it as a product-line feature model. Figure 10 shows this model (cf. Fig. 2a) along with constraints coupling its features to the variability points of the template. The template in Fig. 9 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 11 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. 1d. Instance generators [3] can automatically instantiate the product line by deriving a configuration of the architectural template as shown in Fig. 12.

Clafer offers the same syntax for specifying both models and instances (configurations), and the latter can be partial. Figure 12 shows a Clafer model that encodes exactly one configuration that was previously

```
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 ]
```

Fig. 10: Feature model with mapping constraints

specified by constraints. The encoding is done by hierarchical *redefinition* among clafers — subclassing among clafers and subclassing among references. For example, in Fig. 9 the clafer plaDisplay is nested under *plaECU*, thus in Fig. 12 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. 9 the clafer master points to ECU1 and is nested un-

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

Fig. 11: Constraints specifying a single product

```
t1 : telematics
  c1 : channel
    d1 : dual
  ed1 : extraDisplay
  s5 : size
    l5 : large
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
```

Fig. 12: A sample configuration (instance) in Clafer

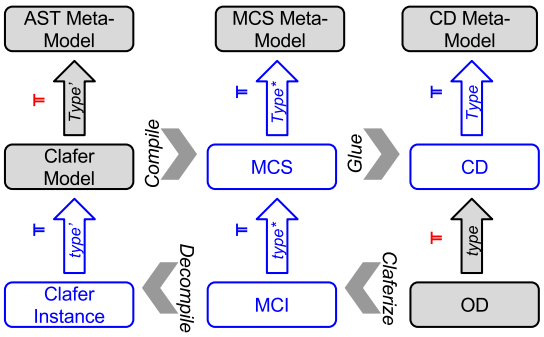


Fig. 13: Architecture of Clafer syntax and semantics

der ECU2; in Fig. 12 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.

This capability of seamlessly expressing abstractions (model) and examples (instances) in a single notation is critical for effective *example-driven modeling* [9,2].

5 Anatomy of a Clafer Model and Its Instantiation

This section discusses the basic ingredients of Clafer syntax (details in Appendix C) and presents instantiation of Clafer models which plays an important role in model analyses. 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. Furthermore, we give semantics to Clafer models via instantiation (cf. Fig. 13).

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 shown in Fig. 13. In the figure, the rounded rectangles represent artifacts (e.g., Clafer Model), the arrows *type* represent typing mappings (e.g., *Type'*), and the chevrons represent transformations (e.g., *Compile*). Another convention used in figures througout the paper is that shaded shapes are assumed to exist, whereas blank shapes are assumed to be fully derived.

Figure 13 illustrates that a Clafer Model is typed over and required to conform to the Abstract Syntax Tree (AST) Meta-Model (cf. the constraint $\models$). Then, the Clafer Model is compiled into an intermediate representation, a Multi-Clafer Shape (MCS), which is typed over and created so that it conforms to the MCS Meta-Model (cf. the constraint $\models$). The MCS structurally resembles the Clafer Model and it is not yet a class diagram as class names can repeat when classes play different roles. Therefore, the MCS needs to be transformed into a Class Diagram (CD), which is typed over and created so that it conforms to the CD Meta-Model. Roughly, the transformation glues classes with the same name playing different roles into single classes.

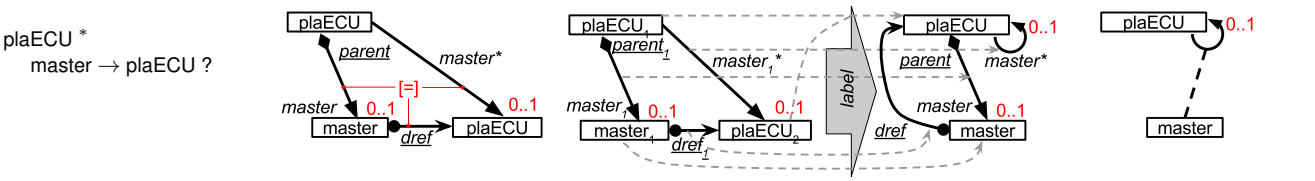
At this point, the class diagram can be given to a backend reasoner for instantiation, which creates an Object Diagram (OD) typed over and required to conform to the class diagram CD. Now, the object diagram must be "claferized", that is, transformed into an instance Multi-Clafer Instance (MCI) typed over and conforming to the MCS. In the MCI, objects are replicated so that they can appear in the original positions, as before the gluing. The instance MCI can now be decompiled into a Clafer Instance. Note, that the Clafer Instance is transitively typed over the AST Meta-Model, which explains why instances in Clafer have the same notation as the Clafer models - they have the same abstract syntax. For example, compare the clafer options from Fig. 7 and its instances o1-o4 in Fig. 12, lines 10, 16, 25, and 31, respectively.

In our toolchain [3], the Clafer compiler first parses a textual Clafer model into its abstract syntax tree Clafer Model and then compiles it into an MCS. Next, the compiler transforms the MCS into an encoding of a class diagram CD in the language of the chosen backend solver, such as, Alloy, SMT-LIB, and Choco 3. The backend solver then produces object diagrams OD which are instances of the class diagram CD. Finally, these ODs are translated back to Clafer syntax.

5.1 Clafers as Views onto Class Diagrams

Figure 14a 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 14d 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. 14b by an MCS that follows the concrete syntax of Clafer. MCS defines Clafer models in terms of *formal class diagrams* [29] (formal CDs or just CDs for short), which we use as a notation for our semantic domain. In general, an MCS is a tree-like structure composed by joining Clafer Shapes (CSs). A single CS represents a single clafer declaration. In the example, the MCS is composed of only one CS.

A formal CD is a graph with additional labels encoding constraints. For the CD in Fig. 14b, the graph encompasses three nodes and three edges, and the constraint labels denote multiplicities and a commutativ-



(a) Clafer model (b) MCS (c) Labeling as a graph mapping (d) UML Class Diagram

Fig. 14: (a) A Clafer model, (b) its compilation, (c) label extraction, (d) the rendering using reified association

ity constraint ([=]). 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 *dref*. In fact, this equality means that class master together with its pair of associations (*parent*, *dref*) can be considered as reification of association *master** (which is shown by a dashed line in the UML diagram in Fig. 14d).

We use the following notation for maps (edges). Predefined maps, e.g., *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. 14b. An open arrow head (e.g., arrow *dref*) means that the map is *single-valued*: each instance of master points to at most one instance of plaECU. A black bullet arrow tail means that the map is *total*: each instance of master points to at least one instance of 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 instance of plaECU for an instance of master) and *existence dependency* (deletion of an instance of plaECU implies the deletion of its nested instance of master). The first condition is often referred to as *non-sharing* (and the 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 Class Diagrams with dynamic predicates described in [30]. Our arrow notation for maps is summarized in Tab. 2. Table 3 summarizes the diagram predicates we use. They are formally defined in Appendices A and B. We mark a non-constrained bag-valued mapping with the label [bag] while being set-valued is assumed by default and we 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).

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

Table 2: Notational conventions for maps

Notice, however, that the MCS in Fig. 14b 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 MCS) distinguishes two different *roles* that instances of the class plaECU can play: (i) being the parent of reference master, and (ii) being the target of the reference. What Fig. 14b actually encodes is a *mapping* from a diagram of *roles* to a diagram of classes and associations, as shown in Fig. 14c. The source of the mapping is the carrier graph of the diagram from Fig. 14b. The target is a formal class diagram that makes the meaning of the class diagram from Fig. 14d precise. Indeed, as an object of class master is supposed to reify a *master* link, such an object must have 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. 14c, these projection references are given respectively by *parent* and *dref* associations (maps) in the target CD.

The mapping specified in diagram Fig. 14c (*label*) 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 associa-

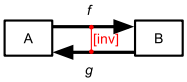
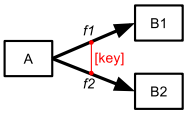
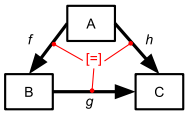
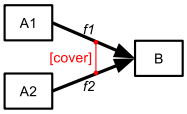
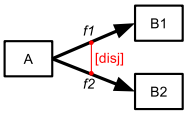
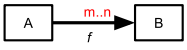
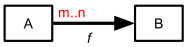
Predicate name/symbol	Shape	Intended semantics (elements a, a', b, b' range over A, B resp.)
inv		maps f, g are mutually inverse iff their spans f^*, g^* are such.
key		$f_i(a) = f_i(a')$ for both $i = 1, 2$ implies $a = a'$.
=		$f^*.g^* = h^*$
cover		for any $b \in B$ there is $a \in A_i$ s.t. $b \in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

Table 3: A signature of diagram predicates (the labels [bag] are omitted)

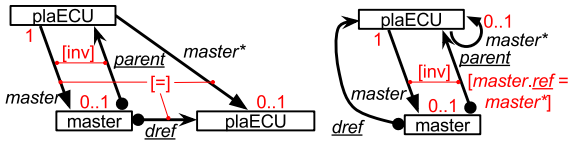
tion *master*. Note that the mapping preserves the graph structure: it maps edges to edges so that their sources and targets are respected. This preservation is an important condition to be respected by labeling.

We will say that Fig. 14c describes a view on the class diagram, and call the mapping *label*. For a complex Clafer model consisting of multiple clafers, the role graph has the shape of multiple triangles joined together into a hierarchical structure (see, for example, Fig. 17c). Thus, a Clafer model is compiled into an MCS, whose labeling encodes a mapping to a class diagram. We will again call this mapping *label*, and say that it is *extracted* from the MCS. The mapping *label* is crucial for claferizing and decompiling of object diagrams that instantiate the back-end class diagrams into instances typed over MCS, in which different roles played by the same object are explicit.

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

5.2 Clafer Shape

The diagram in Fig. 15a is a more detailed representation of the diagram in Fig. 14b. 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



(a) Clafer Shape (b) Class Diagram

Fig. 15: The CS and the corresponding CD of master

inverse (the predicate declaration [inv](*master*, *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 *dref* 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 single clafer declaration is compiled into a specific DP-graph (formal CD) of a specific shape specified in Fig. 16. We call this specific DP-graph a 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 decla-

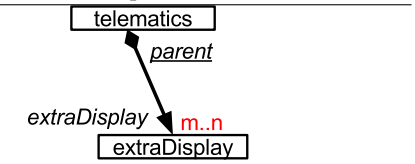
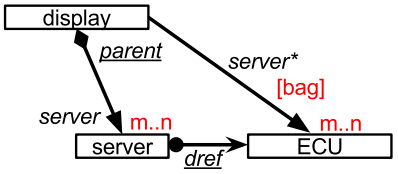
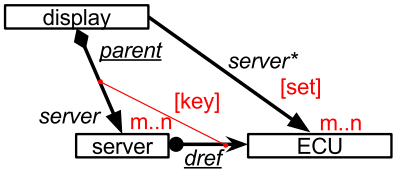
Clafer Kind	Clafer Model	Clafer Shape
Basic	telematics extraDisplay m..n	
Reference bag	display server ↠ ECU m..n	
Reference set	display server → ECU m..n	

Table 4: The meaning of a clafer declaration

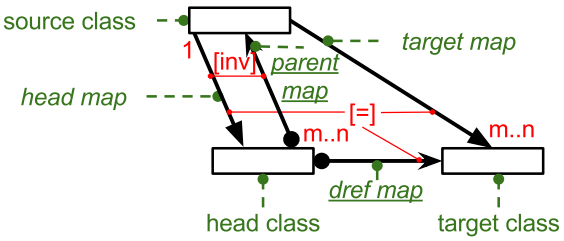


Fig. 16: Clafer Shape (CS) (without labels)

ration; the *source* class relates the introduced clafer to its parent in the containment hierarchy. The *dref* map and the *target* indicate the target type of the reference. Note that mappings *dref* and *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).

5.3 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 summarizes the discussion and shows sample models and their CSs. 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.

5.3.1 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. A distinction of a basic clafer's shape is that the maps *dref* and *target* and the *target* class are excluded.

5.3.2 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 illustrates a compilation of the reference bag clafer server to a corresponding CS. In fact, reference bag clafers follow the structure of basic clafers, but additionally have the *target* class (cf. Fig. 16). In reference bag clafers, the *target* map is bag-valued, hence the annotation [bag] on *server**.

5.3.3 Reference Set Clafers

They are set-valued references and their name is followed by the arrow symbol (→). 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 CSs is similar, but the latter have an additional predicate declaration [key](*parent*, *dref*) (cf. the last row of Tab. 4). The predicate means that each instance this of the *head* class is identified by a pair of instances

(this.*parent*, this.*dref*) 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.

5.3.4 Abstract Clafers

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

5.3.5 Predefined Clafers

There are several clafers that are predefined in the language: the clafer Sing and a family Dom of clafers representing primitive domains (e.g., int for integers, and string for strings of characters). Sing is very important although it does not appear in the concrete syntax. Each Clafer model by default has Sing as the root of the containment hierarchy, and hence 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. Sing only has a *head* class, which is some predefined singleton class $\{\_^{*}\}$ also called Sing and has neither a parent nor a target (see the upper triangle in Fig. 17b). Any clafer in Dom is a child of Sing, its *head* class is the class of the respective values (integers, strings, etc.), and it does not have a *target* class.

5.4 Clafer Nesting

A Clafer model is a tree of clafer declarations. In concrete syntax, that hierarchy is expressed via indentation. For example, master is a child of plaECU in Fig. 17a. 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.

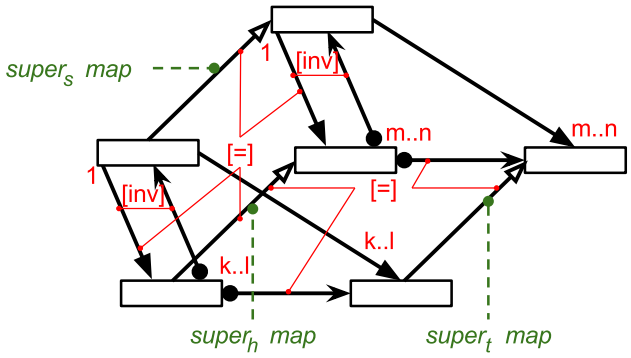


Fig. 18: Inheritance among two CSs

Clafer nesting is realized through cojoining CSs which then form an MCS. If a clafer B is a child of (nested under) clafer A, then the *head* class of A is also the *source* class of B, so that the CS of B is plugged into the CS of A such that:

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

For example, Fig. 17b shows three cojoined CSs where plaECU is a parent of master and Sing is a parent of plaECU.

Section 5.1 showed that a single CS amounts to a *labeling mapping* to a CD. Correspondingly, an MCS amounts to a *labeling mapping* to a bigger CD. Figure 17c presents the extraction of this mapping for our example. The mapping itself is specified in Fig. 17c'. The CD, generated by the mapping *label*, demonstrates that labeling glues together some nodes from the MCS.

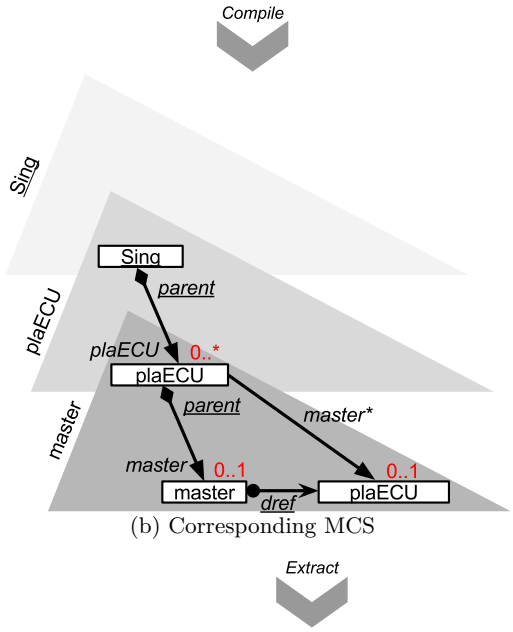
5.5 Inheritance

Inheritance between two clafers is defined at the level of their CSs and means their inclusion: the corresponding classes in CS are related by inclusions, as in Fig. 18. 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 19 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 a CS has no target (is a basic clafer) or no parent (an abstract clafer), then there is no inclusion between the missing classes.

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

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

(a) Clafer model

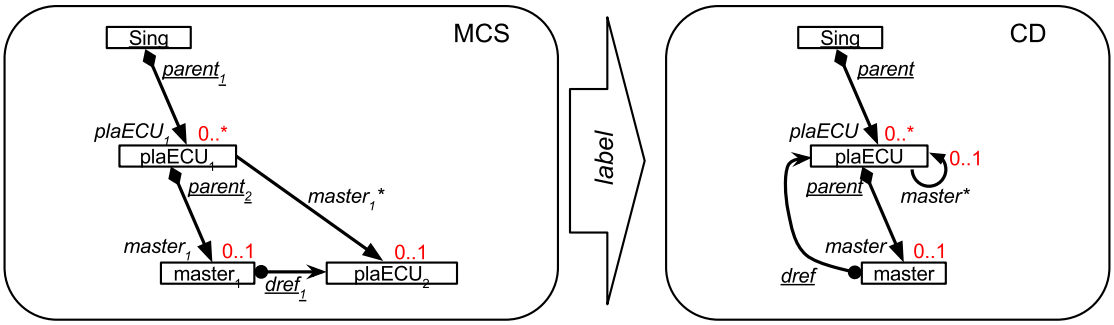


(b) Corresponding MCS

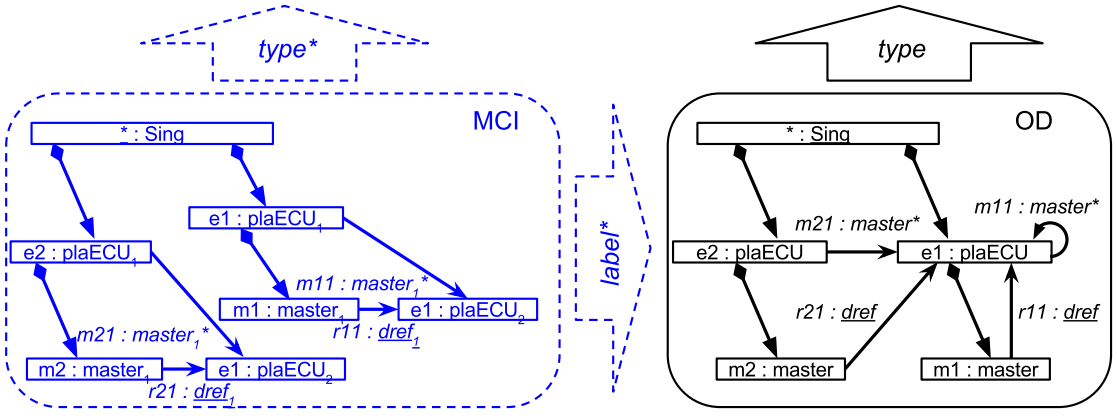
ClassRole	Label
Sing	Sing
$plaECU_1$, $plaECU_2$	plaECU
$master_1$	master

MapRole	Label
$parent_1$	$parent_1$
$plaECU_1$	$plaECU$
$parent_2$	$parent_2$
$master_1$	$master$
$master_1{}^*$	$master^*$
$dref_1$	$dref$

(c') Definition of the mapping *label*



(c) Corresponding MCS mapped via *label* defined in (c') to a CD



(d) An MCI mapped via *label** to a sample OD.

Fig. 17: Compilation of a Clafer model to an MCS, gluing into a CD, and instantiation

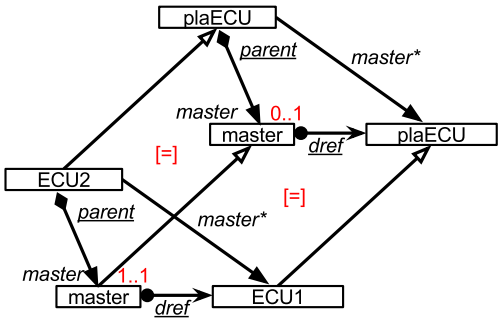
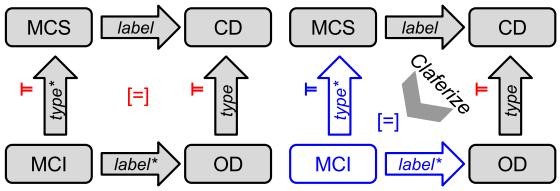


Fig. 19: An example of clafer inheritance



(a) Definition (b) Derivation

Fig. 20: Clafer model instance

– 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 $B_1, \ldots, B_n$, then the constraints [cover] and [disj] are declared in the MCS on the $super_h$ maps relating *head* classes of $B_1, \ldots, B_n$ with their super classes. That way all instances of an abstract clafer must be instances of its concrete subtypes. Finally, observe that the commutativity constraints enforce proper inclusions of mappings as well. For example, in Fig. 19, the maps *master* and *dref* of the inheriting clafer (master under ECU2) are included in the corresponding maps of its super clafer (master under plaECU).

5.6 Instantiation

Every Clafer model (e.g., Fig. 17a) is transformed to a class diagram CD, such as the one in the right part of Fig. 17c. An instance of the class diagram CD is an object diagram OD typed over the CD. A sample instance is shown in the right part of Fig. 17d. The instance, however, needs to be *unfolded* into a multi-clafer-shaped OD typed over given MCS, as shown in the left part of Fig. 17d. We call such a multi-clafer-shaped OD an Multi-Clafer Instance (MCI). Elements of an MCI 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 an MCS, the same object can play different roles in an MCI. For example, the object e1 from the example in Fig. 17 plays three roles: an independent plaECU (shown by its position at the role box e1:plaECU), the master of object e2 (e1:plaECU), and the master of itself (e1:plaECU). The mapping *label** maps roles in an MCI to the OD's objects and links that perform these roles.

Figure 20a gives a general definition of an MCI. As a Clafer model is a hierarchical view on a class diagram, which is defined by a labeling mapping *label*: MCS→CD, we consider an instance of a Clafer model to be a *structurally similar* hierarchical view onto an object diagram OD instantiating CD. That is, a *Clafer model instance* is a graph MCI typed over the graph MCS and labeled by elements of the OD (via mapping *label**: MCI→OD) so that the square diagram in Fig. 20a commutes.

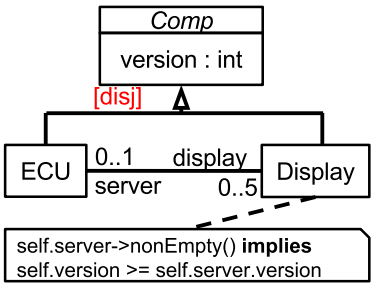
Moreover, for a given *label*: MCS→CD and any instance OD of the CD (shaded elements in Fig. 20b), it is possible to generate a correct MCI and mappings *type** and *label** (blank elements with blue frames) by applying to them an operation *Claferize* (shown by a chevron). There is a unique such MCI. In Sect. 6.3 we will specify how this operation works.

6 Formal Semantics

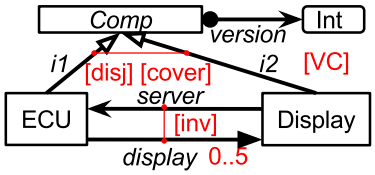
We will specify Clafer's syntax and semantics using formal class diagrams (CDs), a.k.a. DP-graphs, as our meta-meta-notation (which we have already briefly discussed in Sections 5.1, 5.2). That is, we will treat all models and meta-models involved as formal CDs, and mappings between them as structure-preserving 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. 6.2), and finally consider instantiation (Sect. 6.3).

6.1 Formal Class Diagrams and Their Instantiation

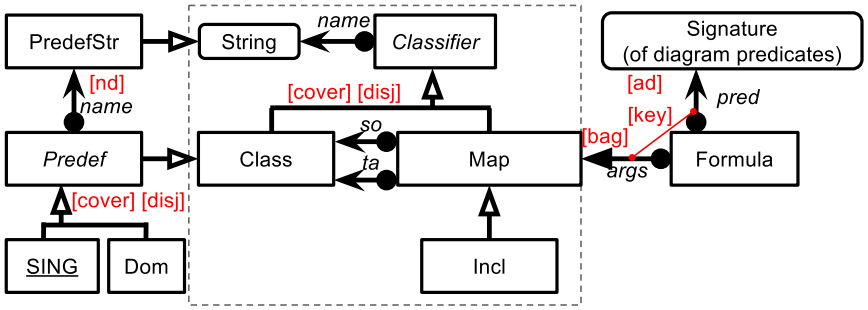
Figure 21a presents a simple UML class diagram D. An abstract class *Comp* has two disjoint subclasses (note the label [disj]), which are interrelated by a bidirectional association. In addition, version numbers of displays and their servers must satisfy a constraint [VC]: "the



(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.)

Fig. 21: Formal Class Diagrams: an instance in UML (a), formal CD rendering (b), and the meta-model (c)

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 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. 21b. 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 in Sect. 6.1.1, and then proceed to constraints in Sect. 6.1.2. Although multiplicities are constraints, we will discuss them in Sect. 6.1.1. In Sect. 6.1.3 we give a formal definition of formal CDs by specifying their meta-model. Formalization of the set-and-mapping semantics of CDs involves many details, which we present in Appendix B.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 B.1 supports this intuition with a system of formal definitions.

6.1.1 Instantiation of Formal CDs, I: The Graph Structure.

Nodes in F_D are to be interpreted as sets: $[\![Comp]\!]$, $[\![\mathsf{Int}]\!]$, $[\![\mathsf{ECU}]\!]$ etc. We will often say "a component" for an element of set $[\![Comp]\!]$, "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 (cf. Tab. 2). 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 of

Fig. 21b. 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. 21b 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 the 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 $e \in [\![\mathit{Comp}]\!]$ typed by $i1$, a link from $e \in [\![\mathit{Comp}]\!]$ 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 $t_O: G_O \to \Gamma[F_D]$ gives rise to a mega-mapping $[\![..]\!]_O: \Gamma[F_D] \to \textsc{SetMap}$ 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* [10].

6.1.2 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], and [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. 21a (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's *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 [47]. 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 (cf. Tab. 3). 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, $e \in [\![\mathit{server}]\!](d)$ iff $d \in [\![\mathit{display}]\!](e)$. 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 de-

clared 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

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

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. 21a, writing the name *Comp* in italic in the formal CD is a pure decoration without semantic meaning.

6.1.3 Meta-model

A meta-model of formal CDs is specified in Fig. 21c. It is itself a formal CD, $\mathsf{M_{CD}}$, and any valid formal CD should have a valid instance $[\![..]\!] : \mathsf{M_{CD}} \to$ 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. 21b. The latter is the following instance of the meta-model (we denote names of meta-classes in Small Capital font):

$$[\![\textsc{Class}]\!] = \{\#Comp, \#\mathsf{Int}, \#\mathsf{ECU}, \#\mathsf{Display}\},$$
$$[\![\textsc{Map}]\!] = \{\#version, i1, i2, \#server, \#display\},$$

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

$$[\![name]\!](\#\mathsf{ECU}) = \text{“}ECU\text{”} \in [\![\textsc{String}]\!],$$
$$[\![name]\!](\#display) = \text{“}display\text{”} \in [\![\textsc{String}]\!],$$
$$\dots,$$
$$[\![name]\!](i1) = [\![name]\!](i2) = \text{“}isA\text{”} \in [\![\textsc{String}]\!],$$

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

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

$$[\![so]\!](\#display) = \#\mathsf{ECU},$$
$$[\![ta]\!](\#server) = \#\mathsf{ECU},$$

and so on. And $[\![\textsc{Incl}]\!]=\{i1,i2\}$ 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 SING is a singleton with a predefined (but optional) instantiation by a class Sing, which, in turn, is instantiated (optionally) by a predefined object *. That is, for any formal CD F instantiating $\mathsf{M_{CD}}$, set $[\![\textsc{SING}]\!]_F$ is (either empty or) the same fixed singleton class $\{\mathsf{Sing}\}$; for any object diagram O instantiating F, $[\![\mathsf{Sing}]\!]_O$ is (either empty or) the same fixed singleton $\{*\}$. Meta-class SING is not instantiated in CD in Fig. 21b, but in formal CDs generated by the Clafer compiler, class Sing is always present as discussed in Sect. 5.3.

Similarly, the meta-class 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. 21b, $[\![\textsc{Dom}]\!]=\{\#\mathsf{Int}\}$, and instances of #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 Signature is instantiated by predicates, which can be used in constraint declarations. For the diagram in Fig. 21b,

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

although not all multiplicities are used. Meta-class 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 B.1.2). 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. 21b, 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. 6.1.2. In detail, the mapping *args* is bag-valued, and the indexing set for a formula ϕ (Appendix B.1.2) is the list of the arrows of the arity graph of predicate $pred(\phi)$; a precise formal definition can be found in Appendix B.1.2. 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. 21c are *(formal) CDs* or, synonymously, *DP-graphs*.

6.2 Formalizing Clafer Syntax

6.2.1 Architecture of Clafer's syntactical mechanism

We already presented the syntactical mechanism of Clafer in Fig. 13. There are three meta-models: Abstract Syntax Tree (AST) Meta-Model, Multi-Clafer Shape (MCS) Meta-Model, and Class Diagram (CD) Meta-Model. The MCS meta-model includes the AST meta-model and extends it with new properties of clafers. The MCS meta-model also includes the CD meta-model but names its elements differently. We describe the meta-models in Sections 6.2.2 and 6.2.3, respectively.

Formally, all nodes at the model and meta-model levels in Fig. 13 are CDs (i.e., DP-graphs) The vertical arrows 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 the entailment symbol $\models$.

6.2.2 Abstract Syntax Tree

The AST meta-model (cf. Fig. 22) specifies the abstract syntax of Clafer. The AST corresponds to a grammar of Clafer models (cf. Fig. 27, Appendix C). We discuss the meta-model starting from the left. Each clafer has a *name*, 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 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 $C \neq$ 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. 7, are simple constraints that relate a clafer with its children. There, the constraint c relates the

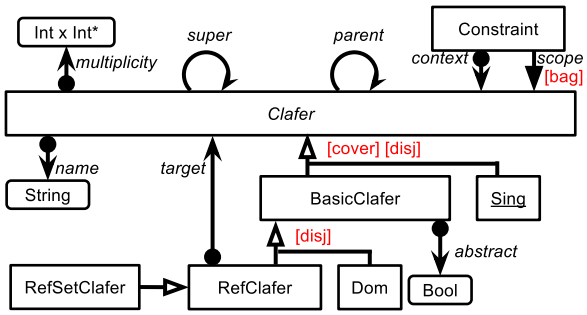


Fig. 22: Clafer AST Meta-Model

clafer size with its children small and large; formally:

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

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

6.2.3 Multi-Clafer Shape

The result of compilation is an MCS. The meta-model in Fig. 23 defines the structure of an MCS. As discussed in Sect. 5.2, in Fig. 16, 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 depending on the kind of clafer. The next four vertical arrows in the middle (*head*, *parent*, *dref*, *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. 17b amounts to the following instance of the meta-model:

$$\begin{aligned}
[\![\text{Sing}]\!] &= \#\#\underline{\mathsf{Sing}},\\
[\![\text{Clafer}]\!] &= \{\#\#\underline{\mathsf{Sing}}, \#\#\mathsf{plaECU}, \#\#\mathsf{master}\},\\
[\![parent]\!](\#\#\mathsf{plaECU}) &= \#\#\underline{\mathsf{Sing}},\\
&\dots\\
[\![\underline{\text{SING}}]\!] &= \#\underline{\mathsf{Sing}},\\
[\![source]\!](\#\#\mathsf{plaECU}) &= \#\underline{\mathsf{Sing}},\\
[\![head]\!](\#\#\mathsf{plaECU}) &= \#\mathsf{plaECU}_1\\
&\dots
\end{aligned}$$

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

MCS is augmented with diagram predicates over maps, which are represented by class Formula and are

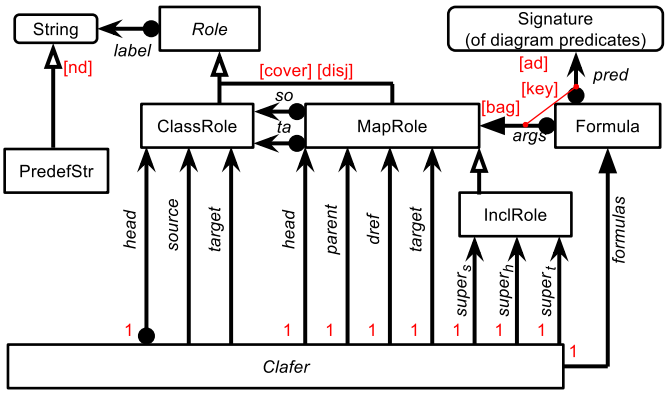


Fig. 23: MCS Meta-Model. It must satisfy the constraints from Tabs. 5–8.

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. 23 allows for any configuration of ClassRoles and MapRoles. They may form incorrect structures that are not CSs. The meta-model is also underconstrained with respect to containment and inheritance hierarchies of clafers (CSs may be cojoined incorrectly). Hence, we augment the metamodel with extra constraints to guarantee that its instances are valid MCSs. There are four groups of constraints, all are given by conditional equations.

1) Incidence Equations define a correct graph structure of a CS, i.e., the correct incidence of nodes and arrows. The constraints are specified in Tab. 5 in Appendix E: first, a description is given, then its formalization follows. (To ease reading the formulas, we qualify map names in the meta-model from Fig. 23 by their targets.) For example, the first row requires that the map *head_map* in CS has the source class as a source (*so*) and the head class as a target (*ta*).

2) Clafer Cojoining Equations specify the overlapping of CSs. Clafers overlap iff there is parent-child relationship between them, or when a clafer is another clafer's target. Table 6 in Appendix E 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 the child's CS.

3) Clafer Kind/Shape Discipline Equations specify the structure of CSs of different kinds of clafers. Table 7 in Appendix E 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 8 in Appendix E presents the naming constraints. For example, the equation from the first row requires that the *head* map in a given CS has the same label as the head class.

6.2.4 Compilation: From Clafer Model to Multi-Clafer Shape

A Clafer compiler recursively traverses a Clafer model and builds a corresponding MCS. It takes a Clafer declaration D, decodes into an Clafer Shape $CS(D)$, and labels the elements of the latter using the name provided by D and adding indices to ensure uniqueness of identifiers, when the clafer names repeat. 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 (Tab. 6), so that shapes $CS(D)$ and $CS(D_i)$ are properly cojoined into an MCS. And so on until the entire Clafer model is traversed. Algebraically, compilation appears as an operation *Compile* in Fig. 13, as we discussed earlier.

6.2.5 Gluing: From Multi-Clafer Shape to 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 MCS 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 the mapping between an MCS and a CD. It considers labeling in an MCS as a formal graph morphism, from the graph of class and map roles (MCS) to the graph of real classes and maps (CD) formed by labels.

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

6.3 Formalizing Instantiation

Figure 20a presents a definition of Clafer instances and Fig. 20b shows their main feature — derivability from

the respective CD instances. Whereas the two model-level artifacts are formal CDs (DP-graphs), the two instance-level artifacts 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, the mappings *label* respect constraints (are DP-graphs morphisms). Chevron ***Claferize*** denotes an operation that completes an instance of a CD, i.e., a pair (OD, *type*) to a Clafer instance (MCI, *type**) (which is traced to the original instance via the mapping *label**). Below we will often refer to instances by their source graphs, and say "instance OD" and "instance MCI".

The operation ***Claferize*** recursively traverses an MCS and OD and builds a corresponding MCI. It takes each source class R from MCS, finds its instances O in OD, such that $label(R) = type(O)$, and, for each O, creates a source object role in MCI. It then finds and adds corresponding instances of the mappings *head*, *source*, etc. If the mapping *target* is instantiated for a given source object O, then it also adds the target object. After processing the class role R, the operation processes its children (pointed to by the mappping *head*) one by one, for each instance O.

Let us illustrate how the operation ***Claferize*** works based on the example from Fig. 17d. Note that in the class diagram CD, the class plaECU plays two roles: the source of the map master and the target of the map *dref*. The ***Claferize*** constructs the MCI in such a way that the diagram commutes, that is, $type^*.label = label^*.type$.

As we have seen in Sect. 6.1, an instance of a CD can be seen as a mega-mapping $[\![..]\!] : CD \to \textsc{SetMap}$. In the example, we have the instance $[\![\mathsf{Sing}]\!]_{\mathsf{OD}}=\{\_^*\}$, $[\![\mathsf{plaECU}]\!]_{\mathsf{OD}}=\{\mathsf{e1}, \mathsf{e2}\}$, $[\![master^*]\!]_{\mathsf{OD}}=\{m21, m11\}$, etc. Sequential mapping composition $label.[\![..]\!]_{\mathsf{OD}}$ then yields an instance of MCS. That is, for any element x of graph MCS, we define $[\![x]\!]=[\![x.label]\!]_{\mathsf{OD}}$, which gives us a mega-mapping $[\![..]\!] : MCI \to \textsc{SetMap}$. The latter can be represented by a typed graph (due to the Grothendieck construction discussed at the end of Sect. 6.1.1), which we denote by MCI. In other words, we set $[\![x]\!]_{\mathsf{MCI}} = [\![x.label]\!]_{\mathsf{OD}}=x.label.[\![..]\!]_{\mathsf{OD}}$ for all elements x of graph MCS.

Note that as the mapping *label* maps two different roles into one class ($[\![\mathsf{plaECU}_1]\!]_{\mathsf{MCI}} = [\![\mathsf{plaECU}_2]\!]_{\mathsf{MCI}} = \{\mathsf{e1}, \mathsf{e2}\}$), each object e_i, $i = 1, 2$ may play two roles of being the source and the target of *master* links. Hence, in the carrier graph MCI we have three clones of e1 (e1:plaECU$_1$ as a source, and e1:plaECU$_2$ as a target played twice: once as a target of e1 and once as a target of e2) and one clone of e2 (only e2:plaECU$_1$ as a source, because e2 does not play the role of a target). Other OD elements are not cloned because the mapping *label* only glues together classes plaECU$_1$ and plaECU$_2$. Mapping $label_0^*{:}MCI \to OD$ provides traceability of roles to objects, particularly, it glues together clones into their original objects.

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

7 Analytical Evaluation

We examine the extent to which Clafer meets its design goals from Sect. 3.

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 [15]. Figure 24 shows the TVL encoding of the feature model from Fig. 7. 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:
   - *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.

```
1  Options group allof {
2    Size group oneof { Small, Large },
3    opt Cache group allof {
4      CacheSize group allof {
5        SizeVal { int val; },
6        opt Fixed
7      }
8    },
9    Constraint { (Small && Cache) → Fixed; }
10 }
```

Fig. 24: Options feature model in TVL

- *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. 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 enable writing constraints that look like propositional logic, although their underlying semantics is FOL. For example, clafer names in the last line in Fig. 7 would be preceded by the quantifier some (cf. lines 11-13 in Fig. 26 in Appendix C). Name resolution rules further contribute to the conciseness of constraints.
- *Navigation over optional clafers*: Navigation expressions of the form $n_1.n_2 \dots n_m$ encompass navigation along the clafer hierarchy and occur in constraints (e.g., in the last line of Fig. 8). 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 25 shows the meta-model of Fig. 8 encoded in KM3 [43], 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.

```
1  class Comp {
2    reference version : integer
3  }
4
5  class ECU extends Comp {}
6
7  class Display extends Comp {
8    reference server : ECU
9    attribute options : Options
10 }
```

Fig. 25: Component meta-model in KM3

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*, *parent*, *dref*, and *head**), and three inclusions ($head_s$, $head_h$, and $head_t$), with all the constraints regulating well-formedness of CS 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 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 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 [39].
   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.

8 Related Work

Clafer builds on our several previous works, including encoding feature models as UML class models with OCL [23]; a Clafer-like graphical profile for Ecore, having a bidirectional translation between an annotated Ecore model and its rendering in the graphical syntax [63]; and the Clafer-like notation used to specify Framework-Specific Modeling Languages (FSMLs) [4]. Moreover, feature models have been characterized as views on class diagrams (referred to as ontologies) in [22]. None of these works provided a proper language definition that unifies feature and class modeling, and did not provide implementation like Clafer; also, they lacked Clafer's concise constraint notation. Although we introduced Clafer earlier [13], our previous work lacked precise semantics because semantic unification posed a major challenge. In our current work, we precisely specify the unification on feature and class modeling constructs and present Clafer's semantics.

Cardinality-based feature models have been formalized in [21] as context-free grammars, and in [50] 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.

TVL is a textual feature modeling language [15]. It favors the use of explicit keywords, which some software developers may prefer. The language covers 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.

Common Variability Language (CVL) is an Object Management Group (OMG) proposal for a standard for specifying and resolving variability [39]. 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. Clafer models, on the other hand, are self-contained. They can specify variability over existing models, but both would be encoded in Clafer (as in our running example). Similarly to UML, CVL 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 Clafer, the three concepts are unified into *clafer*. CVL provides a constraint language that is stratified into: 1) basic constraint language that specifies propositional constraints; and 2) full constraint language that provides full expressivity of OCL. Clafer has one constraint language that is inspired by Alloy constraints. It has the same expressivity as FOL. Furthermore, it treats Boolean features as singleton clafers, therefore navigation over clafers results in an empty set, in the worst case. In CVL, navigation may result in an *unknown value* if an element does not exist. CVL is a diagrammatic language, while Clafer is mostly textual (although graphical renderings exist). Finally, CVL aims at modeling and resolving variability, whereas Clafer excells in variabilty modeling and analysis. It can be used as a back-end for analyses of CVL models.

Kconfig and CDL [12] are languages for modeling variability of operating systems: the Linux kernel and eCos. They have a variety of constructs and mix variability with mechanisms to control how models are presented to users in a configurator. Clafer 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 [6]. 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 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 [45], but integrates it into class modeling. Finally, Forfamel considers a constraint language as out of scope, hinting at OCL. Clafer comes with a concise constraint notation.

Nivel is a meta-modeling language, which was applied to define feature and class modeling languages [5]. 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.

In essence, Alloy [41] is a class modeling language that reduces OOM to two primitives: signatures (typed sets) and relations. The former correspond to classes; the latter to associations. Alloy can encode and analyze more conceptually complex UML class diagrams [1]. In the context of variability modeling, Alloy has the same shortcomings as UML class diagrams (cf. Sect. 3). Clafer, on the other hand, was designed to support variability modeling and hierarchical models.

As mentioned earlier, class-based meta-modeling languages, such as KM3 [43] and MOF [54] cannot express feature models as concisely as Clafer. Furthermore, Clafer covers the same scope as MOF, but is based on fewer concepts, as a clafer represents classes, attributes, and relationships.

Decision models group decisions and focus on product derivation from a product family [20]. Notations for specifying decision models include a tabular notation [61] and Synthesis [62]. Clafer models are more expressive, and therefore subsume decision models (modulo language features controlling UI aspects such as visibility). In fact, owing to unification, Clafer can encode variability, class, and meta-models.

A rigorous approach to unification of different types of associations based on mathematical operations with mappings, particularly, tabulation, was proposed in [29]. 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., [65,25]. 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.

Refactoring of UML class diagrams is a practical activity performed during model evolution [64,8]. 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 [28,30]. 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 [31] (where DP-graphs are called *sketches*). Particularly, considering UML class diagrams as visualizations of formal class diagrams is elaborated in [27,29]. Several applications of DP-graphs to model-driven software engineering are developed in [59,60]

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 [18,36]. 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 meta-level, simply as parts of a single Clafer model. This design allows us to elegantly reuse a single constraint language at all these levels and to relate them. As another example, Janota and Botterweck show how to relate feature and architectural models using constraints [42]. 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 [7], 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 uniform language covering both feature and class modeling, and serving as a platform to derive such domain specific languages, as needed.

9 Conclusion

Clafer closes the gap between feature and class models and provides improvements on both sides. We showed how the essential modeling concepts can be unified both

syntactically and semantically, which allows for arbitrary property nesting and uniform representation of feature and class models. Clafer subsumes cardinality-based feature modeling with references. We are not aware of other works that precisely define semantics of such models. We provided semantics in a structurally explicit way, as MCSs resemble models in concrete syntax. Furthermore, the work on formal semantics gave us new insights and helped to fix the language (e.g., clarified the meaning of reference clafers and navigation through the clafer hierarchy).

Our work contributes to the design of modeling notations and model analyses. Clafer can encode rich structural models, e.g., domain models, variability models, class models. Unified syntax and semantics allow using a common infrastructure for reasoning on a large variety of models. We have implemented reasoning support based on Alloy, Z3 SMT, and Choco 3 CSP solvers.

Our work intends to help modelers express and analyze complex Software Product Lines. We have previously shown that a wide range of realistic feature models, meta-models, and model templates can be expressed in Clafer and that useful analyses can be run on them within seconds [13]. The recent thesis demonstrates using Clafer for modeling the three levels of EAST-ADL architecture on an example of complex automotive electronic/electric architecture and reasoning over such an integrated model [51].

This work has also 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) and protocol buffers.

In the future, we would like to carry out empirical studies to explore the benefits and disadvantages of concept unification in practical modeling. Other promising directions for future work include adding support for: 1) domain-specific abstractions in Clafer, 2) modularity to provide visibility and variability interfaces, and 3) behavioral modeling to express the evolution of state specified by Clafer models over time. Finally, we would like to adapt Clafer and the tools to better support architecture modeling, design exploration, and multi-objective optimization, e.g., to find optiminal function-to-hardware deployment and topology generation.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Model Driven Engineering Languages and Systems (2007)
2. Antkiewicz, M., Bąk, K., Zayan, D., Czarnecki, K., Wąsowski, A., Diskin, Z.: Example-driven modeling using clafer. In: First International Workshop on Model-driven Engineering By Example (2013). URL http://ceur-ws.org/Vol-1104
3. Antkiewicz, M., Bąk, K., Murashkin, A., Liang, J., Olaechea, R., Czarnecki, K.: Clafer Tools for Product Line Engineering. In: Proceedings of the 17th International Software Product Line Conference co-located workshops (2013)
4. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. Software Engineering, IEEE Transactions **35**(6) (2009)
5. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and Systems Modeling **8**(4) (2009)
6. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: Software Product Line Conference, 10th International (2006)
7. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Advanced Engineering Informatics **21**(1) (2007)
8. Astels, D.: Refactoring with UML. In: Proceedings of the 3rd International Conference eXtreme Programming and Flexible Processes in Software Engineering (2002)
9. Bąk, K., Zayan, D., Czarnecki, K., Antkiewicz, M., Diskin, Z., Wąsowski, A., Rayside, D.: Example-driven modeling. model = abstractions + examples. In: New Ideas and Emerging Results (NIER) track of the 35th International Conference on Software Engineering (2013)
10. Barr, M., Wells, C.: Category theory for computing science, vol. 10. Prentice Hall New York (1990)
11. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (2013)
12. Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K.: Variability modeling in the real: a perspective from the operating systems domain. In: Proceedings of the IEEE/ACM international conference on Automated software engineering (2010)
13. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and meta-models in Clafer: mixed, specialized, and coupled. In: Software Language Engineering (2010)
14. Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A.: Partial Instances via Subclassing. In: Software Language Engineering (2013)
15. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming **76**(12) (2011)
16. Clauß, M., Jena, I.: Modeling variability with UML. In: Young Researchers Workshop at GCSE (2001)
17. Consortium, A., et al.: EAST-ADL domain model specification, nov 28, 2013, version 2.1.12
18. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Generative Programming and Component Engineering (2005)
19. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Generative Programming and Component Engineering (2002)
20. Czarnecki, K., Grüenbacher, P., Rabiser, R., Schmid, K., Wąsowski, A.: Cool features and tough decisions: A comparison of variability modeling approaches. In: Proceedings of the sixth international workshop on variability modeling of software-intensive systems (2012)
21. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software process: Improvement and practice **10**(1) (2005)

22. Czarnecki, K., Hwan, C., Kim, P., Kalleberg, K.: Feature models are views on ontologies. In: Software Product Line Conference, 10th International (2006)
23. Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report. In: International Workshop on Software Factories (2005)
24. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Proceedings of the 5th international conference on Generative programming and component engineering (2006)
25. Dahchour, M., Pirotte, A., Zimányi, E.: Generic relationships in information modeling. Journal on Data Semantics IV **3730** (2005)
26. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
27. Diskin, Z.: Visualization vs. specification in diagrammatic notations: A case study with the UML. In: Diagrams (2002)
28. Diskin, Z., Cadish, B.: Variable sets and functions framework for conceptual modeling: Integrating ER and OO via sketches with dynamic markers. In: Object-Oriented and Entity-Relationship Modeling (1995)
29. Diskin, Z., Easterbrook, S., Dingel, J.: Engineering Associations: From Models to Code and Back through Semantics. In: Objects, Components, Models and Patterns (2008)
30. Diskin, Z., Kadish, B.: Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. Data and Knowledge Engineering **47** (2003)
31. Diskin, Z., Kadish, B., Piessens, F., Johnson, M.: Universal arrow foundations for visual modeling. In: Diagrams (2000)
32. Felfernig, A., Friedrich, G.E., Jannach, D.: UML as domain specific language for the construction of knowledge-based configuration systems. International Journal of Software Engineering and Knowledge Engineering **10**(04) (2000)
33. Gaeta, J.A.P.: Modeling and implementing variability in aerospace systems product lines. Master's thesis, University of Waterloo (2014)
34. Gomaa, H.: Designing software product lines with UML. Addison-Wesley Boston, USA; (2004)
35. Group, O.M.: Systems Modeling Language (SysML) (2012). http://www.omg.org/spec/SysML/1.3/
36. Heidenreich, F., Kopcsek, J., , Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion of the 30th international conference on Software engineering (2008)
37. Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., Heymans, P.: Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies. In: Software Language Engineering (2010)
38. Hubaux, A., Xiong, Y., Czarnecki, K.: A user survey of configuration challenges in Linux and eCos. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (2012)
39. IBM, Thales, FOKUS, F., TCS: Proposal for Common Variability Language (CVL) Revised Submission (2012)
40. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology **11**(2) (2002)
41. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2011)
42. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: Fundamental Approaches to Software Engineering (2008)
43. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems (2006)
44. Jussien, N., Rochart, G., Lorca, X., et al.: Choco: an open source java constraint programming library. In: CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08) (2008)
45. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, CMU (1990)
46. Kang, K.C.: FODA: Twenty years of perspective on feature modeling. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (2010)
47. Lambek, J., Scott, P.J.: Introduction to higher-order categorical logic, vol. 7. Cambridge University Press (1988)
48. Liang, J.: Correcting Clafer Models with Automated Analysis. Tech. Rep. GSDLab-TR 2012-04-30, GSD Lab, University of Waterloo (2012)
49. Liang, J.: Solving Clafer models with Choco (GSDLab-TR 2012-12-30) (2012)
50. Michel, R., Classen, A., Hubaux, A., Boucher, Q.: A formal semantics for feature cardinalities in feature diagrams. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (2011)
51. Murashkin, A.: Automotive electronic/electric architecture modeling, design exploration and optimization using Clafer. Master's thesis, University of Waterloo (2014)
52. Murashkin, A., Antkiewicz, M., Rayside, D., Czarnecki, K.: Visualization and Exploration of Optimal Variants in Product Line Engineering. In: Proceedings of the 17th International Software Product Line Conference (2013)
53. Olaechea, R., Stewart, S., Czarnecki, K., Rayside, D.: Modeling and multi-objective optimization of quality attributes in variability-rich software. In: Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages (2012)
54. OMG: Meta Object Facility (MOF) Core Specification (2011)
55. OMG: OMG Object Constraint Language (OCL)2.4 (2014)
56. Partnership, A.D.: Automotive open system architecture (autosar), release 4.1 (2013). http://www.autosar.org/specifications/release-41/
57. Partnership, A.D.: Feature model exchange format (2013). https://www.autosar.org/fileadmin/files/releases/4-1/methodology-templates/templates/standard/AUTOSAR_TPS_FeatureModelExchangeFormat.pdf
58. Reiser, M.O., Kolagari, R.T., Weber, M.: Unified feature modeling as a basis for managing complex system families. In: Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems (2007)
59. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in MDE. The Journal of Logic and Algebraic Programming **79**(7) (2010)
60. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formal approach to the specification and transformation of constraints in MDE. The Journal of Logic and Algebraic Programming **81**(4) (2012)
61. Schmid, K., John, I.: A customizable approach to full lifecycle variability management. Science of Computer Programming **53** (2004)
62. Software Productivity Consortium Services Corporation: Reuse-driven software processes guidebook, version 02.00.03. Tech. Rep. SPC-92019-CMC (1993)
63. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Tech. Rep. 2008-08, University of Waterloo (2008)
64. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.M.: Refactoring UML models. In: UML – The Unified Modeling Language. Modeling Languages, Concepts, and Tools (2001)
65. Wand, Y., Storey, V., Weber, R.: An ontological analysis of the relationship construct in conceptual modeling. ACM Transactions on Database Systems **24**(4) (1999)

A Notation and Terminology

A.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. B.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 $\mathit{id}_A : A \to A$.

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

A.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 and blank.

B 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. B.1–B.3, and then proceed with syntactical means for specifying fragments of SetMap (Sect. B.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.

B.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). Then we show that both essentially define the same construct called a *mapping*.

B.1.1 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*, $r : A \twoheadrightarrow B$, 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:

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

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 $r : A \twoheadrightarrow B$ gives rise to a total single-valued function $\hat{r} : L_r \to A \times B$ (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 $p : A \leftarrow A \times B$ and $q : A \times B \to B$ 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 $L_r! \stackrel{\text{def}}{=} \hat{r}(L_r) \subset A \times B$, 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 $!_{A,B} : \mathsf{Span}(A, B) \to \mathsf{Rel}(A, B)$; 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 $r1, r2 : A \twoheadrightarrow B$ are considered *isomorphic*, $r1 \cong r2$, iff there is a bijection between their heads commuting with legs.

B.1.2 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 $x : I \to X$, 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, $\iota_a.f2^a = f1^a$.

B.1.3 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,

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

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 $r : A \nrightarrow B$ a span and $f : A \twoheadrightarrow B$ an mv-function with $r^* \cong f$ (or, equivalently, $f^* \cong r$).

B.1.4 Set-valued mappings.

Given an mv-function $f : A \twoheadrightarrow B$ and an element $a \in A$, suppose that the indexing function $f^a : I^a_f \to B$ (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 $f \cong f! : A \to \mathsf{Set}(B)$. 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 $r : A \nrightarrow B$ is a relation iff its navigational counterpart, mv-function $r^* : A \twoheadrightarrow B$, is set-valued.*

B.2 Semantic universe: Operations on mappings

B.2.1 (Sequential) Mapping Composition.

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

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* $q.r : A \nrightarrow C$ is defined as follows. The head

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

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 $\beta^* = s_\beta^{-1}.t_\beta = f.g$.

B.2.2 Inversion.

Given a set-valued function $f : A \twoheadrightarrow B$, its *inverse* is a set-valued function $g : A \twoheadleftarrow B$ 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 $r^{-1^{-1}} = r$.

B.3 Semantic universe: Configurations of mappings and their properties

Table 3 presents several important properties of mapping configurations, which we call *diagram predicates*. The left column gives their names, the middle one specifies their *arities*, i.e., configurations of mappings that may have the property, and the right column provides semantics.

B.4 Syntax: DP-graphs

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 associations)*, 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. Here we specify syntax and semantics of DP-graphs/formal CDs formally.

B.4.1 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 $src, trg: G_A \to G_N$ giving each arrow its source and target.

A *graph morphism (mapping)* $f: G_1 \to G_2$ is a pair of functions $f_N: G_{1N} \to G_{2N}$ and $f_A: G_{1A} \to G_{2A}$ 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 $trg(f_A(a)) = f_N(trg(a))$.

B.4.2 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 $G_{\mathrm{ar}}(P)$.

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 $args: G_{\mathrm{ar}}(P) \to G$ a graph morphism binding formal parameters in the arity graph by the actual arguments — elements of graph G. Assume the arity graph $G_{\mathrm{ar}}(P)$ 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 $(P, \mathbf{a})$ with P a predicate symbol and $\mathbf{a}$ a bag of arrows of the carrier graph, whose indexing set is the arity graph $G_{\mathrm{ar}}(P)$ (note that the indexing function must be a correct graph morphism – this constraint was called [ad], arity discipline, in Sect. 6.1.3.

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.

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 $\phi \in \Phi_1$.

```
1  abstract 0..* options 0..* {
2    abstract 1..1 size 1..1 {
3      abstract 0..* small 0..1 {}
4      abstract 0..* large 0..1 {}
5    }
6    abstract 0..* cache 0..1 {
7      abstract 0..* size → integer 1..1 {
8        abstract 0..* fixed 0..1 {}
9      }
10   }
11   [ some this.size.small &&
12     some this.cache ⟹
13     some this.cache.size.fixed ]
14 }
```

Fig. 26: Desugared Clafer model.

B.5 Syntax and semantics together

A major idea of categorical logic [10] 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 Γ[SetMap]), and f, g are mappings between them (i.e., arrows in Γ[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 Φ[SetMap]. Thus, the big set of formulas Φ[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 $[\![..]\!]: S \to$ SetMap. An immediate consequence of such an arrangement is the following 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, $[\![..]\!]: S_2 \to$ SetMap, its composition with f gives us a correct instance of S_1.

C Clafer Concrete Syntax

Conciseness is an important goal for Clafer; therefore, it provides syntactic sugar for common constructions. Figure 26 shows the model from Fig. 7 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. 27). 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 '→' symbol, and *reference bag* (name followed by '↠' symbol); not shown.

⟨Clafer⟩ ⇒ ⟨Abs⟩ ⟨GCard⟩ string ⟨Super⟩ ⟨Target⟩ ⟨Card⟩
⟨Elements⟩
⟨Abs⟩ ⇒ | **abstract**
⟨Elements⟩ ⇒ **{**⟨ElList⟩**}** ⟨ElList⟩ ⇒ | ⟨Element⟩ ⟨ElList⟩
⟨Element⟩ ⇒ ⟨Clafer⟩ | ⟨Constraint⟩
⟨Super⟩ ⇒ | **:** string
⟨Target⟩ ⇒ | ⟨Kind⟩ string
⟨Kind⟩ ⇒ → | ↠
⟨GCard⟩ ⇒ | **xor** | **or** | **mux** | **opt** | ⟨NCard⟩
⟨Card⟩ ⇒ | **?** | **+** | ***** | ⟨NCard⟩
⟨NCard⟩ ⇒ integer **..** ExInteger
⟨ExInteger⟩ ⇒ ***** | integer

Fig. 27: BNF grammar of Clafer (no constraints).

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

Fig. 28: BNF grammar of core Clafer constraints

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.

D Clafer Constraint Language

The Clafer constraint language is essentially borrowed from Alloy [41]. 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. 7) stands for some (Fig. 26).

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 a CS the head class represents the clafer. If in a Clafer model constraint is defined at top-level, then in the corresponding MCS 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.

D.1 Grammar

Figure 28 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. 28 *var* represents variables bound by quantifiers. In the production with binary operators, ⊕ is one of $<, =, +, -$ (logical comparison, equality, addition, and subtraction, respectively). In the production with set operators, ⊗ 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 $n_1.n_2 \dots n_m$, we call such as an expression a *navigation*. The dot between names indicates *relational join*.

D.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 MCS; thus MCS and CD have all names properly resolved. The rules are similar to CVL rules [39], 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*, *dref*, 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 $n_1.n_2 \dots n_m$) the name resolution rules are applied to resolve n_1 first. Once it is resolved, subsequent clafers ($n_2.n_3 \dots n_m$) 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 this.$c_2 \dots c_m$.

D.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.

D.3.1 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.

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

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

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

D.3.2 Set Expressions

Set operators.

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

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

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

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.

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

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

D.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.

D.4.1 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.

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

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.

$[\![\mathsf{SetExp}]\!]_E\ env = [\![\mathsf{SetExp}]\!]_S\ env$

D.4.2 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.

$[\![\mathsf{Name}]\!]_S\ env = env(\mathsf{Name})$

E MCS Constraints

Each Multi-Clafer Shape is only valid if it satisfies incidence constraints (defined in Tab. 5), clafer kind/shape discipline constraints (defined in Tab. 7), clafer cojoining constraints (defined in Tab. 6), and naming discipline constraints (defined in Tab. 8).

Description	Constraints
The map *head_map* goes from class (role) *source_class* to class *head_class*. Analogical constraints hold for the maps *parent_map*, *dref_map*, and *target_map*.	this.*head_map.so* = this.*source_class* this.*head_map.ta* = this.*head_class*

Table 5: Incidence constraints in the context of Clafer, which the MCS meta-model in Fig. 23 must satisfy

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

Table 6: Clafer cojoining MCS constraints in the context of Clafer, which the MCS meta-model in Fig. 23 must satisfy

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

Table 7: Clafer kind constraints in the context of Clafer, which the MCS meta-model in Fig. 23 must satisfy

Description	Constraints
The map *head* has the same name as the class *head*.	this.*head_map.label* = this.*head_class.label*
The map *parent* is named "parent". Analogical constraints holds for the map *dref*.	this.*parent_map.label* = "parent"
The map *target* (if defined) has name derived from the name of the class *head* by concatenating the name with *.	this.*target_map.label* = *concat*(this.*head_map.label*, *)
There is one distinguished element of String named "Sing". Analogical constraints hold for other predefined clafers, such as "int" and "string".	this $\in$ Sing $\iff$ this.*head_class.label* = "Sing"

Table 8: Naming constraints in the context of Clafer, which the operation *Compile* must satisfy

F Full Telematics Model

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

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

abstract comp
  version → integer

abstract ECU : comp

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

abstract plaECU : ECU
  plaDisplay : display 1..2
    [ no 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 ]

[ dual
 extraDisplay
 telematics.size.large ]
```