Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Core Concepts of the Compositional Variability Management Framework (CVM) — A Practitioner's Guide —

Mark-Oliver Reiser
Fakultät IV - Softwaretechnik - TEL 12-3
Technische Universität Berlin
10587 Berlin, Germany
moreiser@cs.tu-berlin.de

Bericht-Nr. 2009-16
ISSN 1436-9915

Contents

1 Introduction and Scope **5**

2 Feature Modeling **7**

3 Feature Configurations **17**

4 Configuration Links **23**

5 Variable Entities **35**

6 Advanced Considerations **45**

A Summary of Configuration Decision Definition **53**

B Glossary **55**

1 Introduction and Scope

This report provides a detailed overview of the core concepts of the Compositional Variability Management framework (CVM), in particular feature modeling, configuration links and variable entities. It favors readability over strict formal accuracy and therefore does not give a comprehensive mathematical formalization of syntax and semantics. Furthermore, it favors a conceptual perspective over elaborate discussions of tool issues. This is why the textual Variability Specification Language (VSL), which was introduced as part of CVM, was chosen here as a basis for the presentation. The textual character of VSL allows us to explore the concepts and examples on a mainly conceptual level, without having to treat implementation and the GUI aspects of CVM in detail. More tool-related instructions can be found in the online help accompanying the CVM Eclipse plugin bundle.

In its feature modeling part, which forms the foundation of the other two concepts, CVM mainly incorporates and integrates the ideas and concepts of countless other researchers and practitioners from the field of product line engineering, in particular basic feature modeling from Kang et al. [4] and cardinality-based feature modeling from Czarnecki et al. [2]. It is not possible to provide references to the originators in each and every case below, but an overview and comparison of common feature modeling techniques with many further references can be found in [8].

The major goal was to provide a quick CVM walk-through of considerable detail but without overly elaborate explanations. At some points, however, a few in-depth considerations were included. These are typeset in a special font (like this paragraph) and the hurried reader may safely disregard them.

The remainder of this text is organized as follows. Basic feature models and their specification as supported by CVM are presented in Chapter 2. Since a detailed understanding of the configuration of such feature models is required for the more advanced concepts of CVM, we will take a closer look at feature configurations in Chapter 3. Based on this, Chapters 4 and 5 then present the novel concepts of configuration links and variable entities. Finally, Chapter 6 concludes with a quick look at some selected further capabilities of CVM.

Both CVM and VSL are likely to evolve after publication of this report. Information on additions and changes to the concepts presented here will be made available on the project web site [1].

Heidelberg
July 2009

2 Feature Modeling

First, we take a look at the core feature-modeling capabilities in CVM, which form the basis of the more advanced variability management facilities such as configuration links or variable entities.

It is important to note that on the level of basic feature modeling CVM does not introduce any new concepts or ideas. Instead, an important design goal was to make it as compatible as possible with the classical feature modeling approaches proposed in the literature. However, the fact that CVM tries to integrate all these approaches into a single, consistent feature-modeling methodology might to some extent be seen as a novel contribution.

As an important exception to this rule of maximum compatibility, the approaches which were used as a basis had to be liberalized sometimes. This was necessary to achieve an integration of different, sometimes contradictory, approaches. Secondly, this was required to foster the highly abstract understanding of the term "feature", as described in the coming section, and to enable the broad range of application scenarios that follow from that. A good example of such liberalization is the restriction of having only a single root feature that some feature-modeling approaches impose: in CVM, you can have as many root features in a feature model as you like, which facilitates applying features as mere configuration parameters and making a feature model a flat list of such configuration parameters.

2.1 Hold on! What is a Feature, Anyway?

Well, a good question indeed! This is already one of the most challenging and most debated questions in feature modeling. Quite a few different definitions were proposed in the literature, which we do not investigate in more detail here. In principle, CVM does not rely on a certain understanding of the term "feature". However, the advanced concepts in CVM—configuration links and compositional variability in particular—will prove most useful when assuming a very broad understanding, as in this definition:

> *A feature is a characteristic or trait in the broadest sense that an individual variant of a variable object of consideration may or may not possess* [7].

In other words, when using feature modeling to specify the variability of some variable object, a feature simply denotes some characteristic of this object that may or may not apply to each of its variants. For example, I might want to model the variability of newspapers; then, all conceivable traits of newspapers will be candidates for being modeled as a feature: such as "has colored headlines", "has pictures on front page", "text in Spanish", etc.

Such a highly abstract understanding of the term then allows feature modeling to be used across the entire development process for very different things: you can model the variability of your entire product line, i.e. the object of consideration is your product line on a global level, or you can model the variability within a particular subsystem or even a single, fine-grained software component far down in your software design. Several examples of this will be presented below in Chapter 5.

2.2 Features and Feature Models

Let's begin simply. In CVM you can, of course, have *features*. Then, *feature models* serve as an aggregation of such features. Listing 1 shows what a simple feature model might look like.

Listing 1: A simple feature model with some features.

```
/*
 * About as simple as it can get ...
 */
featureModel SimpleFM {

    ClimateControl;         // a root feature
    FrontWiper, RearWiper;  // two more root features
    "Rain Sensor";

}
```

The feature model is named "SimpleFM" and contains four root features. Note that it makes no difference whether you specify the features one at a time, each followed by a semicolon, as for "ClimateControl" and "Rain Sensor", or as a comma-separated list with a single semicolon at the end, "FrontWiper" and "RearWiper" in the example. This is very similar to programming languages such as C or Java where

```
int a, b, c;
```

is equivalent to

```
int a;
int b;
int c;
```

In many cases, VSL follows such syntactical traditions of C and Java. Another example are line breaks, which do not have any syntactical or semantical meaning (except to terminate a single-line comment).

The above listing illustrates two other important things. Comments can be introduced using the well-known C-language style, i.e. `//` for single-line and `/*` ... `*/` for multi-line comments. All names in the language, for example that of feature models or features, may contain white-space or other special characters but must then be enclosed in double quotes, as shown above for "Rain Sensor".

2.3 Parent/Child Relations

Feature models are usually organized in tree form. More precisely, feature models in CVM are forests of directed out-trees. Listing 2 shows how to define such a parent/child structure in VSL. To add one or more children to a feature, simply follow the feature

Listing 2: Parent features and their children.

```
featureModel FM {

    ClimateControl ( Simple, Advanced );
    FrontWiper ( RainSensor );
    RearWiper ( RainSensor );
    Root (
        Child1,
        Child2 ( GrandChild1, GrandChild2 ),
        Child3
    );
}
```

declaration with a pair of brackets enclosing a comma-separated list of child features: `Parent ( Child1, Child2 )`. These child declarations can be nested to form complex tree structures, as shown for “Root”, “Child2” and the grandchildren in Listing 2.

Following traditional terminology from graph theory, a feature without a parent is called a *root feature* and one without any children is called a *leaf feature*.

In CVM, features each have a single parent at most. The use cases of having several parents per feature, as supported by some feature-modeling approaches, can be dealt with by using feature inheritance, cf. Section 6.4.

2.4 Cardinalities

Following the conception of cardinality-based feature modeling [2], each feature has a cardinality that can be used to declare features as *mandatory* or *optional*:

Listing 3: Defining feature cardinalities.

```
featureModel FM {
    FrontWiper[1]    ( RainSensor[0..1] );
    RearWiper[0..1] ( RainSensor[0..1] );
    IntervalMode[2..8];
}

featureModel FM_Concise {
    FrontWiper! ( RainSensor? );
    RearWiper?  ( RainSensor  );    // default cardinality is [0..1]
    IntervalMode[2..8];             // no shortcut available here
}
```

Features with a maximum cardinality greater than 1 are also allowed (cf. “IntervalMode” in the above example). Such features are called *cloned features*. As can be seen in Listing 3, a short-hand form is available for the two most common cardinalities: `!` denotes a cardinality of [1] and `?` stands for cardinality [0..1]. Also, specifying `?` or `[0..1]` is not really necessary because [0..1] is the default cardinality; but this additional hint may make specifications more readable. Thus, given these abbreviations, the two feature models "FM" and "FM_Concise" are equivalent.

The semantics of these cardinalities can be summarized as follows: a feature's cardinality states how often this feature must be selected in a valid configuration if its parent is selected[1]. This general rule leads to the following obersvations:

- [0..1] means that the feature may or may not be selected, i.e. it is *optional*,
- [1] means that it must be selected exactly once, i.e. it is *mandatory*,
- [0] means that it may never be selected and thus never appears in any valid configuration (this somewhat degenerate cardinality will be of use below when dealing with feature inheritance),
- [0..∗] or any other cardinality with a maximum greater 1 means that it may be selected several times, i.e. it is a *cloned feature*.

The latter case is special in that each "selection" of the feature actually represents an instantiation, so the cloned feature's child features will be configured separately for each such instantiation or "selection".

2.5 Parameterization

When looking at the "IntervalMode" feature in Listing 3, we encounter a common limitation of basic feature modeling. By selecting 2 to 8 "IntervalMode" features in a configuration, we can only state how many interval modes our wiper will offer, but not how long the duration of the interval will be in each case. This is precisely the purpose of a *feature parameter*, often called feature attribute in the literature. A feature having such a parameter is referred to as a *parameterized feature*. Listing 4 provides examples.

Listing 4: Parameterized features.

```
featureModel ParamFM {
    Axles! : integer;                    // num of axles
    TopSpeed? : integer;                 // speed in km/h
    Wiper! (
        IntervalMode[2..8] : float       // duration in seconds
    );
}
```

The examples show that in VSL a feature can be parameterized by simply appending a colon followed by a type. Now the number of axles can be defined by way of the

[1] If a feature's parent is not selected, a feature may never be present in a configuration.

parameter of feature "Axles"; if the vehicle's speed should be limited to a fixed maximum speed, then the "TopSpeed" feature can be selected and the maximum speed is specified in kilometres per hour; and finally the interval modes can be provided with a duration in seconds for each interval. Note that all forms of cardinalities can be combined with feature parameterization and that in the case of cloned features each instance selected during configuration is provided with a separate value for its parameter, i.e. when selecting 4 interval modes, we can define a different duration for each of them.

More advanced type specifications are available, as summarized by the following pseudo-EBNF grammar:

$$\begin{array}{rcl}
\langle \mathit{type} \rangle & \rightarrow & \langle \mathit{kind} \rangle \\
 & & [\ \langle \mathit{range} \rangle \mid \langle \mathit{pattern} \rangle \mid \langle \mathit{enum} \rangle\ ] \\
 & & [\ \langle \mathit{default} \rangle\ ]\ . \\
\langle \mathit{kind} \rangle & \rightarrow & \text{`}\texttt{\textbf{boolean}}\text{'} \mid \text{`}\texttt{\textbf{integer}}\text{'} \mid \text{`}\texttt{\textbf{float}}\text{'} \mid \text{`}\texttt{\textbf{string}}\text{'}. \\
\langle \mathit{range} \rangle & \rightarrow & \text{`}\texttt{\textbf{[}}\text{'}\ \langle \mathit{value} \rangle\ \text{`}\texttt{\textbf{..}}\text{'}\ \langle \mathit{value} \rangle\ \text{`}\texttt{\textbf{]}}\text{'}. \\
\langle \mathit{pattern} \rangle & \rightarrow & \text{`}\texttt{\textbf{=}}\sim\text{'}\ \ \text{`}\texttt{\textbf{"}}\text{'}\ \langle \mathit{reg\text{-}expr} \rangle\ \text{`}\texttt{\textbf{"}}\text{'}. \\
\langle \mathit{enum} \rangle & \rightarrow & \text{`}\texttt{\textbf{\{}}\text{'}\ \langle \mathit{value} \rangle\ (\ \text{`}\texttt{\textbf{,}}\text{'}\ \langle \mathit{value} \rangle\ )*\ \text{`}\texttt{\textbf{\}}}\text{'}. \\
\langle \mathit{default} \rangle & \rightarrow & \text{`}\texttt{\textbf{=}}\text{'}\ \langle \mathit{value} \rangle.
\end{array}$$

In order to refine the primitive types available, a range with a minimum and maximum value can be provided, a pattern can be supplied in the form of a regular expression that must be matched by the value (only for the primitive type '**`string`**'), and a comma-separated list of legal values can be specified which leads to an enumerate type. Finally, a default value can be defined.

2.6 Groups

There is another important concept of feature modeling which is supported by nearly all feature-modeling techniques in some way or other: *feature groups*. They are used to define, for example, that two or more features are alternative with respect to each other, i.e. only one of them can be present in a configuration at any given time.

In CVM, a feature group aggregates sibling features, i.e. features that are children of the same parent feature, and it is provided with a cardinality in much the same way as features are. As a consequence, root features cannot be grouped because they do not have a parent. Features that are in a feature group are called *grouped features*. To distinguish between a feature's cardinality and that of a group, we speak of a *feature cardinality* or a *group cardinality*.

A group's cardinality states how many of the grouped features may be present in a valid configuration. Thus, a group cardinality of [1] states that *exactly one* of the features in the group must be chosen whenever the parent feature is chosen. This is often called *mandatory alternative*. Similarly, a group cardinality of [0..1] requires that *at most one* of the features in the group can be chosen, which is usually called *optional alternative*. Another common group cardinality is [1..$*$] meaning "*at least one*". All other cardinalities are also allowed for feature groups, but a cardinality of [0..∗] is usually not very useful because it does not constrain the selection of the grouped features in any way, which is usually the sole purpose of a feature group.

Creating feature groups in VSL is simple. Just use the same syntax as for defining a feature with several child features but omit the feature name, as in the following example.

Listing 5: Grouping features.

```
featureModel FM {
    Wiper (
        [1] (               // creates a feature group
            Simple,
            Advanced
        )
    );
}
```

This creates a feature “Wiper” with two children “Simple” and “Advanced” both grouped in a feature group of cardinality [1]. Thus, “Simple” and “Advanced” are two alternative forms of the wiper, one of which has to be chosen whenever a wiper is selected.

The feature groups as implemented in CVM are very similar to feature groups in cardinality-based feature modeling [2]. However, one important difference applies: grouped features in cardinality-based feature modeling are not allowed to have a cardinality. To get rid of this special case on both a syntactic and semantic level, features in CVM are treated exactly the same way, regardless of whether they are grouped or non-grouped. This means that you can provide a cardinality for grouped features other than [0..1], but you will only need this in some rare cases[2].

2.7 Feature Links

We have seen that the various feature-modeling concepts define constraints on how to select or deselect the features in a feature model: parent/child relations state that whenever you want the child, you also need the parent; feature cardinalities state how many instances of the child feature(s) you must have in case you choose the parent; and feature groups further constrain the selection of those children within the group.

Obviously, not all conceivable constraints can be expressed by these core modeling concepts. This is especially true if you wish to somehow constrain the configuration of two or more features which are located far away from each other in different parts of the feature tree. Consider the example in Listing 6. Assume that the advanced variant of the wiper occupies the same installation space in the car body that is required for the cruise control’s radar. In this case, we want to somehow state that feature “Advanced” cannot

[2]In addition to getting rid of special cases in syntax and semantics, the rationale for allowing cardinalities of grouped features is to be able to realize some specific legacy variability modeling techniques as specializations of the CVM approach [5].

Listing 6: Feature model with two incompatible features.

```
featureModel FM {
    Wiper (
        [1] (
            Simple,
            Advanced        // never use this with Radar!!
        ),
        FlexibleInterval
    ),
    CruiseControl (
        Adaptive (
            Radar : integer = 600   // min distance in cm
        )
    );
}
```

be selected together with feature “Radar”. Since these two features are not siblings, i.e. they do not share the same parent, we cannot use a feature group for that. Instead, *feature links* are a possible solution here. A feature link always has a *start feature*, an *end feature* and a *type*; in addition it has a *direction*: either *unidirectional* or *bidirectional*. The type is a simple keyword string. CVM provides these built-in types:

Table 1: Built-in feature link types and their semantics.

Type	Semantics
`needs`	The start feature needs the end feature, i.e. the start feature may only be selected if the end feature is selected, too.
`excludes`	The start and end features exclude each other, i.e. they may never both be selected (a.k.a. “optional alternative”).
`alternative`	The start and end features are alternative, i.e. exactly one of them must be selected (a.k.a. “mandatory alternative”).
`suggests`	The weak form of “needs”: you may select the start without the end feature, but you should have a good reason to do so.
`impedes`	The weak form of “excludes”: you may select both the start and end feature, but you should have a good reason to do so.

In addition to these, you can use your own types of feature links. In contrast to the built-in ones, you should name your custom types in a globally unique form to avoid confusion when sharing models with the outside world. By convention, a Java-style naming scheme should be used, as in `org.`<*mydomain*>`.cvm.linkTypes.`<*mytype*>.

Using this technique, we can now define the constraint between “Advanced” and “Radar” with an appropriate feature link, as shown here:

Listing 7: Feature links.

```
1  featureModel FM {
2      Wiper (
3          [1] (
4              Simple,
5              Advanced
6          ),
7          FlexibleInterval
8      ),
9      CruiseControl (
10         Adaptive (
11             Radar : integer = 600   // min distance in cm
12         )
13     );
14
15     link Advanced <= excludes => Radar;
16 }
```

Note that the exclude link defined here is bidirectional. If we wanted to state that the advanced wiper always requires the radar to be built in (for what ever reason), we would change line no. 15 above to

```
15     link Advanced = needs => Radar;
```

which would result in a unidirectional feature link from “Advanced” to “Radar” because we have an “=” on the left-hand side instead of a “<=”. For the built-in link types, the CVM editors will warn you if you use an inappropriate direction.

2.8 Feature Constraints

Apparently, not all conceivable constraints on a feature model’s configuration can be expressed by feature links. When more complex constraints must be specified, *feature constraints* come into play. Listing 8 on the next page illustrates how to employ such constraints. Lines 15 and 16 define two constraints which are each equivalent to the exclude link from Listing 7. Line 16 uses a more condensed syntax in which the keyword “`not`” may be replaced by `!`, “`and`” by `&` and “`or`” by `|`. Line 17 gives an example of a constraint that could not be represented by feature links: whenever having “Advanced” or “FlexibleInterval” or both, then “Radar” must not be built in and vice versa. In this manner constraints of arbitrary complexity can be formulated in propositional logic.

As a general guideline, feature links and constraints should only be used if it is not possible or feasible to realize the dependency with appropriate parent/child relations, mandatory children or feature groups. Furthermore, feature links should be favored over feature constraints whenever possible because they provide a more standardized scheme of formulating constraints, thus have a better readability and can be represented graphically in feature diagrams.

Listing 8: Feature constraints.

```
featureModel FM {
    Wiper (
        [1] (
            Simple,
            Advanced
        ),
        FlexibleInterval
    ),
    CruiseControl (
        Adaptive (
            Radar : integer = 600   // min distance in cm
        )
    );

    constraint not (Advanced and Radar);
    constraint ! (Advanced & Radar);
    constraint ! ((Advanced | FlexibleInterval) & Radar);
}
```

3 Feature Configurations

Feature models are used to define the variability of a particular variable object, i.e. what variations we have and how they are related. Each feature model thus defines a particular configuration space. Based on this definition, we can then uniquely identify individual variants of the variable object by stating for each feature whether or not it pertains to this variant, providing values for parameterized features, etc. This information is called a *feature configuration*. Figure 1 introduces a simple notation to illustrate that C_{FM} is a configuration of feature model FM. This notation will come in handy when explaining configuration links below.

$$\begin{tikzcd}
FM \arrow[d, dash] \\
C_{FM}
\end{tikzcd}$$

Figure 1: Configuration C_{FM} of feature model FM.

A feature configuration can be incomplete in the sense that for some features it has not (yet) been stated whether they are included or not. Such a configuration is called a *partial configuration* (in contrast to a *full configuration*), with the special case of an *empty configuration* in which nothing is configured at all. CVM offers full support for partial configurations and incremental configuration, i.e. the process of getting from an empty to a full configuration not in a single step but in several consecutive steps of incremental selections.

3.1 Basics

In VSL, you can create a feature configuration as shown in the listing below. It refers to the feature model `FM` from Listing 6. Note how the feature model being configured is defined in line no. 1. You can also see how cardinalities can be set in a configuration, e.g. `Simple[1]` sets the cardinality of feature "Simple" to [1], and how values can be assigned to parameterized features, e.g. with `Radar=510`.

Listing 9: Feature configurations.

```
1 config cfg of FM {
2     Wiper[1];
3     Simple[1];
4 
5     CruiseControl[1], Adaptive[1], Radar[1];
6 
7     Radar = 510;
8 }
```

3.2 Inclusion vs. Selection

It might seem that setting the cardinality of an optional feature to [1] within a configuration will suffice to effectively select this feature. This is not the case, however. An optional feature is only actually selected in a given configuration if its cardinality and those of all its predecessor features are set to [1]. For example, assume we had set `Adaptive[1]` and `Radar[1]` but `CruiseControl[0]`; then, we would eventually not have a radar because "Radar" cannot possibly be present in a car if "CruiseControl" is not, which we defined by setting the cardinality of "CruiseControl" to [0].

Obviously, we have to distinguish two things here: simply setting a feature's cardinality to [1] versus effectively selecting a feature by setting its cardinality and those of all its predecessor features to [1]. When dealing with feature configurations, it is of paramount importance to clearly keep these two cases apart. To facilitate this, we need a sound terminological foundation for configuring features:

- to *include* an optional feature
  = setting the feature's cardinality to [1]
- to *exclude* an optional feature
  = setting the feature's cardinality to [0]
- to *select* an optional feature
  = setting the feature's cardinality and the cardinalities of all its predecessor features to [1]
- to *deselect* an optional feature
  = setting the feature's cardinality to [0]

Deselecting is identical to excluding because in order to effectively deselect a feature "F" from a configuration it is enough to have a single feature on the path from the root feature to "F" with its cardinality set to [0]. We will see below, in Chapter 4, why it still makes sense to differentiate the two terms "exclude" and "deselect".

We have, then, a terminology to distinguish between merely including or effectively selecting a feature in a given configuration. But selection is still rather cumbersome because we have to set the cardinalities of all predecessors. To help out here, VSL provides the following shortcuts: `SomeFeature[+]` and `SomeFeature[-]`. With these at hand, we can now simplify Listing 9 from above as follows:

Listing 10: Selection in a feature configuration.

```
config cfg of FM {
    Simple[+];
    Radar[+];
    Radar = 510;
}
```

3.3 Resolving Ambiguities

In feature models, it is perfectly legal to use the same name several times for different features, as long as the same-named features are not root features or siblings, i.e. children of the same parent feature. But how can we then differentiate between such features within a configuration? Take this example:

Listing 11: Ambiguously named features.

```
featureModel FM {
    Wiper (
        [1] ( Simple, Advanced )
    );
    CruiseControl (
        [1] ( Simple, Advanced )
    );
}

config cfg of FM {
    Wiper.Simple[+];
    CruiseControl.Advanced[+];
}
```

To make an ambiguous feature identifier unique, you simply prepend its parent with the common dot notation. If the parent is not enough, i.e. the two same-named features even have same-named parents, you can also prepend the feature's grandparent, and so on, until you obtain a unique identifier.

3.4 Configuring Cloned Features

When it comes to configuration, cloned features introduce a significant increase in complexity. When configuring a feature model, we normally assume that we simply include or exclude elements that are already defined in the feature model—namely, the features. But with cloned features, we actually introduce new configurable elements within the configuration itself[3].

Listing 12 on the next page shows how to create instances of cloned features within a configuration. The feature model in this listing defines the variability of a fictitious wiper system and allows a highly flexible configuration of the interval modes available: any number of interval modes is allowed, and in each case the interval's duration in seconds is provided through the cloned feature's parameter.

Instances of cloned features are created with `ClonedFeature$instanceName` (see line 8 in Listing 12). Once created, they are treated as optional features, i.e. as if they were defined in the feature model with a cardinality of [0..1]. Therefore, they can be included, excluded, selected or deselected just like ordinary optional features. This is evident in line

[3]In fact, we have a full-blown instantiation here, cf. [3]

Listing 12: Configuring cloned features.

```
 1  featureModel FM {
 2      Wiper (
 3          IntervalMode[*] : float     // duration of interval in sec
 4      );
 5  }
 6
 7  config cfg of FM {
 8      IntervalMode$long;
 9      long:IntervalMode[+],  long:IntervalMode = 4;
10      IntervalMode$short;
11      short:IntervalMode[+],  short:IntervalMode = 1.2;
12  }
```

9, where the instance "long" of cloned feature "IntervalMode" is selected (which implicitly comprises an inclusion of feature "Wiper", cf. Section 3.2) and its parameter is set to 4, which means the wiping interval will have a duration of 4 seconds in this mode. Lines 10 and 11 create another mode with a shorter interval duration. As can be seen, we are now able to assign distinct values to the parameter of cloned feature "IntervalMode" for each of the two instances "long" and "short".

When creating an instance for a cloned feature, we not only create a single configurable element that represents the feature instance, but in addition the entire subtree below the cloned feature in the feature model will be available for configuration. The listing below contains a refinement of the above listing in which the cloned feature has two alternative child features and shows how they can be configured.

Listing 13: Configuring children of cloned features.

```
featureModel FM {
    Wiper (
        IntervalMode[*] ( [1] (
            Fixed : float,      // duration of interval in sec
            Flexible            // duration changeable by driver
        ))
    );
}

config cfg of FM {
    IntervalMode$short;
    short:IntervalMode.Fixed[1];
    short:IntervalMode.Fixed = 1.2;

    IntervalMode$long;
    long:IntervalMode.Flexible[+];
}
```

As in the previous section the common dot notation is applied in lines 12, 13 and 16, but this time it is used not to disambiguate a feature name but to denote a particular instance of a parent cloned feature (which could be seen as a kind of disambiguation, too). The pseudo syntax for specifying successors of cloned features is

```
in1:ClonedPred1. … .NonClonedPredN. … .in2:ClonedPred2. … .Feature
```

where `inN` denote instance names, `ClonedPredN` predecessor features that are cloned and `NonClonedPredN` any number of non-cloned predecessor features. As can be seen, cloned features may well have other cloned features among their children or successors. For example:

Listing 14: Cloned features with other cloned features among their successors.

```
 1  featureModel FM {
 2      Root (
 3          Child[*] (
 4              GrandChild[*] (
 5                  Example?
 6      )));
 7  }
 8
 9  config cfg of FM {
10      // creating an instance 'x' of GrandChild
11      // in scope of instance 'a' of Child
12      a:Child . GrandChild$x;
13
14      // selecting feature Example in instance 'x'
15      x:GrandChild . Example[+];      // ERROR: no instance of Child specified
16
17      // selecting feature Example in instance 'x' in instance 'a'
18      a:Child . x:GrandChild . Example[+];
19  }
```

Note that instance names have to be provided for *all* cloned predecessors of the feature in question. In line 15 above, for example, it is not sufficient to specify instance “x” of feature “GrandChild” because it is not clear which instance of feature “Child” we refer to: several instances of “Child” may contain an instance of “GrandChild” with name “x”. Line 18 shows how to correctly achieve this.

Of course, these configurations of cloned features can get cumbersome when having many cloned features containing other cloned features. But this is not a problem in practice because usually cloned features are only used sparingly in certain special use cases and CVM provides tool support for creating and editing configurations, so the configuration specifications need not be typed-in manually (the screenshot on page 49 shows this tool support for a very simple example).

4 Configuration Links

In this chapter, we take a closer look at configuration links, one of the core concepts of the CVM framework. As before, the emphasis will be on concepts and how they are represented in VSL, not on tooling or the use of the CVM editors.

The primary purpose of a *configuration link* is to define how to configure one feature model, referred to as the *target feature model*, depending on a given configuration of a second, referred to as the *source feature model*. With the information captured in a configuration link, we can then derive a configuration of the target feature model (a *target configuration*) from any given configuration of the source feature model (*source configuration*). This is illustrated by the solid arrow from left to right in Figure 2. Alternatively, a configuration link can also be thought of as defining a transformation from configurations of the source feature model to configurations of the target feature model.

$$\begin{tikzcd}
FM_{src} \arrow[r, dashed] \arrow[d, dash] \arrow[dr, phantom, "CL_{FM_{src} \rightarrow FM_{trgt}}"] & FM_{trgt} \arrow[d, dash] \\
C_{FM_{src}} \arrow[r] & C_{FM_{trgt}}
\end{tikzcd}$$

Figure 2: Relations defined by a configuration link $CL_{FM_{src} \rightarrow FM_{trgt}}$.

On a conceptual level, a configuration link also associates the two feature models—the source and target feature model—with respect to their configuration. This way, we can think of a configuration link as a conceptual link between two feature models, which is illustrated in Figure 2 by the dashed arrow from FM_{src} to FM_{trgt}.

In CVM, the technical concept for representing a configuration link is called *configuration decision model* (or CDM for short). Each such model is an aggregation of many *configuration decisions* (or CDs for short). Each of these configuration decisions captures a single decision on how to configure the target feature model, which may be conditioned on a particular configuration of the source feature model. Examples of configuration decisions might be

- *"whenever feature 'Comfort' is selected in the source feature model, select feature 'RainSensor' in the target"* or
- *"when having 'USA' in the source feature model, then choose feature 'CupHolder' in the target"*.

In much the same way as for requirements in requirements engineering, there is the general rule of configuration decision atomization, i.e. configuration decisions should be formulated such that they each capture a single smallest unit of information on how to configure the target feature model. Then, all the configuration decisions within a configuration decision model will together define how to overall configure the target feature model, depending on a given source configuration.

You might wonder what the difference is between a configuration link and a configuration decision model. The model is the technical representation, whereas the link refers to the association between the source and target feature model on a more conceptual level. More concretely, configuration decision models composed of configuration decisions are just one particular technique for realizing the abstract concept of a configuration link; but there might be other conceivable technical realizations for implementing configuration links, e.g. using model transformation.

4.1 Basics

You think an example says more than a thousand words? So let the following listing speak to you ...

Listing 15: A basic configuration decision model.

```
 1  featureModel CustomerFM {
 2      Type! ( [1] ( Classic, Comfort ) );
 3  }
 4  featureModel TechFM {
 5      Wiper! ( RainSensor? );
 6      CruiseControl?;
 7  }
 8
 9  configLink CDM ( CustomerFM --> TechFM ) {
10      Classic --> RainSensor[0];
11      Comfort --> RainSensor[1];
12      true --> CruiseControl;         // [1] is the default
13  }
```

We have two feature models, “CustomerFM” and “TechFM”, together with a configuration decision model called “CDM”. Think of “CustomerFM” as a customer-oriented view on our system’s variability, which is intended for direct configuration by the end customer. It defines a single mandatory feature “Type” with two alternative children “Classic” and “Comfort”, which represent the two model types we have on offer. Correspondingly, “TechFM” captures the technical view on our system’s variability: we always have a wiper with an optional rain sensor and we have an optional cruise control.

Configuration decision model “CDM” now defines how to configure “TechFM” depending on a given configuration of “CustomerFM”. Note the syntax in line 9 for specifying the source and target feature model of a configuration decision model. Lines 10 to 12 show three configuration decisions. On the left side of the “`-->`” keyword, an expression in propositional logic can be provided to define when the configuration decision is active. This is called the configuration decision’s *inclusion criterion*. The right-hand side then provides a comma-separated list of configurations on the target feature model which will be performed whenever the inclusion criterion evaluates to true. This is referred to as the decision's *effect*. Here, precisely the same syntax can be applied as outlined for feature configurations in Chapter 3 above (with a few additional capabilities discussed below as we move along). If a configuration decision's inclusion criterion evaluates to true we say that the configuration decision is *active* or that it *triggers*.

Let us briefly examine what is defined in "CDM" above: for model type "Classic", we do not have a rain sensor (line 10), while for type "Comfort" it will be included (line 11); the cruise control is always included because we set the inclusion criterion of the last decision to "`true`" (line 12). Taking this information together, we can now derive a configuration of our system's technical variability from any given end-customer configuration. The CVM framework provides tool functionality to automatically generate a target configuration from a given source configuration.

As noted before, this report is not intended to provide an extensive discussion of motivation and methodological considerations. However, the example from Listing 15 illustrates the methodological background of configuration links quite well, so let us take a quick look at methodology at this point. The source feature model "CustomerFM" allows us to package and present the system's technical variability to the customer; it thus provides an orthogonal view on our technical variability. This way, the technical variability can be presented in a suitable form for end customers, it can be partially simplified (as for "CruiseControl" above, which no longer appears in the customer-oriented model), and its overall viewpoint can be changed (in the example from technical to marketing). Most importantly, the structuring and configuration of the technical—we could say *internal*—variability can be hidden entirely from the customer (referred to as *configuration hiding* in the authors' earlier publications). Thus, the feature model "CustomerFM" becomes an interface for our variability management. Finally, this interface can be of use when it comes to evolution: when the internal "TechFM" changes, the "CustomerFM" can remain unaltered if "CDM" is adapted appropriately.

4.2 Inclusion vs. Selection (Revisited)

As pointed out above, when working with configuration decisions it is extremely important to carefully distinguish between inclusion versus selection and exclusion versus deselection[4]. And in doing so, we have to be aware of whether we are on the left side of a configuration decision, i.e. defining the inclusion criterion, or on its right side, i.e. defining its effect on the target configuration.

First of all, in the inclusion criterion you can also distinguish between inclusion/ exclusion versus selection/deselection. The syntax `SomeFeature[1]` will test if feature "SomeFeature" is included in the source configuration, i.e. if its cardinality is set to [1]. This will fail in two cases: if its cardinality has already been set to [0] or if it has not yet been set in this configuration, i.e. its configuration still remains undecided. By contrast, the syntax `SomeFeature[+]` used in an inclusion criterion will test whether

[4]We will use important terminology from Section 3.2 in the sequel, so you might want to read that section now, if you have not done so already.

Table 2: Summary of the different activities in configuring an optional feature.

	Syntax	**Semantics**	
		within inclusion criterion of configuration decisions	within feature configurations *or* effect of configuration decisions
Inclusion	`Feature[1]`	Is cardinality of feature set to [1] ?	Set cardinality of feature to [1] !
Exclusion	`Feature[0]`	Is cardinality of feature set to [0] ?	Set cardinality of feature to [0] !
Selection	`Feature[+]`	Is cardinality of feature **and all** its predecessors set to [1] ?	Set cardinality of feature **and all** its predecessors to [1] !
		(check if / make sure that feature will appear in final configuration)	
Deselection	`Feature[-]`	Is cardinality of feature **or any** of its predecessors set to [0] ?	Set cardinality of feature to [0] !
		(check if / make sure that feature won't appear in final configuration)	

feature "SomeFeature" is selected in the source configuration, i.e. if its cardinality and that of all its predecessor features is set to [1]. Finally, the propositions `SomeFeature[0]` and `SomeFeature[-]` can be used to test if a feature is excluded or deselected in the source configuration.

Table 2 presents an overview of all the different ways to test for or effectuate a particular configuration of an optional feature. The second semantics column (the rightmost column in the table) shows what inclusion, selection, etc. mean when performing configurations and therefore merely recapitulates what we have defined above in Section 3.2 on page 18. The next column to the left shows what inclusion, selection, etc. mean when checking an existing configuration.

Consider the following snippet (in which feature name "Ps" stands for "Parent in source", "Cs" for "Child in source", and so on):

Listing 16: Inclusion vs. selection in CDMs.

```
1  featureModel FMsrc  {  Ps ( Cs );  }
2  featureModel FMtrgt {  Pt ( Ct );  }
3
4  configLink CDM ( FMsrc --> FMtrgt ) {
5      Cs[1] --> Ct[1];
6      Cs[0] --> Ct[0];
7      Cs[+] --> Ct[+];
8      Cs[-] --> Ct[-];
9      Cs --> Ct;
10 }
```

The configuration decisions in lines 5 and 6 will trigger whenever feature "Cs" is set to [1] or [0], respectively, in the source configuration. That in line 7 will trigger only if both feature "Cs" **and** "Ps" are set to [1]. That in line 8 will trigger if feature "Cs" **or** "Ps" is set to [0]. The effects, on the other hand, are as follows: the configuration decisions of lines 5 and 6 will only set the cardinality of "Ct" (to [1] and [0], respectively). That of line 7 will set both "Ct" and "Pt" to [1], whereas that of line 8 will only set "Ct" to [0] because this is sufficient for making sure that "Ct" is definitely removed from the configuration. This is just a direct application of the semantics given in Table 2. But what does line 9 mean, then? The kind of configuration is not explicitly stated here, so the default will apply: within an inclusion criterion the default is `[+]`, i.e. test for selection of the feature; within the effect the default is [1], i.e. inclusion of the feature. Thus, line 9 means that whenever "Cs" is selected, i.e. its cardinality and that of all predecessors is set to [1], feature "Ct" will be included, i.e. its cardinality will be set to [1].

At first sight all the different cases might seem rather intricate. However, for normal everyday work, it is sufficient to keep in mind the following guideline:

> **Within an inclusion criterion:**
> use inclusion/exclusion whenever you want to check if a feature's cardinality is set to [1] or [0], and use selection/deselection whenever you want to check if it is certain that the feature will appear in the configuration.
> **Within the effect of a configuration decision (or feature configuration):**
> use inclusion/exclusion whenever you want to set a feature's cardinality, and use selection/deselection whenever you want to make sure, in a single step, that the feature will appear in the configuration.

You could also say that inclusion/exclusion does not take into account a feature's predecessors, whereas selection/deselection also takes into account the predecessors and their effect on the lower-level feature.

4.3 Redundancy

It is perfectly legal to have redundant configuration decisions in a single configuration decision model. That is a deliberate design decision and an important prerequisite for several intended use cases of compositional variability management, which we cannot explore in detail here. A simple example of such a redundancy might be:

Listing 17: Redundant configuration decisions.

```
1  featureModel CustomerFM  {
2      Market! ( [1] ( USA, Canada, Europe, Japan ) );
3  }
4  featureModel TechFM {
5      CruiseControl? ( Adaptive? );
6  }
7
8  configLink CDM ( CustomerFM --> TechFM ) {
9      USA | Canada --> CruiseControl[+];  // customer expectation
10     Canada       --> Adaptive[+];       // local legislation
11     USA | Canada --> Adaptive[+];       // competitors have it
12 }
```

The configuration decisions in lines 10 to 12 are highly redundant. Line 12 comprises the effects of lines 10 and 11, so we could delete lines 10 and 11 without changing the overall configuration of “TechFM” as defined by “CDM”. However, when looking at the rationale behind each of these three configuration decisions, the benefit of these redundant specifications becomes clear: line 10 was defined by Ted from marketing, saying “All North American cars should have cruise control because customers simply expect that feature there”; Al from the legal department added line 11 to reflect that local legislation in Canada requires adaptive cruise control (what ever the reason; we don’t want to question the lawmakers’ wisdom, right?); finally, Pete from management pointed out that all our competitors on the North American market have adaptive cruise control, so we should, too (line 12). In the methodology underlying CVM, such factors that drive our configuration are called *configuration considerations*.

Now, if we had all these three configuration considerations compiled into a single configuration decision `USA | Canada --> Adaptive[+]`, this would be a nightmare to maintain and evolve. Having made the consideration explicit, however, we can deal with change more easily: if Canada were to drop its law requiring adaptive cruise control, we would simply remove or comment out line 11 but would see that there were other good reasons for shipping Canadian cars with adaptive cruise control, namely our competitors, as defined in line 12. Similarly, if our competitors changed their strategy, we would remove line 12 and in this case we would see that we can now ship standard cruise control in the U.S., whereas in Canada we must stick to the adaptive variant.

There are several other benefits from explicitly defining all configuration considerations in distinct configuration decisions, even if this may lead to redundancy. For example, when adding a new configuration decision to a large, long-evolved configuration decision model, it is not uncommon to spot some conflict between the newly added and some legacy decisions. If the legacy configuration decisions each reflect a single configuration consideration (and not complex composite rules into which a lot of diverse considerations were merge in), it is a straightforward matter to resolve the conflict based on the rationale documented for the legacy decisions.

4.4 Contradictions

If we allow redundant configuration specifications within a single configuration decision model, this can, of course, lead to contradictions. The formalization of the CVM concepts defines several terms to cope with such situations in a well-defined manner. We only look at the most important cases here:

Listing 18: Contradiction #1: including and excluding a feature.

```
featureModel TechFM {  CruiseControl? ( Adaptive? );  }

configLink CDM (  --> TechFM ) {
    true --> CruiseControl[1];
    true --> CruiseControl[0];
}
```

Note that we do not provide a source feature model of “CDM” here, which is legal as long as all the configuration decision model’s decisions only have `true` or `false` as their inclusion criterion. Here, we simultaneously include and exclude feature “CruiseControl” (lines 4 and 5, respectively). This form of contradiction is resolved on the level of the core definition of configuration decisions by the following rule

An exclusion is prioritized over an inclusion.

In other words, whenever we have an inclusion and an exclusion of the same feature, the feature will be excluded. In the example, "CDM" will thus exclude feature "CruiseControl", i.e. set its cardinality to [0].

As a side remark, it should be noted that this is only the simplified special case of a more general rule. In the effect of a configuration decision, we are actually allowed to "narrow" a feature's cardinality; for example, if for feature "SomeFeature" the feature model defines a cardinality of [2..8], we may narrow this cardinality in the effect of a configuration decision to any subset of [2..8], maybe [2..4, 8]. The general rule for resolving contradictions in setting cardinalities states that the intersection of all cardinalities will be set or [0] will be set if the intersection is the empty set.

The second major contradiction results from the tree structure of the target feature model, illustrated in this listing:

Listing 19: Contradiction #2: including a child and excluding a predecessor.

```
1  featureModel TechFM {  CruiseControl? ( Adaptive? );  }
2
3  configLink CDM (  --> TechFM ) {
4      true --> Adaptive[1];
5      true --> CruiseControl[0];
6  }
```

Here, we include the child while excluding one of its predecessors (its direct parent in this simple example). This contradiction is *not* resolved by definition. This means that in the target configuration everything is configured exactly as stated here: "Adaptive" will have its cardinality set to [1] and "CruiseControl" to [0]. Of course, this will mean that "Adaptive" is not selected in the configuration (because not all its predecessors have their cardinality set to [1]), but we don't care. This is perfectly in line with what we have said about inclusion versus selection: line 4 only includes, it does not select, feature "CruiseControl", so whether the feature will eventually appear in the variant is partly up to the configuration of its predecessor features which are not configured in line 4.

To deal with such contradictions in this way proves highly valuable when specifying large configuration decision models in a real-world context. You can define when the cruise control will be adaptive in one place, and in some other part of your model you can define when your complete system will have the cruise control at all. The considerations that drive the latter question won't need to be copied into the condition for having the adaptiveness. This allows, to some extent, a separation of concerns within a configuration decision model.

One more contradiction is worth mentioning here. Strictly speaking, it is merely a combination of the two previous ones:

Listing 20: Contradiction #3: selecting a child and excluding a predecessor.

```
1 featureModel TechFM {  CruiseControl? ( Adaptive? );  }
2
3 configLink CDM (  --> TechFM ) {
4     true --> Adaptive[+];
5     true --> CruiseControl[0];
6 }
```

This is almost the same situation as before, but this time we select(!) feature "Adaptive". To help understand such situations, there exists a simple rule of thumb: selection of a feature is treated exactly as an inclusion of this feature and all its predecessors (in fact this is simply an application of what is defined in Table 2). Now, when interpreting `Adaptive[+]` as `Adaptive[1]` and `CruiseControl[1]`, we can resolve the contradiction between the inclusion of "CruiseControl" resulting indirectly from line 4 and its exclusion from line 5 by applying the priority rule from above: exclusion has priority over inclusion; so "CruiseControl" will be excluded, i.e. have its cardinality set to [0]. More succinctly:

A selection does not override the exclusion of a predecessor.

You might wonder about another common form of contradiction: two features might be selected in the target feature model for which, in the target feature model, a feature link of type "excludes" was defined (or similar situations). Such contradictions are not disallowed by definition. You simply have defined a configuration link that creates invalid target configurations. This is called an *inconsistency* in the configuration decision model.

4.5 Several Sources or Targets

A simple but powerful extension to the core concept of configuration links is to allow several source and/or several target feature models.

Listing 21: Several source and target feature models.

```
featureModel CustomerFM { Type! ( [1] ( Classic, Comfort )); }
featureModel PredefFM   { Market! ( [1] ( USA, Europe, Japan )); }
featureModel TechFM1    { CruiseControl? ( Adaptive? );  }
featureModel TechFM2    { Wiper! ( RainSensor? );  }

configLink CDM ( CustomerFM, PredefFM --> TechFM1, TechFM2 ) {
    Comfort & USA --> Adaptive[+], RainSensor[+];
    // more ...
}
```

In this example, such configurations, which are usually not a deliberate choice of the end customer, e.g. the country in which the car is being bought, were factored out of the "CustomerFM" feature model and put in a separate source feature model "PredefFM". The choice of a particular supplier for a certain subsystem might be another configuration that could go into this separate source model. On the target side, we have two feature models, each of which defines the variability of a particular subsystem.

In the case of name conflicts, the feature model can be specified in the inclusion criteria and effects:

Listing 22: Ambiguities with several target feature models.

```
1 featureModel CustomerFM { Type! ( [1] ( Classic, Comfort )); }
2 featureModel FrontWiperFM {  RainSensor?;  }
3 featureModel RearWiperFM  {  RainSensor?;  }
4
5 configLink CDM ( CustomerFM --> FrontWiperFM, RearWiperFM ) {
6     Comfort --> RainSensor;     // ERROR: ambiguous feature id !!!
7     Comfort --> FrontWiperFM#RainSensor;
8 }
```

Line 6 will cause an error because it is not clear whether `RainSensor` refers to that of the front or rear wiper. Line 7 shows how to resolve such an ambiguity by stating a feature model. In addition, it is possible to provide names for the source and target feature models for this purpose, so we could define “CDM” above as

```
5 configLink CDM ( CustomerFM --> front of FrontWiperFM,
6                                 rear of RearWiperFM ) {
7     Comfort --> front#RainSensor;
8 }
```

The scope of these names is the configuration decision model in which they are declared, i.e. they are similar to local variables.

In fact, what we can do is to use a single feature model twice as source or target. This way, we can get rid of the ugly duplication of “FrontWiperFM” and “RearWiperFM” above. This is what it looks like:

Listing 23: Using a single feature model twice.

```
1 featureModel CustomerFM { Type! ( [1] ( Classic, Comfort )); }
2 featureModel WiperFM {  RainSensor?;  }
3
4 configLink CDM ( CustomerFM --> front of WiperFM,
5                                 rear of WiperFM ) {
6     Comfort --> RainSensor;     // ERROR: ambiguous feature id !!!
7     Comfort --> front#RainSensor;
8 }
```

We will make extensive use of this capability when dealing with variable entities and compositional variability management in Chapter 5.

4.6 Applying Configuration Decision Models

As already noted above, given a configuration decision model, it is possible to derive target configurations from source configurations. More precisely, given a configuration decision model with source feature models FM^1_{src} to FM^n_{src} and target feature models FM^1_{trgt} to FM^m_{trgt}, we need to supply n feature configurations C^i_{src} of FM^i_{src} with $1 \leq i \leq n$ and obtain m feature configurations C^j_{trgt} of FM^j_{trgt} with $1 \leq j \leq m$. This is called *applying* the configuration decision model *on* the source configurations C^1_{src} to C^n_{src}. Note how this can be compared to a method call, where we have to supply several parameters of a particular type, in our case several source configurations of a particular feature model each. This suggests that we can think of feature configurations as variables and feature models as their type, which is an analogy that will prove very helpful in understanding the Chapter 5 below.

The CVM framework supplies implementations for applying configuration decision models. These also support partial configurations on the target side, which will result in a partial application of the configuration specifications captured in the configuration decision model, which in turn will usually result in partial target configurations. Furthermore, CVM also supports *incremental configuration* of target feature models through configuration links. This is supported by allowing the user to also supply target configurations when applying a configuration decision model, that may already contain partial configurations.

4.7 Configuration Graphs

Configuration links as defined above—and as realized by configuration decision models—allow the target(!) configurations of one configuration link to be used as source(!) configurations of another configuration link. This way, it is possible to build chains of consecutive configuration links. Combined with the possibility of incremental target configuration as outlined above, this also enables us to build networks of configuration links or, more precisely, directed acyclic graphs (DAGs) in which the nodes represent feature configurations and the edges are realized by configuration links. These are called *configuration graphs* or *configuration networks*. Figure 3 shows an example.

In this figure, C_{Name} stands for a configuration of feature model FM_{Name}. The two marketing-related source configurations $C_{Customer}$ and C_{Predef} capture end-customer configuration and predefined, customer-invisible configuration, respectively (as in the

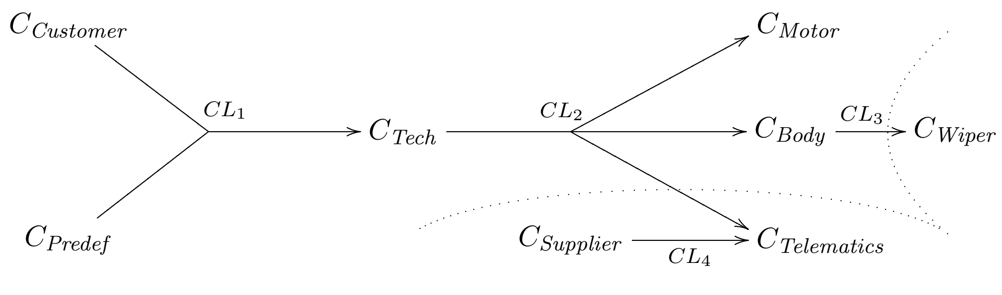


Figure 3: Four configuration links forming a configuration graph.

example of Listing 21). These two, together, drive the configuration of our company's core technical feature model in C_{Tech}. This target configuration of the first configuration link is in turn used as a source configuration for the next configuration link, which defines how to configure the local feature models of our three subsystems: motor control, body electronics and telematics. The wiper system within the body electronics system is provided by an external supplier S_1, who has his own feature model FM_{Wiper}, which is related to the body system variability with another configuration link. The telematics system, too, is developed externally by a second supplier S_2. Here, the manufacturer's core technical configuration C_{Tech} is not sufficient to fully configure S_2's $C_{Telematics}$ because the supplier's telematics system contains variability that the supplier provides only for other clients and hides from our company. To bind this additional variability, he uses $C_{Supplier}$ as an input to yet another configuration link to incrementally configure $C_{Telematics}$ in order to obtain a full configuration. For strategic reasons, S_2 hides this from the manufacturer.

Note that such a feature model managed by an external supplier—as in the example FM_{Wiper} and $FM_{Telematics}$—or, of course, its corresponding configuration, actually constitutes a product line in its own right. We can thus perceive the overall product family setting, as sketched in Figure 3, as an integration of three product lines: that of the manufacturer, and those of the suppliers. From that point of view, configuration links become a means for partitioning a complex product family into several smaller, subordinate product lines.

5 Variable Entities

Applying configuration decision models manually—i.e. manually specifying each time the source to be used as input and the target configurations to write the output to—can be a tedious process. This is especially true if you need to derive target configurations frequently or for a great number of different source configurations, or if you do not have a single configuration link but a chain of consecutive configuration links or even networks of them. To help out here, CVM provides the concept of *variable entities*, which is presented in this chapter. They enable us to predefine chains and networks of feature configurations that are interrelated by configuration links. Based on that, you only need to supply a configuration of a few overall source configurations, and all the configuration links within the network will be applied automatically in a single step. To emphasize this configurability, we sometimes synonymously speak of *configurable entities*.

5.1 Basics

A *variable entity* in CVM represents some unit within an overall product family setting. This may be the entire product family, the complete system from a technical perspective, a particular subsystem or some unit without a direct correspondence on the technical level, such as the aggregation of all subsystems provided by a certain external supplier. More concrete examples will be provided below.

To define the variability within such a variable entity, one or more feature configurations are used. This is illustrated in the following listing.

Listing 24: A simple variable entity.

```
featureModel WiperFM  {
    RainSensor?, FlexibleIntervalMode?;
}

entity WiperSystem {
    config cfg of WiperFM;
}
```

Here, a wiper system is described and its variability is defined by feature configuration “cfg” of feature model “WiperFM”. Note that the entity’s variability is *not* directly defined by a feature model, but instead by a feature configuration. This follows the analogy of variables and types encountered before: think of “WiperSystem” as a class, of configuration “cfg” as a member variable, and of “WiperFM” as this variable’s type. The benefit of this becomes obvious when extending the example to distinguish between a front and a rear wiper within our wiper system:

Listing 25: The wiper system now with a front and rear wiper.

```
entity WiperSystem {
    config frontCfg of WiperFM;
    config rearCfg of WiperFM;
}
```

Let us return to Listing 24, in which another very important aspect is to be noted. Line 6 refers to feature model “WiperFM”, which is defined outside the context of the entity “WiperSystem”. This is fine in some use cases, but there is another option: an entity can bring along the definition of the feature models that type its configurations. The following listing modifies Listing 24 accordingly:

Listing 26: A more self-contained definition of the wiper system entity.

```
1 entity WiperSystem {
2     featureModel WiperFM  {
3         RainSensor?, FlexibleIntervalMode?;
4     }
5 
6     config cfg of WiperFM;
7 }
```

Here, the feature model definition of “WiperFM” within “WiperSystem” can be compared to an inner class, as available in many object-oriented programming languages such as Java. At this point, it is essential to realize that even if the feature model definition now resides inside the entity definition—in lines 2 to 4 in the listing—it still does *not* directly define variability of the containing entity, i.e. the wiper system. The above rule stating that an entity’s variability is defined by its feature configurations—not feature models—applies without change. Thus, when defining the wiper system as

Listing 27: Without a feature configuration, the wiper system becomes invariant.

```
entity WiperSystem {
    featureModel WiperFM  {
        RainSensor?, FlexibleIntervalMode?;
    }
}
```

we ended up with a wiper system that does not “contain” any variability, i.e. an invariant wiper system. The inner class analogy holds here, too: an inner class will make an impact on its outer class only if it is used as the type of a member variable; otherwise it will merely be a declaration which may or may not be used elsewhere, for example in a method or outside the containing class.

Let us conclude this section with a summary of the most important facts about the concept of an entity:

- CVM entities represent units of variability management within an overall product family setting,
- The variability within such an entity is defined by an arbitrary number of contained feature configurations; an entity with such contained feature configurations is called a variable entity, otherwise an invariant entity,
- Within an entity, additional feature models can be defined, or the entity can refer to feature models defined outside its scope.

5.2 Applications

The introduction to this chapter promised to provide means to define chains or graphs of feature configurations interrelated by configuration links. So far, an entity only provides for defining the nodes of such a graph: its contained feature configurations. Let us now add the edges.

To illustrate this, we will take up the example of Listing 15 from Section 4.1 above. There, two feature models "CustomerFM" and "TechFM" were related by a configuration decision model. To define this overall arrangement in a reusable form, we can use an entity with an explicitly defined configuration decision model application, or *CDM application* for short:

Listing 28: A variable entity with an explicit CDM application.

```
 1  featureModel CustomerFM {
 2      Type! ( [1] ( Classic, Comfort ) );
 3  }
 4  featureModel TechFM {
 5      Wiper! ( RainSensor? );
 6      CruiseControl?;
 7  }
 8
 9  configLink CDM ( CustomerFM --> TechFM ) {
10      Classic --> RainSensor[0];
11      Comfort --> RainSensor[1];
12      true --> CruiseControl;         // [1] is the default
13  }
14
15  entity E {
16      config customerCfg of CustomerFM;
17      config techCfg of TechFM;
18
19      apply CDM (customerCfg --> techCfg);
20  }
```

The feature models and the configuration decision model were copied from Listing 15 without change. Entity "E" has two feature configurations, one for each of the two feature models, and states in line 19 that the configuration "techCfg" should always be derived from configuration "customerCfg" by applying configuration decision model "CDM". The VSL syntax of such an application is pretty straightforward:

⟨*application*⟩ → '**apply**' ⟨*name*⟩ '**(**' [⟨*names*⟩] '**-->**' ⟨*names*⟩ '**)**'

⟨*names*⟩ → ⟨*name*⟩ ('**,**' ⟨*name*⟩)∗

The first ⟨*name*⟩ must denote a configuration decision model, the others inside the parentheses must refer to feature configurations. The keyword `-->` separates the source and target configurations. The number and type of source and target configurations must match the source and target feature models of the configuration decision model being applied.

Note again how this corresponds to a method call: we have to provide a certain number of parameters—the feature configurations—that must each match a particular type—the corresponding feature model. The main difference is that we have several return values here, i.e. the target configurations, in much the same way as for methods that provide output via parameters which were passed by reference.

Having only a single configuration decision model applied in an entity is not too exciting. In fact, we do not gain much. However, the mechanism presented here can be extended to chains and graphs of feature configurations of arbitrary size. To briefly illustrate this, we define the configuration graph from Figure 3 on page 32 in the form of a variable entity (feature models not shown):

Listing 29: The configuration graph of Figure 3 defined in VSL.

```
1  entity E {
2      config customerCfg of CustomerFM;
3      config predefCfg of PredefFM;
4
5      config techCfg of TechFM;
6
7      config motorCfg of MotorFM;
8      config bodyCfg of BodyFM;
9      config wiperCfg of WiperFM;
10     config telematicsCfg of TelematicsFM;
11     config supplierCfg of SupplierFM;
12
13     configLink CL1 ( CustomerFM, PredefFM --> TechFM );
14     configLink CL2 ( TechFM --> MotorFM, BodyFM, TelematicsFM );
15     configLink CL3 ( BodyFM --> WiperFM );
16     configLink CL4 ( SupplierFM --> TelematicsFM );
17
18     apply CL1 ( customerCfg, predefCfg --> techCfg );
19     apply CL2 ( techCfg --> motorCfg, bodyCfg, telematicsCfg );
20     apply CL3 ( bodyCfg --> wiperCfg );
21     apply CL4 ( supplierCfg --> telematicsCfg );
22 }
```

Note that a single feature configuration can be used as a target in several applications, as is the case for “telematicsCfg”, which is the target for “CL2” and “CL4”.

The listing above might seem rather lengthy. In particular the fact that we distinguish between the definition of a configuration decision model and its application leads to the two specifications of source/target feature models (lines 13 through 16) and source/target feature configurations (lines 18 through 21), which might seem redundant. However, if this were done in a single step, we would lose a great deal of flexibility: a configuration decision model could only be applied once in a single entity, and thus its configuration information could not be reused in other contexts. For simple standard cases, however, there is a convenient short form, which will be described below in Section 5.5.

5.3 Visibility

For all contents of an entity, a visibility, either `public` or `private`, can be provided. This is useful for separating externally visible constituents from purely internal ones. To motivate and illustrate this, we should briefly revisit our sample wiper system from Listing 24. Let us assume that this wiper system had some additional, more complex variability. For example, we could think of a cloned feature `IntervalMode[2..8]:float` that allows us to flexibly specify the number of interval modes of the wiper and the precise interval duration in each case (as explained in Section 2.5 on page 10). In practice however, we know that the wiper system should only be used with particular sets of interval modes: a standard and an advanced one.

To represent this situation in VSL, we could, of course, simply add the additional feature `IntervalMode[2..8]:float` to "WiperFM" in Listing 24, but this would publish this enormous variability—and thus complexity—to the environment and would in no way reflect the fact that we only want to use two particular sets of interval modes. A better solution would be to clearly separate an external and internal view on our variability:

Listing 30: Visibilities of the constituents of a variable entity.

```
entity WiperSystem {
    public featureModel WiperFM  {
        RainSensor?, FlexibleIntervalMode?;
        AdvancedIntervalModes?;         // added as public view on
                                        // additional variability
    }
    private featureModel ModesFM {
        IntervalMode[2..8] : float;     // duration in sec
    }

    public config cfg of WiperFM;
    private config cfgModes of ModesFM;

    apply ModeBinding ( cfg --> cfgModes);
    private configLink ModeBinding ( WiperFM --> ModesFM ) {
        --> IntervalMode$medium, medium:IntervalMode = 2.1;
        --> IntervalMode$long, long:IntervalMode = 4;

        // medium interval always available
        --> medium:IntervalMode[1];
        // long interval only in advanced mode
        AdvancedIntervalModes --> long:IntervalMode[1];
    }
}
```

Note how a simplified view on our internal variability was added to the public "WiperFM" by way of the optional feature "AdvancedIntervalMode". The actual, complex variability we require to configure our implementation is captured in the private feature model "ModesFM" and thus hidden from the outside. Configuration decision model "Mode-

Binding" defines how to configure the internal, complex variability depending on a given configuration of the simplified public view on that variability. So clients of the wiper system will simply configure feature configuration "cfg", and a configuration of the internal variability will then be derived from that and stored in "cfgModes", as stated by the application in line 14.

This way, configuration "cfg" together with its feature model "WiperFM" becomes a sort of "interface for variability" of the wiper system. This interface hides the details of the wiper's internal variability (as defined by "ModesFM" and "cfgModes") as well as its binding (as captured in "ModeBinding"). Such an interface for variability provides several of the benefits that are generally associated with interfaces in other areas of computer science, such as interfaces in programming languages [6]. In particular, they provide a sort of information hiding related to variability and variability binding, which could be called *configuration hiding*.

5.4 Entity Composition

Entities may be composed of other entities. For example, based on the wiper system from Listing 30 in the previous section, we can define the body electronics system as:

Listing 31: Composing entities of subentities.

```
 1  entity BodySystem {
 2      public featureModel BodyFM {
 3          Wiper! ( [1] ( Simple, Advanced ) );
 4      }
 5
 6      public config cfg of BodyFM;
 7
 8      private part ws of WiperSystem;
 9
10      apply BodyBinding ( cfg --> ws.cfg ) ;
11      private configLink BodyBinding (BodyFM-->WiperSystem.WiperFM){
12          --> FlexibleIntervalMode[0];    // never built in
13          Advanced --> RainSensor;
14          Advanced --> AdvancedIntervalModes;
15      }
16  }
```

Two things are particularly noteworthy here. First, line 8 defines a so-called *entity part*, thus stating that each body electronics system contains a wiper system called "ws" (for wiper system). By declaring this part private, the outside world will know nothing about this constituent of the body electronics system and can only configure it indirectly through the public configuration "cfg" of "BodySystem". Second, note how line 11 denotes the target feature model: since "WiperFM" was defined inside the "WiperSystem" entity, we need to use the dot notation here. Similarly, `cfg` in line 10 refers to the feature configuration "cfg" of "BodySystem", whereas `ws.cfg` refers to the feature configuration with the same name of "WiperSystem".

This approach of organizing a system's overall variability within a number of configurable units that are composed of one another—i.e. the variable entities—was called *compositional variability management* in the authors' previous publications. A detailed description of this conception, including use cases and further methodological considerations, can be found in [6].

5.5 Anonymous Feature Models and Configuration Decision Models

As mentioned above, the syntax presented so far can be quite lengthy. An abbreviation is therefore provided for the most common use cases. To get a quick impression, compare the listings on the next two pages: the first employs the ordinary syntax, as previously introduced, and the second makes use of the abbreviated form. The two listings are equivalent.

Very often, you have a variable entity with only a single public feature configuration, which is typed by a public feature model that is used solely for this purpose. In the listing on page 42, the entities "RainSensor", "WiperController" and "WiperMotor" are good examples of this. Here, it would be nice to be able to define the feature model and its corresponding configuration in a single step.

More importantly, you often encounter variable entities that contain many other lower-level entities of this simple structure (i.e. only a single public configuration). In that case, the configuration link declaration (lines 36 to 40 on page 42) and its application (lines 33 and 34) seem very obvious: the configuration of the containing entity, i.e. "cfg" of "WiperSystem", is always used as source configuration in these cases, and the public configurations of all contained entities make up the target configurations.

This standard case can be conveniently formulated in VSL with *anonymous feature models* and *anonymous configuration decision models*, which may only appear within entities. When defining a feature model without a name in an entity—hence the term "anonymous" feature model—a public feature configuration typed by this feature model will implicitly be created for the containing entity. Examples are provided in Listing 33 on page 43: "RainSensor" and the other entities are only defined by a single anonymous feature model. Similarly, an unnamed configuration decision model defined within an entity—hence the term "anonymous" configuration decision model—does not have its source and target feature models defined explicitly. Instead, it always uses the anonymous feature model of the containing entity as source feature model and the anonymous feature models of all visible lower-level entities as target feature models. In addition, such an anonymous configuration decision model is always automatically applied on the implicit configuration of the related entities.

Based on this definition of anonymous feature models and anonymous configuration decision models, we can state that Listing 33 is equivalent to Listing 32. Compare the way each of the four entities is defined in both listings. Of course, this abbreviation can only be used in standard cases; to formulate special arrangements of interrelated configurations, we have to resort to the more powerful syntax of named configuration decision models with explicit application. As another option, the two styles of specification can also be combined as explained in Section 6.3.

Listing 32: Entity with several subentities in normal syntax.

```
entity RainSensor {
    public featureModel RainSensorFM {
        Sensitivity[1] : float;
    }
    public config cfg of RainSensorFM;
}

entity WiperController {
    public featureModel WiperControllerFM {
        Wiper[1] ( [1](Simple, Advanced) );
    }
    public config cfg of WiperControllerFM;
}

entity WiperMotor {
    public featureModel WiperMotorFM {
        AutoReturn[0..1];
    }
    public config cfg of WiperMotorFM;
}

entity WiperSystem {
    public featureModel WiperSystemFM {
        HighEnd;
    }
    public config cfg of WiperSystemFM;

    part rs of RainSensor;
    part ctrl of WiperController;
    part frontMotor of WiperMotor;
    part rearMotor of WiperMotor;

    apply WiperBinding (cfg --> rs.cfg, ctrl.cfg, frontMotor.cfg,
                                rearMotor.cfg );

    configLink WiperBinding (WiperSystemFM -->
                        RainSensor.RainSensorFM,
                        WiperController.WiperControllerFM,
                        frontMotor of WiperMotor.WiperMotorFM,
                        rearMotor of WiperMotor.WiperMotorFM) {
        HighEnd --> Advanced;
        not HighEnd --> Simple;

        true --> frontMotor#AutoReturn[0],
                 rearMotor#AutoReturn[1];
    }
}
```

Listing 33: Entity with several subentities in brief syntax.

```
entity RainSensor {
    featureModel { Sensitivity[1] : float; }
}

entity WiperController {
    featureModel { Wiper[1] ( [1](Simple, Advanced) ); }
}

entity WiperMotor {
    featureModel { AutoReturn[0..1]; }
}

entity WiperSystem {
    featureModel { HighEnd; }

    part rs of RainSensor;
    part ctrl of WiperController;
    part frontMotor of WiperMotor;
    part rearMotor of WiperMotor;

    configLink {
        HighEnd --> Advanced;
        not HighEnd --> Simple;

        true --> frontMotor#AutoReturn[0],
                 rearMotor#AutoReturn[1];
    }
}
```

5.6 Entity Configurations

Based on the definition of a variable entity, such as the one for "WiperSystem" in Listing 33, we can construct a so-called *entity configuration.* The entity configuration for some entity E will contain one feature(!) configuration for every feature configuration defined in entity E and one nested, lower-level entity(!) configuration for each entity part defined in E. Thanks to this recursive definition, an entity configuration will eventually aggregate feature configurations for binding the variability in all variable entities of the complete system and will thus capture the entire, overall configuration of a given product family setting. Such an entity configuration thus has the character of an instance tree for variability.

For example, the entity configuration for entity "WiperSystem" from page 43 will contain one feature configuration for each feature configuration defined in the entity—either explicitly or implicitly through anonymous feature models. In this case, no feature configurations have been defined explicitly but there is an anonymous feature model (line 14), so our entity configuration will contain a single feature configuration typed by this anonymous feature model. Next, the entity configuration will get a lower-level entity configuration for each entity part defined in "WiperSystem". In our case, we have the four parts "rs", "ctrl", "frontMotor" and "rearMotor". Our entity configuration for "WiperSystem" will thus contain a single feature configuration and four nested entity configurations, each of which will contain a feature configuration of the corresponding anonymous feature model. Note that eventually "frontMotor" and "rearMotor" get a distinct feature configuration each, even though the corresponding parts refer to the identical entity "WiperMotor".

Moreover, the CDM applications defined in the entities—either explicitly with the `apply` keyword or implicitly through anonymous configuration decision models—will enable us automatically derive the lower-level feature configurations from the one or more top-level configurations. Here as well it is possible to employ partial configuration, which means that the automatic derivation might bind the lower-level variability only partially, leaving the remaining variations to the engineer for manual configuration.

6 Advanced Considerations

Chapters 2 to 5 focused on the core concepts in CVM and discussed only the most important usage aspects. This chapter deals with some selected further topics, which are not treated exhaustively here. They are merely intended to give an impression of what lies beyond the basic core concepts outlined above.

6.1 Optional Entity Parts

In the standard case of compositional variability management, a variable entity's configuration decision model is only used to configure lower-level feature models. For example, in Listing 33 on page 43 the configuration decision model of "WiperSystem" configures the feature models of the rain sensor, the controller and the two motors. But what if we want to specify that an entire subentity is optional? In the example, this could mean that the rain sensor should only appear in the "WiperSystem" in certain cases—depending on the configuration of the wiper system's public feature model—and should be entirely removed from the system in all other cases. This can be realized with *optional entity parts*, i.e. we mark the entity part "rs" in "WiperSystem" as being optional. The following listing is based on Listing 33 above and changes "WiperSystem" accordingly:

Listing 34: Entity part "rs" is now optional.

```
13  entity WiperSystem {
14      featureModel { HighEnd; }
15
16      part rs? of RainSensor;      // the ? makes 'rs' optional
17      part ctrl of WiperController;
18      part frontMotor of WiperMotor;
19      part rearMotor of WiperMotor;
20
21      configLink {
22          HighEnd --> rs[+];
23      }
24  }
```

By appending a `?` to the definition of entity part "rs", we stated that the rain sensor may be entirely removed from the wiper system in some configurations. Note how we formulate the selection and deselection of this optional entity part in the configuration decision model: `rs[+]` (in line 22). Compare this to the configuration of an anonymous feature model of part "rs", for example: `rs#Sensitivity=12`.

6.2 Configuration Across Hierarchy Levels

The conception of compositional variability management would be entirely infeasible in practice if we had to introduce a public feature model on each level in our system's composition hierarchy because this would significantly increase complexity without clear benefit. Instead, we want to introduce a public feature model only in cases, where configuration hiding—i.e. having an "interface" for variability management that encapsulates the internal variability and its binding—is actually required and its benefits outweigh the cost of increased complexity.

CVM thus allows us to define variable entities that do not have a public feature model but instead directly publish all their contained variability to the outside world. Then, this contained variability is accessible to be bound from outside that entity. To illustrate this, we use Listing 33 from page 43 again as a basis and change the definition of entity "WiperSystem" accordingly.

Listing 35: Entity part "rs" is now optional.

```
13 entity WiperSystem {
14     public part rs? of RainSensor;
15     public part ctrl of WiperController;
16     public part frontMotor of WiperMotor;
17     public part rearMotor of WiperMotor;
18 }
19
20 entity Car {
21     featureModel { HighEnd; }
22
23     part ws of WiperSystem;
24
25     configLink {
26         not HighEnd --> ws.ctrl#Simple[+];
27         HighEnd --> ws.ctrl#Advanced[+];
28
29         HighEnd --> ws.rs[+];
30     }
31 }
```

Note how "WiperSystem" has changed: no public feature model and no configuration decision model is provided for it; instead, its parts are now all marked as public and are thus directly visible and configurable from the outside. The new higher-level entity "Car" illustrates how: it includes "WiperSystem" as part "ws" and configures the contents of this wiper system directly from within its configuration decision model (lines 26-29), without the need for an intermediate feature model and feature configuration in entity "WiperSystem". Such a direct configuration may span an arbitrary number of levels in the containment hierarchy, as long as the lower-level parts are visible.

From a methodological point of view, two aspects are particularly noteworthy here. First, by consistently applying this technique, we could realize a traditional, non-compositional—or flat—variability management in which all variability is bound on the top-level of the complete system. Second, we can seamlessly mix the two approaches of either directly publishing an entity's internal variability or hiding this internal variability and its binding behind a well-defined interface in the form of a public feature model. For example, in Listing 35 we could change two of the four entity parts, perhaps “frontMotor” and “rearMotor”, to be private and introduce a public feature model for “WiperSystem” that only configures these two private parts, while at the same time the other entity parts are directly published for immediate configuration from the outside.

Taking these two observations together, we can employ an incremental approach when introducing CVM and the concept of compositional variability management in large, ongoing projects. And we can limit the use of compositional variability to those areas in a system design where this actually proves beneficial, without introducing unnecessary complexity in areas where this is not required. Such flexibility proves to be an important prerequisite for the overall applicability and feasibility of the concepts of compositional variability and configuration hiding.

6.3 Combining Named and Anonymous FMs/CDMs

In practice, when dealing with larger trees of hierarchically contained entities, the concise syntax of anonymous feature models and anonymous configuration decision models, as introduced in Section 5.5, proves very helpful in keeping things as simple as possible. Often, however, there are a few areas in a complex system hierarchy where a more flexible arrangement is needed and where the full power and flexibility of the standard syntax is therefore required.

To deal with such situations, it is possible to mix these two styles of specification. Let us consider first the case where a lower-level entity has its variability defined only by an anonymous feature model and we have to target that explicitly with a named configuration decision model and its explicit application. Listing 36 shows how to realized this.

Listing 36: From named to anonymous syntax style.

```
1  entity WiperSystem {
2      featureModel  { RainSensor; }
3  }
4  entity BodySystem {
5
6      part ws of WiperSystem;
7
8      configLink CDM ( --> WiperSystem) {
9          true --> RainSensor[+];
10     }
11
12     apply CDM ( --> ws);
13 }
```

As can be seen in line 8, we can refer to an anonymous feature model by denoting the containing entity, “WiperSystem” in this case. And line 12 shows how to refer to the implicit configuration of an anonymous feature model: simply by denoting the entity part containing the implicit configuration. So this deals with a transition from the named syntax style to the anonymous style. What about the other way round? This involves using an anonymous feature model and its implicit configuration as a source(!) within a configuration decision model and application, respectively. Here is a small example:

Listing 37: From anonymous to named syntax style.

```
 1  entity WiperSystem {
 2      public featureModel WiperFM { RainSensor; }
 3      config cfg of WiperFM;
 4  }
 5  entity BodySystem {
 6      featureModel { Luxury; }
 7
 8      part ws of WiperSystem;
 9
10      configLink CDM ( BodySystem --> WiperSystem.WiperFM ) {
11          Luxury --> RainSensor[+];
12      }
13
14      apply CDM ( this --> ws.cfg );
15  }
```

This time, the higher-level entity “BodySystem” has an anonymous feature model which we have to use as source, but the lower-level entity “WiperSystem” has a named feature model “WiperFM” with an explicitly defined configuration “cfg”. Thus, we cannot use an anonymous configuration decision model in this case. The solution is to explicitly refer to the anonymous feature model by denoting the containing entity (`BodySystem` in line 10), exactly as above, and refer to the implicit configuration in “BodySystem” with the special keyword `this` (line 14).

6.4 Inheritance

While exploring cloned features above, we encountered some important limitations. These occur especially in cases where we already know that particular instances of our cloned feature will be used for a particular purpose. For example, within a feature model we might model the situation of having a mandatory front and an optional rear wiper as a cloned feature `Wiper[1..2]`. However, the fact that the mandatory instance represents the front wiper and the other the rear wiper could only be specified in the documentation. Similarly, differences between the two wipers, such as an optional rain sensor only available for the front wiper, cannot clearly be represented. In such a case, it’s feature inheritance to the rescue!

Listing 38 shows how to precisely specify such a situation using inheritance. We have two features “FrontWiper” and “RearWiper” both inheriting from a third feature “Wiper”, which defines all common characteristics. We say that “Wiper” is a *template feature* for

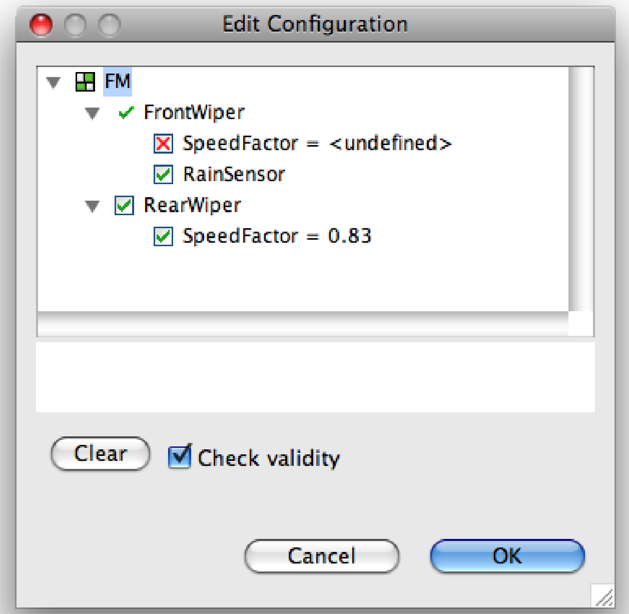
"FrontWiper" and "RearWiper". Since this is the sole purpose of "Wiper", we give it a cardinality of [0], which means that it will never appear in any configuration of feature model "FM" and in fact the configurator tool of CVM will completely hide such features, as shown in Figure 4. Such features with a cardinality of [0] are called *abstract features* in CVM.

Listing 38: Feature inheritance.

```
featureModel FM {
    Wiper[0] ( SpeedFactor:float );

    FrontWiper! extends Wiper ( RainSensor? );
    RearWiper?  extends Wiper;
}
```

As can be seen, VSL uses the common keyword `extends` to specify that one feature inherits from another. In the case of multiple inheritance, the keyword `and` is used to separate the template features. Note that we can now define the front wiper as mandatory and the rear wiper as optional, even though they inherit from the same template feature "Wiper". We can also add additional features to the front wiper that will not appear as children of the rear wiper, like "RainSensor" in the example.


Edit Configuration

FM
FrontWiper
SpeedFactor = <undefined>
RainSensor
RearWiper
SpeedFactor = 0.83

Figure 4: The feature model from Listing 38 shown in the CVM Configurator.

6.5 Value Propagation

In addition to feature selection and deselection it is also possible to copy values of parameterized features from source to target side. The following listing shows two typical situations.

Listing 39: Propagating values from source to target side.

```
 1 featureModel FMsrc {
 2     TempC! : float;     // temperature in Celcius
 3 
 4     ValueA? : integer;
 5     ValueB? : integer;
 6 }
 7 featureModel FMtrgt {
 8     TempF! : float;     // temperature in Fahrenheit
 9 
10     Sum! : integer;     // sum or -1
11 }
12 
13 configLink CL ( FMsrc --> FMtrgt ) {
14     true --> TempF = ( TempC * 1.8 ) + 32;
15 
16     ValueA & ValueB --> Sum = ValueA + ValueB;
17     not (ValueA & ValueB) --> Sum = -1;
18 }
```

Line 14 copies the temperature defined for the parameterized feature “TempC” on the source side to feature “TempF” on the target side and converts from Celcius to Fahrenheit. The inclusion criterion can be set to `true` here because this value propagation should be performed in all cases. Similarly, feature “Sum” will be set to the sum of “ValueA” and “ValueB” if both these features are selected (line 16), otherwise it will be set to -1 to indicate that one or both of the source-side value features is missing (line 17). Note here how a special value is used on the target side to indicate a state that is represented by selection and deselection on the source side, i.e. special value -1 for parameterized feature “Sum” corresponds to a certain selection state of the optional features “ValueA” and “ValueB”.

6.6 User Attributes

Being a simple, highly abstract modeling technique, feature models are ideally suited for assisting communication across departments, business units and companies. In such use cases, however, models often need to be augmented with project- or company-specific meta data. For this purpose, CVM provides a very simple but flexible mechanism called *user attributes*.

They are defined in two steps. First, most elements in CVM can be adorned with a *user attribute value*, which simply constitutes a key/value pair, both key and value being plain strings. Whenever models are exchanged with external actors, the key must be formulated in a globally unique manner as described above for feature link types (cf. Section 2.7). By convention, a Java-style naming scheme should be used such as in `org.`<*mydomain*>`.cvm.uaTypes.`<*mykey*>.

Second, most models in CVM, for instance feature models or configuration decision models, can be supplied with *user attribute types*. These types are valid for elements contained in the model and state that the value of a particular key must conform to a certain type. Here, exactly the same types can be used as are available for parameterized features (cf. Section 2.5). Take the following example:

Listing 40: Example of a global user attribute valid for features.

```
 1  // define a global user attribute type for key "myStatus":
 2  attr myStatus : string {"NEW", "OK", "REVIEW"} of feature;
 3
 4  featureModel FM {
 5
 6      Wiper (
 7          [1] ( Simple, Advanced ),
 8          RainSensor
 9      );
10
11      feature RainSensor {
12          myStatus = "NEW";
13          anotherAttrib = 22;      // user attribute value without type (legal)
14      }
15  }
```

Here, we define a user attribute type for key “myStatus”, which is valid for all features, as denoted by “`of feature`” in line 2. Lines 11 to 14 show how to assign a value to a feature’s “myStatus” attribute; within the curly braces, you can provide any number of user attribute assignments. If the key does not contain any special characters, the double quotes can be omitted.

It is important to understand that user attribute *types* are only an optional add-on. User attribute *values* can be defined without having a type defined for that particular key (line 13 in the example). And conversely, having defined a user attribute *type* for a particular key, you need not necessarily define a *value* for it (as for features “Simple” and “Advanced” in the example). Only if you have defined a value and a type for the same key must the value conform to the type declaration in the user attribute type. For example, replacing line 12 with

```
12          myStatus = "hello";      // ERROR: invalid value
```

would raise an error in the CVM editors, because only `"NEW"`, `"OK"` and `"REVIEW"` are legal values for key “myStatus”.

In summary, user attribute values can be though of as key/value pairs attached to the modeling elements in CVM. In principle, this can be done without any limitation. Care must only be taken to choose globally unique keys when exchanging models with the outside world. Then, as an optional add-on, user attribute types can be defined which state that for particular keys only values of a particular type may be specified (but it is still allowed to not specify a value at all for this particular key).

A Summary of Configuration Decision Definition

Configuration Activities

Summary of the different activities in configuring an optional feature.

	Syntax	Semantics within inclusion criterion of configuration decisions	 within feature configurations *or* effect of configuration decisions
Inclusion	`Feature[1]`	Is cardinality of feature set to [1] ?	Set cardinality of feature to [1] !
Exclusion	`Feature[0]`	Is cardinality of feature set to [0] ?	Set cardinality of feature to [0] !
Selection	`Feature[+]`	Is cardinality of feature **and all** its predecessors set to [1] ?	Set cardinality of feature **and all** its predecessors to [1] !
		(check if / make sure that feature will appear in final configuration)	
Deselection	`Feature[-]`	Is cardinality of feature **or any** of its predecessors set to [0] ?	Set cardinality of feature to [0] !
		(check if / make sure that feature won't appear in final configuration)	

Typical Contradictions

- including and excluding a single feature
- including a feature and excluding a predecessor
- selecting a feature and excluding a predecessor

Resolving Contradictions

An exclusion is prioritized over an inclusion.

A selection does not override the exclusion of a predecessor.

B Glossary

Abstract Feature A feature with a cardinality of [0].

Cloned Feature A feature having a cardinality with an upper bound greater 1, such as [0..2] or [1..$*$].

Configuration Consideration Some factor that drives the configuration of a target feature model and therefore leads to the addition of a particular configuration decision. For example: "all our competitors on the North American market have adaptive cruise control, so we should have that too.".

Contradiction Occurs when two or more configuration decisions with intersecting inclusion criteria state target configurations which cannot be put into effect at the same time or when a configuration decision model states an invalid target configuration.

Deselect Deselecting an optional feature in a feature configuration means setting its cardinality to [0]. Same as "exclude".

Exclude Excluding an optional feature in a feature configuration means setting its cardinality to [0]. Same as "deselect".

Include Including an optional feature in a feature configuration means setting (only) its cardinality to [1]. Compare this to "select".

Inconsistency A form of contradiction which constitutes an error in the configuration specification.

Mandatory Feature A feature with a cardinality of [1].

Optional Feature A feature with a cardinality of [0..1].

Select Selecting an optional feature in a feature configuration means setting its cardinality *and* the cardinality of all its predecessor features to [1]. Compare this to "include".

Variability Specification Language (VSL) A textual specification language which was introduced as part of the CVM framework with a syntax inspired by programming languages such as C or Java. Supports all concepts and modeling entities of CVM and thus provides a textual alternative to the XMI format for capturing CVM variability models.

References

[1] CVM-Framework Project Web-Site, 2009. `www.cvm-framework.org`.

[2] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practices*, 10(1):7–29, 2005.

[3] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*, pages 41–51, 2006.

[4] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) – feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), Carnegie Mellon University, 1990.

[5] Mark-Oliver Reiser, Ramin Tavakoli, and Matthias Weber. Unified feature modeling as a basis for managing complex system families. In *Proceedings of the 1st International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS), Lero Technical Report 2007-01*, pages 79–86. University of Limerick, Ireland, January 2007.

[6] Mark-Oliver Reiser, Ramin Tavakoli, and Matthias Weber. Compositional variability. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS-42)*. IEEE Computer Society Press, 2009.

[7] Mark-Oliver Reiser and Matthias Weber. Product lines in automotive electronics. In Nicolas Navet and Francoise Simonot-Lion, editors, *Automotive Embedded Systems Handbook*, chapter 7. CRC Press Inc., December 2008.

[8] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.