Domain-Specific Language Design Requires Feature Descriptions

Arie van Deursen Paul Klint

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

http://www.cwi.nl/$\sim${arie,paulk}/

ABSTRACT

A domain-specific language (DSL) provides a notation tailored towards an application domain and is based on the relevant concepts and features of that domain. As such, a DSL is a means to describe and generate members of a family of programs in the domain.

A prerequisite for the design of a DSL is a detailed analysis and structuring of the application domain. Graphical feature diagrams have been proposed to organize the dependencies between such features, and to indicate which ones are common to all family members and which ones vary.

In this paper, we study feature diagrams in more details, as well as their relationship to domain-specific languages. We propose the Feature Description Language (FDL), a textual language to describe features. We explore automated manipulation of feature descriptions such as normalization, expansion to disjunctive normal form, variability computation and constraint satisfaction. Feature descriptions can be directly mapped to UML diagrams which in their turn can be used for Java code generation. The value of FDL is assessed via a case study in the use and expressiveness of feature descriptions for the area of documentation generators.

1998 ACM Computing Classification System: D.2.2, D.2.9, D.2.11, D.2.13.

Keywords and Phrases: Domain engineering, tool support, software product lines, UML, constraints.

Note: To appear in the *Journal of Computing and Information Technology*, 2001.

Note: Work carried out under CWI project SEN 1.2, Domain-Specific Languages, sponsored by the Telematica Instituut.

1 Introduction

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [8]. As such, a DSL can be used to generate members of a family of systems in an application domain. The well-designed DSL is based on a thorough understanding of the underlying application domain, giving exactly the expressive power to generate required family members easily. Potential advantages of DSLs include reduced time to market, reduced maintenance costs, and higher portability, reliability, optimizability, and testability [8, 7]

A prerequisite for the design of a DSL is a detailed analysis and structuring of the application domain. Guidelines for acquiring such an understanding are provided by the research area of *domain analysis* which investigates ways of modeling domains. Following [22], a *domain analyst* is a person who examines the needs and requirements of a collection of systems which seem "similar". Neighbors emphasizes that this is work that only can be done by a person who has built many systems for different customers in the same problem area. The domain analyst is like a systems analyst, except that the goal is to support the development of families of related systems, not just one-of-a-kind productions [26].

Domain analysis originates from software reuse research, and can be used when constructing domain-specific reusable libraries, frameworks, languages, or product lines. Several domain analysis methodologies exists, of which ODM (Organization Domain Modeling [23]), FODA (Feature-Oriented Domain Analysis [19]), and DSSA (Domain-Specific Software Architectures [26]) are best known.

The most important result of domain analysis[1] is a *feature model* [5, Chapter 4]. A feature model covers the commonalities and variabilities of software family members, as well as the dependencies between the variable features. The feature model documents feature rationales, stakeholders, constraints (for example features may exclude each other), binding sites, and priorities.

A key element of the feature model is the *feature diagram*, which is a graphical notation for describing dependencies between (variable) features (see Figure 1 for an example). These feature diagrams are the topic of our paper. Feature diagrams originate from the FODA method [19]. They concisely describe all possible configurations (called *instances*) of a software system, focusing on the features that may differ in each of the configurations. Czarnecki and Eisenacker's book on *generative programming* includes a recent account of feature diagrams; Van Gurp *et al* discuss the role of feature diagrams in software product lines [16].

The purpose of this paper is to get a better understanding of feature diagrams, and their potential for supporting DSL design. In particular, we address the following concerns (see also Section 6 on related work):

- The notation of feature diagram is only superficially described, and is mostly explained by way of examples. We formalize the notion of feature diagram, by providing a DSL for feature definitions called FDL, together with a suite of formally defined operations for manipulating FDL expressions;

- Tool support for feature diagrams is still in its infancy. We construct prototype FDL tools, and discuss various ideas for further advancing them.

- Feature diagrams are hardly used in practice, and it is difficult to find actual examples of feature diagrams used in concrete projects. We attempt to address this by providing an additional case study in the use of FDL.

- It is unclear how to proceed once a feature diagram exists. To address this, we discuss what can be done with an FDL description, and how it can be mapped to a UML class diagram to get a first version of the configuration interface.

The plan for the paper is as follows. In Section 2 we introduce the graphical notation for feature diagrams and also present our textual Feature Description Language (FDL) that is able to express everything that can

[1]Issues related to domain engineering and analysis are also discussed on the *Program Transformation Wiki* at `www.program-transformation.org/`.

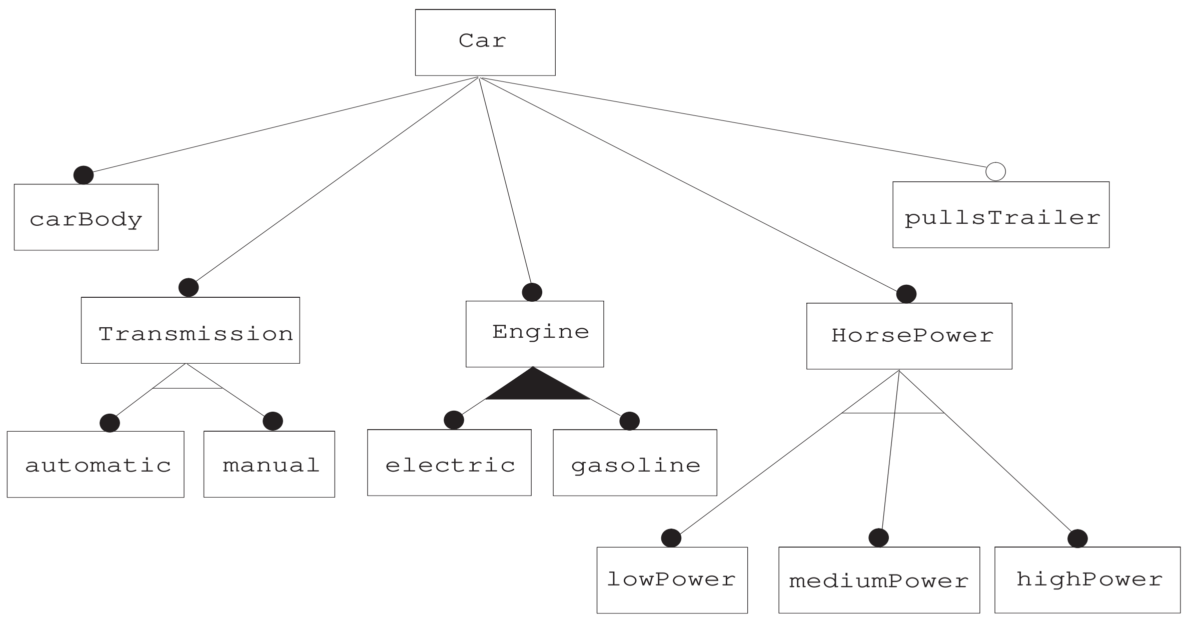

Figure 1: Feature diagram for a simple car

be expressed in a graphical feature diagram (and more). Next, we introduce in Section 3 a *feature diagram algebra* that allows the normalization of feature diagrams. In addition, we introduce a notion of *satisfaction* that enables us to answer the following question: *given a feature diagram and a list of user requirements, does this feature diagram contain software configurations that satisfy the user requirements?* In this way, a feature diagram can be actively queried and the initial investments in its construction start to pay off. Implementation issues are addressed in Section 4. We show how feature diagrams can be mapped to UML and Java classes. In Section 5 we perform a case study and describe how the variability of an existing commercial product for documentation generation can be modeled using FDL. Conclusions in Section 6 complete the paper.

2 Feature Diagrams

2.1 Graphical Notation for Feature Diagrams

Figure 1 shows a feature diagram for a simple car inspired by [19, 5]. The diagram states that a car consists of a `carBody`, `Transmission`, `Engine` and `HorsePower`. These four features are mandatory as indicated by the closed dot on top of each feature. The last feature of the car is `pullsTrailer`. It is optional as indicated by the open dot. `carBody` and `pullsTrailer` are *atomic features* which cannot be further subdivided in other features. In the sequel, we will call features that are defined in terms of other features *composite features*. We will use the convention that names of atomic features start with a lower case letter and names of composite features start with an upper case letter. Note that atomic and composite features are called features, respectively, subconcepts in [5].

The `Transmission` may be either `automatic` or `manual`. The open triangle joining the lines from `Transmission` to its sub-features indicates an exclusive ("one-of") choice: either `automatic` or `manual` may be selected but not both.

The `Engine` may either be `electric` or run on `gasoline` or both. The closed triangle joining the lines from `Engine` to `electric` and `gasoline` indicates non-exclusive "more-of") choice: either `electric` or `gasoline` or both may be selected.

```
Car: all( carBody, Transmission, Engine, HorsePower, pullsTrailer? )

Transmission: one-of( automatic, manual )

Engine: more-of( electric, gasoline )

HorsePower: one-of(lowPower, mediumPower, highPower)
```

Figure 2: Feature diagram for simple car in FDL

The `HorsePower` may either be `lowPower`, `mediumPower` or `highPower`. The open triangle joining the lines to the sub-features of `HorsePower` indicate an exclusive, one-of, choice.

An *instance* of a feature diagram consists of an actual choice of atomic features matching the requirements imposed by the diagram. An instance corresponds to a *product configuration* of a system family. A simple case analysis learns that the number of possible car instances is 36: 1 (`carBody`) $\times$ 2 (`Transmission`) $\times$ 3 (`Engine`) $\times$ 3 (`HorsePower`) $\times$ 2 (`pullsTrailer`) = 36.

2.2 Textual Notation for Feature Diagrams

Feature Diagrams yield nice pictures that describe a system's features. To enable email communication and discussion as well as creation of automatic tools for processing Feature Diagrams, a textual representation is, however, preferable. This textual representation should not only contain all information contained in the graphical diagram but it should also be suited for automatic processing.

One proposal, primarily intended for email communication of Feature Diagrams, is given in [2]. We show in Figure 2 how we will represent the graphical notation from Figure 1 in a textual form. They may also contain *constraints* (not shown in this example), but we postpone their description until Section 3.4.

An FDL definition consists of a number of *feature definitions*: a feature name followed by "`:`" and a *feature expression*. A feature expression can consist of

- an atomic feature,
- a composite feature: a named feature whose definition appears elsewhere,
- an optional feature: a feature expression followed by "`?`",
- mandatory features: a list of feature expressions enclosed in `all( )`,
- alternative features: a list of feature expressions enclosed in `one-of( )`,
- non-exclusive selection of features:[2] a list of feature expressions enclosed in `more-of( )`,
- a default feature value: `default =` followed by an atomic feature,
- and remaining features of the form `...`, indicating that a given set is not completely specified.

This structure is formally described in a complete grammar of FDL given in Figure 3. It is written in SDF [17, 27]. The layout conventions (white space, comments) of FDL are defined in module `Layout` that is not shown here.

[2] Called "or-features" in [5].

```
module Fdl
  imports Layout

  exports
    sorts FeatureName AtomicFeature FeatureDefinition
          FeatureDiagram FeatureExpression FeatureList
          Constraint DiagramConstraint UserConstraint
    lexical syntax
      [A-Z][a-zA-Z0-9]*                     -> FeatureName
      [a-z][a-zA-Z0-9]*                     -> AtomicFeature

    context-free syntax

      FeatureDefinition*  Constraint*       -> FeatureDiagram
      FeatureName ":" FeatureExpression     -> FeatureDefinition

      { FeatureExpression "," }+            -> FeatureList
      all(FeatureList)                      -> FeatureExpression
      one-of(FeatureList)                   -> FeatureExpression
      more-of(FeatureList)                  -> FeatureExpression

      FeatureName                           -> FeatureExpression
      AtomicFeature                         -> FeatureExpression
      FeatureExpression "?"                 -> FeatureExpression
      "default" "=" AtomicFeature           -> FeatureExpression

      "..."                                 -> AtomicFeature

      DiagramConstraint                     -> Constraint
      UserConstraint                        -> Constraint

      AtomicFeature "requires" AtomicFeature -> DiagramConstraint
      AtomicFeature "excludes" AtomicFeature -> DiagramConstraint

      "include" AtomicFeature               -> UserConstraint
      "exclude" AtomicFeature               -> UserConstraint
```

Figure 3: Grammar for Feature Definition Language

Variable	Type
F	FeatureExpression
Fs	{ FeatureExpression "," }*
Ft	{ FeatureExpression "," }+
A	AtomicFeature
C	Constraint
Cs	Constraint*

Figure 4: Variables used in FDL rules.

3 Feature Diagram Algebra

Given the textual representation of feature diagrams, we can now start developing rules to operate on feature diagrams. The result is a *feature diagram algebra*.

Overview of feature algebra rules The feature diagram algebra consists of four sets of rules:

- Normalization rules (Section 3.1): the purpose is to slightly simplify the feature expression by eliminating duplicate features and degenerate cases of the various constructors.
- Variability rules (Section 3.2): serve to count the number of possibilities for a given feature diagram.
- Expansion rules (Section 3.3): expand a normalized feature expression into a *disjunctive normal form*.
- Satisfaction rules (Section 3.4): given a feature expression in disjunctive normal form and given constraints, we determine which of the disjuncts satisfy the constraints. These satisfaction rules are the rules we are really interested in: they allow the formulation of constraints that are inherent in a feature diagram (system constraints) as well as constraints imposed by the user (user constraints). The satisfaction rules enable us to *query a feature diagram* for solutions that satisfy both system and user constraints.

Relation between the FDL definition and Feature Expressions Recall from Figure 3 that an FDL definition consists of a number of feature definitions that define a composite feature by associating a feature expression with a feature name. In that feature expression names of other composite features may occur.

From the perspective of the feature diagram algebra it is more convenient to manipulate a single feature expression. We assume therefore that one composite feature (by default the first one defined in the FDL description) is the *feature of interest* and all rules operate on the feature expression corresponding to this feature of interest. To further simplify the presentation we also assume that all names of composite features in the feature of interest have been replaced (recursively) by their definition as given in the FDL definition. This is a simple variable substitution process that we do not further explain.

Variable conventions In the presentation of the rules, we will use the conventions for variables shown in Figure 4 (each may be followed by digits or apostrophes). Observe that `Fs` represents comma-separated lists of *zero or more* feature expressions and that `Ft` represents comma-separated lists of *one or more* feature expressions.

The rules that we will present have been prototyped using the ASF+SDF Meta-Environment [3, 6, 20]. Detailed knowledge of the ASF+SDF specification formalism is, however, not necessary for a good understanding of the following sections.

```
equations

 [N1]  Fs, F, Fs', F?, Fs''           = Fs, F, Fs', Fs''
 [N2]  Fs, F, Fs', F, Fs''            = Fs, F, Fs', Fs''
 [N3]  F??                            = F?
 [N4]  all(F)                         = F
 [N5]  all(Fs, all(Ft), Fs')          = all(Fs, Ft, Fs')
 [N6]  one-of( F )                    = F
 [N7]  one-of(Fs, one-of(Ft), Fs')    = one-of(Fs, Ft, Fs')
 [N8]  one-of(Fs, F?, Fs')            = one-of(Fs, F, Fs')?
 [N9]  more-of( F )                   = F
[N10]  more-of(Fs, more-of(Ft), Fs')  = more-of(Fs, Ft, Fs')
[N11]  more-of(Fs, F?, Fs')           = more-of(Fs, F, Fs')?
[N12]  default = A                    = A
```

Figure 5: Normalization rules

```
all(carBody,
    one-of(automatic, manual),
    more-of(electric, gasoline),
    one-of(lowPower, mediumPower, highPower),
    pullsTrailer?)
```

Figure 6: Normalized feature expression for Car

3.1 Normalization Rules

The normalization rules N1–N12 are shown in Figure 5. An informal explanation of these rules is as follows:

N1 combines mandatory and optional features in a list.

N2 removes duplicates in a list.

N3 joins duplicate optionals.

N4-N5 normalize special cases of `all`. Nested `all`s are flattened.

N6-N7 normalize special cases of `one-of`. Nested `one-of`s are flattened.

N8 transforms a `one-of` containing one optional feature into an optional `one-of`.

N9-N10 normalize special cases of `more-of`. Nested `more-of`s are flattened.

N11 transforms a `more-of` containing one optional feature into an optional `more-of`.

N12 eliminates the `default =` annotation.

The normalized feature expression for Car is shown in Figure 6.

3.2 Variability Rules

An important purpose of feature diagrams is to describe the variability of a software system and it is therefore interesting to count the possibilities. Given a normalized feature diagram, the variability rules V1–V8 shown in Figure 7 define the variability for each construct.

```
equations

[V1]  var(A)                 = 1
[V2]  var(F?)                = var(F) + 1
[V3]  var(all(F, Ft))        = var(F) * var(all(Ft))
[V4]  var(all(F))            = var(F)
[V5]  var(one-of(F, Ft))     = var(F) + var(one-of(Ft))
[V6]  var(one-of(F))         = var(F)
[V7]  var(more-of(F, Ft))    = var(F) + (var(F)+1)* var(more-of(Ft))
[V8]  var(more-of(F))        = var(F)
```

Figure 7: Rules for computing variability

The variability of an atomic feature is one [V1] and the variability of an option adds one to the variability of its argument [V2]. The variability of `all` is the product of the variabilities of its arguments [V3,V4]. The variability of `one-of` is the sum of the variabilities of its arguments [V5,V6]. The variability of `more-of`$(F_1,\ldots,F_n)$ is slightly more complex and amounts to computing 2^n-1 for the case that $\mathtt{var}(F_i)=1$ for $i=1,\ldots,n$, corresponding to switching each feature on or off, but disallowing the empty configuration.

Assuming that $N_1 = \mathtt{var(F)}$ and $N_2 = \mathtt{var(more\text{-}of(Ft))}$, the variability of `more-of(F, Ft)` equals $N_1+N_1*N_2+N_2$, representing the cases that only `F` is used, that the combination of `F` and `Ft` is used, or that only `Ft` is used. In [V7] this written in the format $N_1+(N_1+1)*N_2$ (which avoids recalculation of N_2 when executing these laws as rewrite rules).

In [12] the variability for the more-of case is formulated in the following (equivalent) manner:

$$\mathtt{var}(\mathtt{more\text{-}of}(F_1,\ldots,F_n)) = (\mathtt{var}(F_1)+1)*(\mathtt{var}(F_2)+1)*\cdots*(\mathtt{var}(F_n)+1)-1$$

The variability for the feature expression for Car (Figure 6) is 36. The variability clearly grows exponentially as can be appreciated by calculating the variability of the feature expression for the documentation generator that we will discuss later on (Figure 14, Section 5). In that case, the variability is 3771425280!

3.3 Expansion Rules

The next step is to expand a normalized feature expression into a *disjunctive normal form* defined as follows:

$$\mathtt{one\text{-}of(\ all(}A_{11},\ldots,A_{1n_1}\mathtt{),\ldots,\ all(}A_{m1},\ldots,A_{mn_m}\mathtt{))}$$

The outermost operator of a disjunctive normal form is thus `one-of`. and its arguments are all `all`s with only atomic features as arguments. The resulting representation is essentially a list of all possible configurations.

The expansion rules E1–E4 are shown in Figure 8 and amount to eliminating optionals, `one-of`s, and `more-of`s that occur nested within an `all`:

E1,E2 translates an `all` containing an optional feature expression in two cases: one with and one without the feature.

E3 translates an `all` containing a `one-of` in two cases: one with the first alternative and one with the `one-of` with the first alternative removed.

E4 translates an `all` containing a `more-of` into three cases: one with the first alternative, one with the first alternative and the remaining `more-of`, and one with only the remaining `more-of`.

The expansion of the feature expression for Car (Figure 6) is shown in Figure 9. As expected from the variability computed in the previous section, it contains 36 alternatives.

```
equations

[E1]  all(Fs, F?, Ft)
           = one-of(all(Fs, F, Ft), all(Fs, Ft))

[E2]  all(Ft, F?, Fs)
           = one-of(all(Ft, F, F), all(Ft, Fs))

[E3]  all(Fs, one-of(F, Ft), Fs')
           = one-of(all(Fs, F, Fs'), all(Fs, one-of(Ft), Fs'))

[E4]  all(Fs, more-of(F, Ft), Fs')
           = one-of(all(Fs, F, Fs'),
                  all(Fs, F, more-of(Ft),Fs'),
                  all(Fs, more-of(Ft), Fs')
                 )
```

Figure 8: Expansion rules

3.4 Satisfaction Rules

Now we are in a good position to explain *constraints* in feature diagrams. As defined in Figure 3, a constraint can have one of the following forms:

- `A1 requires A2`: if feature `A1` is present, then feature `A2` should be present as well.
- `A1 excludes A2`: if feature `A1` is present, then feature `A2` should not be present.
- `include A`: feature `A` should be present.
- `exclude A`: feature `A` should not be present.

The first two kinds of constraints are called *diagram constraints* since they express fixed, inherent, dependencies between features in a diagram.

The last two kinds of constraints are called *user constraints* since they express the user requirements regarding presence or absence of a feature. The user constraints may vary between subsequent uses of the feature diagram.

The purpose of constraints is to further limit the variability of a feature diagram. This can be achieved by introducing a notion of *satisfaction* that determines for each disjunct of a feature expression in disjunctive normal form whether it satisfies given constraints.

The satisfaction rules S1–S8 are shown in Figure 10. Typically, they check for a given disjunct whether there is an applicable constraint and, if so, whether that constraint is satisfied or not. The binary constraints `excludes` and `requires` are handled in [S1,S2], respectively, [S3,S4]. Typically, if the disjunctive normal form `all(Fs, A1, Fs')` and one of the constraints have an `A1` in common, the appropriate check is performed whether a corresponding `A2` in the constraint is absent or present. In as similar fashion, [S5,S6] and [S7,S8] handle the unary constraints `includes`, respectively, `excludes`.

If we introduce the following two constraints in the car example:

- `pullsTrailer requires highPower` (not unreasonable if you don't want to ruin your engine), and
- `include pullsTrailer` (a user requirement).

and reduce the disjunctive normal form for Car we get the result shown in Figure 11. Observe that the original 36 possibilities have been reduced to just 6.

```
one-of(all(carBody, automatic, electric, lowPower, pullsTrailer),
       all(carBody, automatic, electric, gasoline,lowPower,
           pullsTrailer),
       all(carBody, automatic, gasoline, lowPower, pullsTrailer),
       all(carBody, automatic, electric, mediumPower,
           pullsTrailer),
       all(carBody, automatic, electric, gasoline,mediumPower,
           pullsTrailer),
       all(carBody, automatic, gasoline, mediumPower, pullsTrailer),
       all(carBody, automatic, electric, highPower, pullsTrailer),
       all(carBody, automatic, electric, gasoline,highPower,
           pullsTrailer),
       all(carBody, automatic, gasoline, highPower, pullsTrailer),
       all(carBody, manual, electric, lowPower, pullsTrailer),
       all(carBody, manual, electric, gasoline,lowPower,
           pullsTrailer),
       all(carBody, manual, gasoline, lowPower, pullsTrailer),
       all(carBody, manual, electric, mediumPower, pullsTrailer),
       all(carBody, manual, electric, gasoline,mediumPower,
           pullsTrailer),
       all(carBody, manual, gasoline, mediumPower, pullsTrailer),
       all(carBody, manual, electric, highPower, pullsTrailer),
       all(carBody, manual, electric, gasoline,highPower,
           pullsTrailer),
       all(carBody, manual, gasoline, highPower, pullsTrailer),
       all(carBody, automatic, electric, lowPower),
       all(carBody, automatic, electric, gasoline,lowPower),
       all(carBody, automatic, gasoline, lowPower),
       all(carBody, automatic, electric, mediumPower),
       all(carBody, automatic, electric, gasoline,mediumPower),
       all(carBody, automatic, gasoline, mediumPower),
       all(carBody, automatic, electric, highPower),
       all(carBody, automatic, electric, gasoline,highPower),
       all(carBody, automatic, gasoline, highPower),
       all(carBody, manual, electric, lowPower),
       all(carBody, manual, electric, gasoline,lowPower),
       all(carBody, manual, gasoline, lowPower),
       all(carBody, manual, electric, mediumPower),
       all(carBody, manual, electric, gasoline,mediumPower),
       all(carBody, manual, gasoline, mediumPower),
       all(carBody, manual, electric, highPower),
       all(carBody, manual, electric, gasoline,highPower),
       all(carBody, manual, gasoline, highPower))
```

Figure 9: Disjunctive normal form for Car (36 disjuncts)

```
equations

[S1]  is-element(A2, Fs) | is-element(A2, Fs') = true
       ==================================================
      sat(all(Fs, A1, Fs'), Cs A1 excludes A2 Cs') = false

[S2]  is-element(A2, Fs) | is-element(A2, Fs') = false
      ==================================================
      sat(all(Fs, A1, Fs'), Cs A1 excludes A2 Cs') =
      sat(all(Fs, A1, Fs'), Cs Cs')

[S3]  is-element(A2, Fs) | is-element(A2, Fs') = false
      ==================================================
      sat(all(Fs, A1, Fs'), Cs A1 requires A2 Cs') = false

[S4]  is-element(A2, Fs) | is-element(A2, Fs') = true
      ==================================================
      sat(all(Fs, A1, Fs'), Cs A1 requires A2 Cs') =
      sat(all(Fs, A1, Fs'), Cs Cs')

[S5]  is-element(A,Ft) = true
      ==================================================
      sat(all(Ft), Cs include A Cs') = sat(all(Ft), Cs Cs')

[S6]  is-element(A,Ft) = false
      ==================================================
      sat(all(Ft), Cs include A Cs') = false

[S7]  is-element(A,Ft) = true
      ==================================================
      sat(all(Ft), Cs exclude A Cs') = false

[S8]  is-element(A,Ft) = false
      ==================================================
      sat(all(Ft), Cs exclude A Cs') = sat(all(Ft), Cs Cs')

[default-S9]
      sat(all(Ft), Cs) = true
```

Figure 10: Satisfaction rules

```
one-of(
   all(carBody, automatic, electric, highPower, pullsTrailer),
   all(carBody, automatic, electric, gasoline,highPower, pullsTrailer),
   all(carBody, automatic, gasoline, highPower, pullsTrailer),
   all(carBody, manual, electric, highPower, pullsTrailer),
   all(carBody, manual, electric, gasoline,highPower, pullsTrailer),
   all(carBody, manual, gasoline, highPower, pullsTrailer)
)
```

Figure 11: Reduced feature expression for Car

```
car.transmission=automatic
car.pullsTrailer=false
car.engine=electric,gasoline
```

Figure 12: Car instance specified as Java property file

4 Implementing Feature Diagrams

A feature diagram describes possible system configurations. To actually arrive at a working system, these configurations must be implemented. In this section we analyze how feature diagrams can be implemented using object-oriented models. We focus on the use of UML and Java as implementation targets. Observe that the resulting UML only describes the *configuration interface* of a family of systems such as a product line. The actual implementation of the underlying framework will involve many more classes, mostly dealing with the features *common* to all software products, as opposed to those that are *variable* between them.

A first question is how to represent actual configurations, that is, feature diagram instances. Recall that a configuration is just a set of features selected from the diagram. This suggests that a simple *property list*, as for example available through Java property files, suffices to indicate which features are switched on or off. As an example, an instance of a car having automatic transmission, no trailer, and both an electric and a gasoline engine is given as a Java property file in Figure 12.

The names of the properties are derived from the diagram, and constitute a path from the root to the selected feature. In practice, many of the published feature diagrams are in fact very *flat*, so these property names will be sufficiently simple. For example, the feature diagram for describing ways in which different window managers move windows covered by [19, p.64] is essentially a flat list of elementary features such as *overlappedLayout*, *moveIcon*, etc. Likewise, none of the feature diagrams presented in [5] have a depth larger than 3. Even if the depth would be larger, we can always normalize feature diagrams, as we have seen in the previous section, to a disjunctive normal form, in which the depth is at most 2.

Methods implementing the features need to be aware of the configuration chosen. In the simplest approach, such methods perform an explicit check on the property values, and adapt their behavior accordingly. The disadvantage of this is that it amounts to including if-then-else or case statements at various places, which is generally considered bad object-oriented programming style [13, 21].

A more involved approach is to turn features into classes, and if possible to use inheritance in order to specialize methods to particular feature instances. This amounts to deriving a UML class diagram from a feature diagram. In Figure 13 we have done this in a systematic way for the car example of Figure 2. From this diagram, we can make the following observations:

- Every feature corresponds to a class.
- Associations between classes are tagged with a ⟨⟨*stereotype*⟩⟩ indicating the sort of feature dependency they originate from.
- The mandatory dependency between Car and CarBody is mapped to an aggregation between these classes.
- The optional dependency between Car and PullsTrailer corresponds to an association with a cardinality of 0 or 1.
- The `one-of` and `more-of` lists for Engine, Transmission, and Horsepower results in abstract classes *Engine* and *Transmission*, with specific subclasses for each of the alternatives.
- The `one-of` dependency for Transmission and HorsePower results in a one-to-one association with Car; The `more-of` dependency between Engine and Car results in a one-to-many association, with multiplicity equal to the cardinality of the number of or-features (in this case “1..3”).

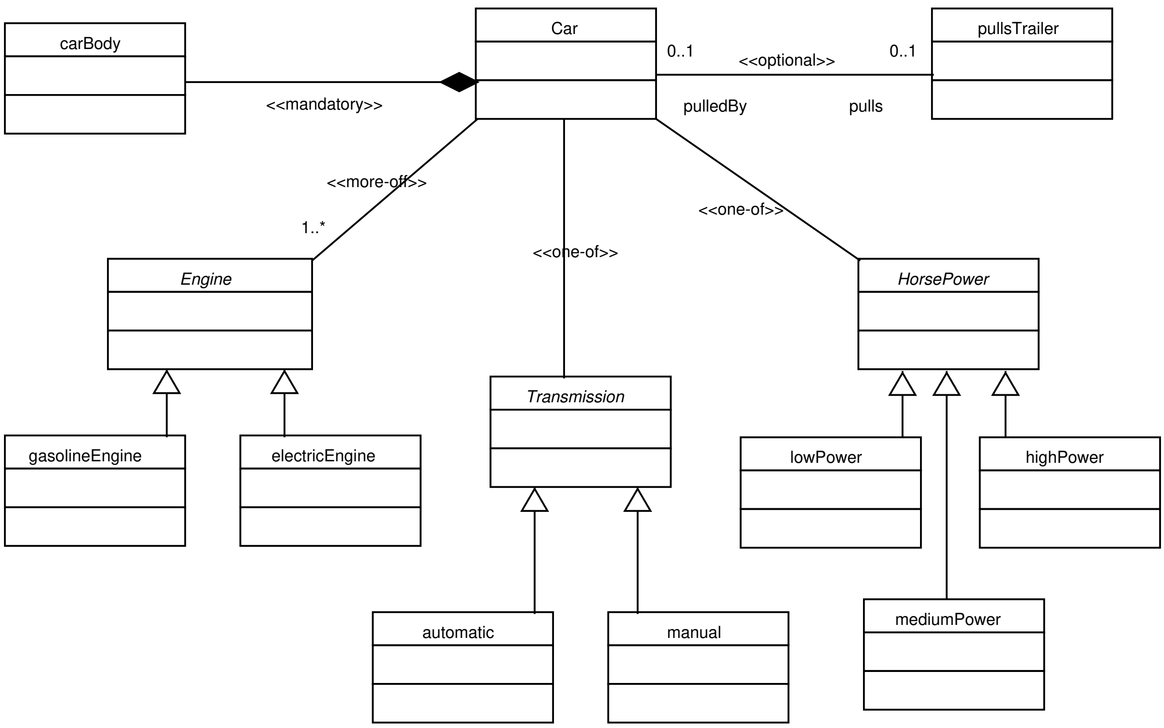

Figure 13: UML Class Diagram for Car Features

Since this example includes all FDL features, this approach corresponds to a systematic translation of FDL to UML class diagrams. The mapping is based on the discussion Czarnecki and Eisenacker provide in [5, Section 4.5]. They discuss several C++ implementations, involving static and dynamic parameterizations, mixins, parameterized inheritance, and multiple inheritance. They do not, however, provide the intuitive diagram of Figure 13. Compared to the options they discuss, our approach is simple, systematic, and independent of the availability of a parameterization mechanism.

The result of translating FDL to UML can be represented using XMI, the XML Meta data Information exchange format.[3] XMI documents can be imported into UML modeling tools such as Rational, TogetherJ, and Argo/UML, which in turn can use the diagrams to generate, for example, Java classes.

In addition to that, an FDL specification can be used to generate a "configuration editor". Such an editor is a user interface panel in which a product builder can select which features to include. The result of an editing session is the input to the generated configuration classes. It consists of a concrete property file, or executable code for creating the appropriate configuration objects.

A logical next step is to create a domain-specific language based on the FDL definition. This is particularly useful if a natural language for expressing feature instantiations exist, or if product instances should be read, manipulated, and put under revision control. Observe that the operators in an FDL definition are very close to the operators in, for example, BNF or (as we have used) SDF: `one-of` corresponds to alternative productions, `?` to optional productions, and `more-of` to a list construct. Thus, an FDL definition can easily become the basis for a grammar of a language for building systems for the underlying application domain.

5 DocGen Case Study

5.1 Background

In this section, we explore the use of feature diagrams for the purpose of designing a configuration DSL for DocGen, a commercial documentation generator for software systems [9, 11]. It has been instantiated for various languages, including Cobol, SQL, JCL, as well as various proprietary languages.

DocGen operates by populating a repository with a series of facts derived from legacy sources. These facts are used to derive web-based documentation for the systems analyzed. This documentation includes textual summaries, overviews, various forms of control flow graphs, architectural information, and so on. Information is available at different levels of abstraction, which are connected through hyper links.

DocGen customers have different wishes regarding the languages to be analyzed, the specific set of analyses to be performed, and the way in which the collected information should be retrieved. Thus, DocGen is a *software product line*, providing a set of reusable assets well-suited to express and efficiently implement different customized documentation generation systems.

The construction of DocGen is characterized by evolutionary design (DocGen is being developed following the principles of extreme programming [1]). DocGen started as a research prototype described by [9]. This prototype was not implemented as a reusable framework; instead it just produced documentation as desired by one particular customer. As the commercial interest in applications of DocGen grew, more and more variation points were introduced, evolving DocGen into a system suitable for deriving many different documentation generation systems.

At present, DocGen is an object-oriented application framework written in Java and using a relational database as its repository. It provides a range of core classes for analysis and presentation purposes. In order to instantiate family members, a package specific to a given customer is created, containing specializations of core classes where needed, including methods called by the DocGen factory for producing the actual DocGen instantiation.

With the number of customer configurations growing, it is time to rethink the way in which DocGen product instantiations are created, and what sort of variability the DocGen product line should offer. In this section, we explore how FDL can help to organize the variable features of DocGen.

[3] See `http://xml.coverpages.org/xmi.html`.

```
DocGen:
  all(Analysis, Presentation)

Analysis:
  all( RelationSet, AnalysisSpecializations? )

RelationSet:
  more-of( annotationRelation, callRelation, entitiesRelation,
           entityOperationRelation, ... )

AnalysisSpecializations:
  more-of( callHandlers,
           columnPositions,
           codingConventions,
           fileNameConventions, ... )

Presentation:
  all( Localization,
       Interaction,
       MainBlocks,
       SourceSections,
       PresentationSpecializations? )

Localization:
  one-of( default = english, dutch )

Interaction:
  one-of( crawled, default = dynamic )

MainBlocks:
  all( UsersGuide,
       more-of( programBlock, copybookBlock, statisticsBlock, ... ) )

SourceSections:
  more-of( annotationSection, activationSection, entitiesSection,
           parametersSection, ... )

UsersGuide:
  more-of( indexpage, programHelp, copybookHelp, statisticsHelp, ...,
           annotationHelp, activationHelp, entitiesHelp,
           parametersHelp, ... )

PresentationSpecializations:
  more-of( frameSize, ... )
```

Figure 14: Configurable Features of DocGen

5.2 DocGen Features

A selection of the variable features of DocGen and their constraints are shown in Figures 14 and 15. The features listed describe the variation points in the current version of DocGen. One of the goals of constructing the FDL specification of these features is to search for alternative ways in which to organize the variable features, in order to optimize the configuration of DocGen family members.

The features listed focus on just the `Analysis` and `Presentation` configuration of `DocGen`, as specified by the first dependency of Figure 14.

The `Analysis` features show how the DocGen analysis can be influenced. First, the data-model used can be specified through the `RelationSet`, which indicates which relations of the DocGen data model are to be populated. The Java implementation follows the UML diagram suggested in Section 4, where a `more-of` results in a one-to-many association between a series of classes all inheriting from the abstract *Relation* class.

Second, an optional list of `AnalysisSpecializations` can be provided. Such specializations can be implemented in several ways. Some are just simple parameter settings, such as the `columnPositions`, which are encoded in Java property files. Others correspond to specialized methods for performing certain analyses — such features indicate which customer-specific classes need to be included.

The `Presentation` features affect the way in which the facts contained in the repository are presented to DocGen end users. One of the more obvious features is the need for `Localization`, which in this case amounts to choosing between English and Dutch. This is implemented through the web-browsers localization scheme, thus making it a feature that the end-user can determine at any point during a session.

The `Interaction` feature determines the moment the HTML pages are generated. In `dynamic` interaction, a page is created whenever the end-user requests a page. This has the advantage that the pages always use the most up-to-date information from the repository, and that interactive browsing is possible. In `crawled` mode, all pages are generated and stored on, for example, a CD-ROM. This has the advantage that no web-server is needed to inspect the data, and that they can be easily easily viewed on a disconnected laptop. As we will see, the `Interaction` feature puts constraints on other presentation features.

The `MainBlocks` feature indicates the contents of the root page of the derived documentation. It is a list of standard blocks that can be reused, implemented again as a many-to-one association to subclasses of the abstract *Block* class. If necessary for a particular customer, a specific subclass of one of the blocks can be created, and specified as one of the `PresentationSpecializations`. The `SourceSections` is a similar configuration of the contents of the main page used for documenting individual source files.

A mandatory block in the `MainBlocks` is the `UsersGuide`. The contents of the user's guide can vary, and depends on the features included. The feature description given indicates that it consists of a series of Help sections, which together constitute the user's guide (which is available as an integrated pdf file, as well as per section from relevant pages).

5.3 DocGen Feature Constraints

Figure 15 lists several constraints restricting the number of valid DocGen configurations of the features listed in Figure 14.

First, there are dependencies between the tables of the data-model. In other words, not every selection from `RelationSet` is valid. For example, if a table uses a foreign key, the table providing that key as primary key should be available as well. The figure shows this dependency for the `entitiesRelation` and the `entityOperationRelation`.

Second, the pages that can be presented depend on the analyses that are conducted. For example, in order to show the activation of modules, the call relation between modules must be extracted. Likewise, the contents of the user's guide depends on the pages that are presented to the user. Thus, if the documentation presented should include statistics on the McCabe index, fan-in, and fan-out, the user's guide should include a section explaining what the meaning of these metrics is.

Last but not least, certain features are in conflict with each other. In particular, the `annotationSection` can be used to let the end-user interactively add annotations to pages, which are then stored in the repository.

```
%%
%% Some of the dependencies between the tables in the repository
%%
entityOperationRelation requires entitiesRelation

%%
%% Some of the constraints between presentation blocks and sections,
%% and the RelationSet used for analysis.
%%
annotationSection requires annotationRelation
activationSection requires callRelation
entitiesSection   requires entitiesRelation
entitiesSection   requires entityOperationRelation

%%
%% Some of the constraints between presentation blocks and sections,
%% and the contents of the User's Guide.
%%
programBlock      requires programHelp
statisticsBlock   requires statisticsHelp
annotationSection requires annotationHelp

%%
%% Mutually exclusive features
%%
crawled excludes annotationRelation
crawled excludes annotationSection
```

Figure 15: Constraints on variable DocGen features.

This is only possible in the dynamic version, and cannot be done if the `Interaction` is set to `crawled`.

6 Concluding Remarks

6.1 Related Work

In this paper, we have tried to get a better understanding of the nature of feature diagrams as originating from the FODA domain analysis methodology.

We do this based on a textual representation of feature diagrams. A similar representation is provided by [2], primarily inspired by the need to exchange feature diagrams over the Internet.

The notion of *normalized* feature diagrams is also discussed by Czarnecki and Eisenacker [5, Section 4.4.1.5]. They provide two rewrite rules in a visual representation. They focus on the elimination of optional features occurring within a `one-of` or `more-of` context, which corresponds to our rules `N8` and `N11` (Figure 3.1) Mapping feature diagrams to UML is also discussed by Czarnecki and Eisenacker [5, Section 4.5]. They present a number of advanced alternatives, where as our focus is more on a simple, but systematic approach to mapping feature diagrams to class diagrams.

Not many feature diagram case studies have been published, making it difficult to assess the true benefits of feature diagrams. We have collected every feature diagram we could find in the published literature (mostly from [5, 19]) and translated them into FDL. Our DocGen case study aims at helping to fill this gap. Moreover, the textual format of FDL makes it easier to exchange feature diagrams via, e.g., web sites.

The specifications we provide are directly executable in ASF+SDF, and thus can be the basis for tool support for feature diagrams. The original FODA method already contains some Prolog-based tools for checking feature diagrams [19]. We were, however, unable to discover their precise functionality.

Most work on Domain Analysis & Engineering focuses on the development and refinement of a *process* that will lead to a set of reusable assets (components and other work products) that can be used to construct a family of related applications. This process has to solve a knowledge acquisition and management problem: given knowledge sources (domain experts, documentation, source code, market surveys, pricing strategies, and the like) create a structured view of the domain. The main goals are to build up a domain vocabulary, identify features, identify variation points and ultimately construct feature diagrams that capture all this information. Tool support mostly provides a blackboard-like architecture to accomplish these tasks. Examples of systems are Sherlock Holmes [24, 25], DARE-COTS [14], and Feature RSEB [15]. A survey of this kind of systems is given in [25]. The emphasis of these tools is on the *domain engineering process*, and in this sense our work can be seen as complementary to the systems mentioned above.

6.2 Results

We have presented results in three areas. First, we have formalized the notion of feature diagram. The Feature Diagram Algebra presented in this paper has two benefits:

- It can be used as the basis for tool development. The rules presented here can be directly executed by the ASF+SDF Meta-Environment [3] yielding prototype tools we have experimented with.
- It can be used to mediate between the options provided by software applications as expressed in their feature diagram and the requirements of a user. Typically, a user indicates on a check list which features he wants and which he certainly does not want. Given the disjunctive normal form of the feature diagram, we can apply the satisfaction rules and obtain a reduced feature expression that contains zero or more satisfactory disjuncts. These disjuncts form the alternatives in an offering that can be made to the user. Each disjunct may be enriched with additional information such as total costs or planning constraints.

Second, we have shown how feature diagrams can be directly mapped to UML diagrams which in their turn can be used for Java code generation. Although we did not completely formalize this two stage process, we strongly believe that a large degree of automation can be achieved.

Third, we have presented a case study of the use of FDL by analyzing the variation points of the documentation generator DocGen.

The design of domain-specific languages requires a detailed analysis of the domain of interest. The operational view on feature descriptions as presented in this paper is a powerful tool to support such a domain analysis.

6.3 Future work

This paper is only a first step in the direction of using feature diagrams for designing domain-specific languages and for describing product families.

A major obstacle for using feature diagrams is the question how to create them at all. Usually, a flat list of atomic features is known about a software family or application area and it is not so easy to find concepts that can be used to introduce hierarchical structure in this list. One possible approach is to use *cluster analysis* or *concept analysis* to find these concepts, as for example also used to find objects in legacy procedural code [10].

In the current paper, we have used only very simple constraints. Experience shows that these are sufficient to describe realistic systems but it is conceivable that more expressive constraints may lead to more concise feature diagrams. Examples of extensions are:

- Add Booleans expressions, e.g., `include A1 or include A2`.
- Associate numeric values with atomic features, e.g., `HorsePower = 75`.
- Add relational operator, e.g., `HorsePower > 100`.

Another limitation of feature diagrams is that they do not contain information relevant for the binding time of features. For instance, *cross-cutting* features like error reporting and transaction logging have a major impact on the actual classes that implement a given feature diagram. It is of practical importance to explore how feature diagrams can be extended with all the information that is needed to fully automatically generate an implementation. Such information will also have to contain a mapping between feature names and existing code that implements the feature. In our DocGen case study it is important not include class files for packages of which it can be statically concluded that they are not needed. We are currently investigating whether usage of *software packages* and *package bundles* [18] can be used to influence code packaging for features with a static binding time.

In this paper we have given a very naive approach to computing the satisfaction of the constraints in a feature diagram. Since the size of a disjunctive normal form grows exponentially, it becomes very soon infeasible to compute it, let alone to check the constraints. We envisage that using well-known techniques from model checking such as *ordered binary-decision diagrams* [4] will make it possible to avoid computing the disjunctive normal form and check the constraints directly. Note that one has to strike a balance between efficient satisfaction techniques and further extension of the expressive power of constraints as outlined above.

To summarize, a lot of work is still needed to turn feature diagrams in a widely applicable technique. The longer term perspective is to have a “Feature Analysis and Manipulation Environment” to develop a product family and its feature diagram simultaneously. The feature diagram then really gets a dual purpose: it helps to structure the code base into independent features implemented by independent components or packages *and* it can be used by a customer to explore the possibilities of the product family and to make a selection that suits its needs at acceptable costs.

Acknowledgements

We gratefully acknowledge the comments on drafts of this paper made by Krzysztof Czarnecki, Leon Moonen, Joost Visser, the editors Marjan Mernik and Ralf Lämmel, and the anonymous referees.

References

[1] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.

[2] Feature Model Diagrams in text and HTML, 2001. `http://www.boost.org/more/feature_model_diagrams.htm`.

[3] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[4] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[6] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[7] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[8] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. On line at `www.program-transformation.org/`.

[9] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society, 1999.

[10] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[11] Automatic documentation generation; white paper. Software Improvement Group, 2001. `http://www.software-improvers.com/PDF/DocGenWhitePaper.pdf`.

[12] U. Eisenecker, M. Selbig, F. Blinn, and K. Czarnecki. Feature modeling of software system families (in german). *OBJECTspektrum*, pages 23–30, September/Octobre 2001. `www.objectspektrum.de`.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE-COTS: A domain analysis support tool. In *Proceedings 17th International Conference of the Chilean Computer Science Society*. IEEE Computer Society, 1997.

[15] M. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85. IEEE Computer Society, 1998.

[16] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA)*, pages 45–54. IEEE Computer Society, 2001.

[17] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[18] M. de Jonge. Source tree composition. Technical report, CWI, 2001.

[19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[20] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[22] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–74, September 1984.

[23] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization domain modelling (ODM) guidebook version 2.0. Technical Report STARS-VC-A025/001/00, Synquiry Technologies, Inc, 1996.

[24] G. Succi, A. Eberlein, J. Yip, K. Luc, M. Nguy, and Y. Tan. The design of Holmes: a tool for domain analysis and engineering. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM'99)*, 1999.

[25] G. Succi, J. Yip, and E. Liu. Analysis of the essential requirements for a domain analysis tool. In *ICSE Workshop on Software Product Lines Economics, Architectures, and Implications*, 2000.

[26] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–37, 1995.

[27] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.