Change-based FODA diagrams

Bridging the gap between feature-oriented design and implementation

Dr. Peter Ebraert[*]
Universiteit Antwerpen
Middelheimlaan 1
2020 Antwerpen, Belgium
peter@ebraert.be

Quinten David Soetens[†]
Universiteit Antwerpen
Middelheimlaan 1
2020 Antwerpen, Belgium
quinten.soetens@ua.ac.be

Prof. Dr. Dirk Janssens
Universiteit Antwerpen
Middelheimlaan 1
2020 Antwerpen, Belgium
dirk.janssens@ua.ac.be

ABSTRACT

Feature Oriented Design Analysis (FODA) diagrams present the design of feature-oriented software applications. In some cases, however, the actual implementation of such an application does not correspond to the design that was set forward by the FODA diagram. Such discrepancies are referred to as the gap between design and implementation.

We present a bottom-up approach for generating FODA diagrams from the changes to the source code. Unlike ordinary FODA diagrams, those diagrams are based on the implementation. Thanks to that, they do not only contain coarse-grained design information, but also incorporate fine-grained implementation details, which can be used to bridge between feature-oriented design and implementation.

Categories and Subject Descriptors

D2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

Separation of concerns, Feature-oriented programming, Documentation, Design, Implementation

1. INTRODUCTION

Software is usually developed by means of a development process that encompasses the following phases: extraction and specification of requirements, software design, implementation, verification and maintenance. One of the undesired side effects of this development process is the emergence of a gap between those phases. Researchers already reported extensively on problems that stem from that gap [1, 14, 16, 19]. In this paper, we atempt to bridge between the design and implementation phases and apply it to the domain of *software product line engineering* [20].

Software product line engineering is a software engineering paradigm that promotes reuse throughout software development. Central is the management of *variability*, i.e. "*the commonalities and differences in the applications in terms of requirements, architecture, components, and test artifacts*" [20]. Concretely, software is modularized around features: "any prominent and distinctive aspect or characteristic that is visible to various stakeholders (i.e. end-users, domain experts, developers, etc.)" [15].

FODA Diagrams (FDs) were introduced by Kang et al. as part of the Feature Oriented Domain Analysis (FODA) method [15], and have become one of the standard modeling languages for variability in software product line engineering [20]. The purpose of an FD is to define concisely which combinations of features are allowed. They are commonly accepted as a mechanism to bridge the gap between requirements and design.

In our approach, we do not only use FDs to bridge between requirements and design, but also use them to bridge between the design and implementation. Concretely, we propose a bottom-up method that can be used to automatically generate FDs from the changes to the source code. Afterwards, we show that the generated FDs (1) provide links between implementation and design and (2) can be used to reveal possible inconsistencies between the intended design and the actual implementation.

The paper is structured as follows. Section 2 recalls the state-of-the-art concepts of feature-oriented programming, FDs, change-oriented programming and introduces change-based feature-oriented programming. Moreover, it presents the first contribution of this paper: the change-based FDs. Section 3 presents the second contribution of this paper: a formal model for change-based FOP, which is mapped to FDs in Section 4. In order to validate the approach, we present a proof-of-concept in Section 5 after which we conclude and discuss about future work in Section 6.

2. CHANGE-BASED FEATURE-ORIENTED PROGRAMMING

Feature-oriented programming (FOP) is a programming paradigm that targets the separation of concerns [18]. In FOP, every concern is modularized as a separate *feature*:

[*]Dr. Ebraert is funded by the "Agentschap voor Innovatie door Wetenschap en Technologie" via the Optimma research project.

[†]This work has been carried out in the context of the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'11 March 21-25, 2011, TaiChung, Taiwan.
Copyright 2011 ACM 978-1-4503-0113-8/11/03 ...$10.00.

a first-class entity that forms the basic building block of a software system [2, 4, 6]. Features satisfy intuitive user-formulated requirements on the software system and can be composed to form different variations of the same system [3].

Most state-of-the-art approaches to FOP specify a feature as a set of building blocks. An alternative, already pointed out by Batory et al. in [4], is to see a feature as a function that modifies a program, so that feature composition becomes function composition. A feature is then interpreted as a function that takes a program, modifies it by adding the functionality that implements the feature's requirement, and returns the modified program. The application of a feature to a program yields a new program, extended by that feature, to which in turn some other feature can be applied. In that view, a system is obtained by applying a sequence of features to a base program. The AHEAD tool chain [6] was recently formalized this way [2].

2.1 FODA Diagrams

Basically, a FODA Diagram (FD) is a hierarchy of features, where the hierarchy relation denotes decomposition. An example FD is shown in Figure 1. It presents the design of a buffer application, which is closely mapped to the required functionalities of a buffer (logging, restoration, multiple restoration). The FD represents the set of allowed feature combinations (called *configurations*), thus a set of sets of features, and several types of decomposition operators determine what is allowed and what not. An *and* decomposition, for instance, means that all child-features need to be included in a configuration if their parent is, while an *or* decomposition requires at least one child-feature to be included. These two decomposition types can be represented with a generic cardinality decomposition $\langle i..j \rangle$ where i indicates the minimum number of children required in a configuration and j the maximum. In fact the buffer from Figure 1 only contains *and* decompositions, and simulates *or* decompositions by making use of optional features, generally represented with a hollow circle above them. They do not need to be included in a configuration, even if mandated by the decomposition operator. In addition to the decomposition operators, an FD can be annotated by constraints in a textual language, such as propositional logic [5], that further restrain the set of allowed configurations.

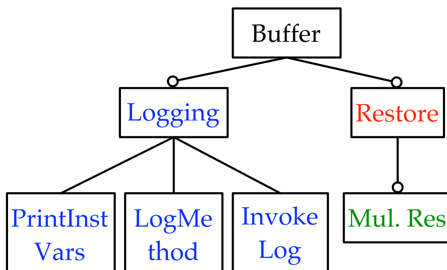

Figure 1: Buffer FODA diagram

A number of FD dialects have appeared in the literature since their original proposal [22]. In this paper, we use the visual syntax of Czarnecki and Eisenecker [8], and the formal semantics of Schobbens et al. [22] which we recall in the following definitions.

DEFINITION 1 (FD ABSTRACT SYNTAX). *an FD d is a 6-tuple $(N, P, r, \lambda, DE, \Phi)$ where:*

- *N is the (non empty) set of features (or* nodes*),*
- *$P \subseteq N$ is the set of primitive features (i.e. those considered relevant by the modeller),*
- *$r \in N$ is the root,*
- *$DE \subseteq N \times N$ is the decomposition (hierarchy) relation between features. For convenience, we will write $n \to n'$ sometimes instead of $(n, n') \in DE$.*
- *$\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition operator of a feature, represented as a cardinality $\langle i..j \rangle$ where i indicates the minimum number of children required in a configuration and j the maximum (we use angle brackets to distinguish cardinalities from other tuples).*
- *$\Phi \in \mathbb{B}(N)$ is a propositional logic formula over the features N, expressing additional constraints on the diagram.*

Furthermore, each d must satisfy the following rules:

- *r is the root $\forall n \in N (\nexists n' \in N \bullet n \to n') \Leftrightarrow n = r$,*
- *DE is acyclic $\nexists n_1, .., n_k \in N \bullet n_1 \to .. \to n_k \to n_1$,*
- *Leaf nodes are $\langle 0..0 \rangle$ decomposed.*

DEFINITION 2 (FD SEMANTICS). *Given an FD d, its semantics $[\![d]\!]$ is the set of all* valid feature combinations *$FC \in \mathcal{PP}N$[1] restricted to primitive features: $[\![d]\!] = \{c \cap P | c \in FC\}$, where the valid feature combinations FC of d are those $c \in \mathcal{P}N$ that:*

- *contain the root: $r \in c$;*
- *satisfy the decomposition type:*
  $f \in c \wedge \lambda(f) = \langle m..n \rangle \Rightarrow m \leq |\{g | g \in c \ \wedge \ f \to g\}| \leq n$;
- *include each selected feature's parent: $g \in c \wedge f \to g \Rightarrow f \in c$;*
- *satisfy the additional constraints: $c \models \Phi$.*

The semantics of the diagram in Figure 1 is the set

$$\{\{Buffer\},\\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog\},\\ \{Buffer, Restore\},\\ \{Buffer, Restore, Mul.Res\},\\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog, Restore\},\\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog, Restore, Mul.Res\}\}$$

For the remainder of the paper, we always assume d to denote an FD, and $(N, P, r, \lambda, DE, \Phi)$ to denote the respective elements of its abstract syntax. Before we can explain how we obtain change-based FDs, we first recapitulate the basics of change-oriented programming and then explain how it can serve as a basis for change-based FOP.

2.2 Change-oriented programming

The central idea of change-oriented programming is that a system can be specified by a set *change objects*. These are maintained as tangible, first-class entities, which represent the development actions that were (or have to be) taken in order to produce the system. In [11, 12], we first presented change-oriented programming and explained that the first-class changes can be obtained by logging the developer's actions as he is programming. Other researchers pointed out some uses of encapsulating change as first-class entities. In [21], Robbes shows that the information from the change

[1] $\mathcal{P}N$ is the powerset of N

objects provides a lot more information about the evolution of a software system than central code repositories. In [9], Denker shows that first-class changes can be used to define a scope for dynamic execution and that they can consequently be used to adapt running software systems.

2.3 Change-based FOP

The principles of change-oriented programming can be applied in order to support FOP. This section elaborates on this technique, which we call change-based FOP.

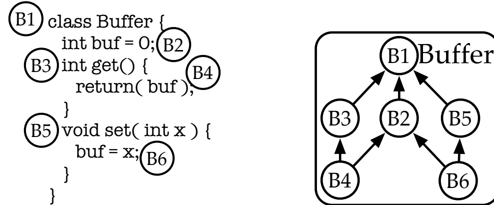

Figure 2: Buffer: (left) source (right) change objects

Figure 2 shows the source code of a $Buffer$ class on the left, and a diagram on the right. In *change-based FOP*, a *feature* is a set of change objects, representing the incremental development actions taken to implement it. It is specified by the meta-data provided by the developer in each change and it is drawn as a rectangle with rounded corners, a name, and change objects inside it. The example consists of the sole feature $Buffer$. Each change object has a unique ID drawn inside a circle: $B1$ is a change that creates the class, $B4$ and $B3$ add an accessor for the instance variable buf, and so on. To make the example more intuitive, the code is annotated with the IDs of the change objects to indicate in what they consist. The arrows between change objects denote *dependencies*. For instance, $B4$ depends on the change that adds the method itself (viz. $B3$) and on the change that adds the instance variable that it accesses (viz. $B2$). These dependencies stem from the programming language used, and reflect its grammar (e.g. a statement must be inside a method) and static semantics (e.g. a variable must exist in the scope in which it is used). A diagram of features, changes and dependencies is called a *change specification*. The dependencies make explicit that the depending change cannot be applied without applying the dependent change. They represent the interaction between the change objects and can be used to detect anomalies in the composition of software variations.

Continuing with the illustration, Figure 3 shows how the $Buffer$ class is gradually extended with three more features: *Restore*, *Logging*, and $MultipleRestore$. The restore feature restores the value of the buffer, the logging feature logs the values of all instance variables whenever a method of the buffer is executed, and the multiple restore feature allows the buffer to restore more than one value. The left half of Figure 3 shows the annotated code, while the right half shows the change specification. As can be seen in the change specification, features can be *nested* by drawing them inside another. In addition, sub-features and changes can be optional or mandatory, where optionality is denoted by a dashed border. This optionality is also specified by the meta-data in the changes, which is provided by the developer. When a nested feature is applied to a system, all its mandatory sub-features and changes have to be applied as well, whereas optional ones may be left out. Take, for instance, the *Logging* feature. Indeed, its implementation can be broken down into three distinct activities: (1) a method *logit* needs to be created, (2) it needs to be filled with print statements for all instance variables, and (3) it has to be called in all methods of $Buffer$, resulting in three sub-features $LogMethod$, $PrintInstVars$ and $InvokeLog$ respectively. They are all mandatory wrt. *Logging*, whereas the changes inside $PrintInstVars$ and $InvokeLog$ are not. This is because they are only needed when one of the logged variables, introduced by the other features, exists.

Unlike the other FOP approaches (that only support top-down FOP), change-based FOP also supports bottom-up FOP in which the individual base elements of the system are first specified in great detail. These elements are then linked together to form features, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

2.4 Change-based FODA diagrams

The change specification of an application can be generated automatically by recording the changes applied to the source code of an application. A change specification is very similar to a FODA diagram – in the sense that it contains the notions of “features” and “feature decomposition”. But it is also different – in the sense that it is created in a bottom-up way (starting from the implementation) and that it does not only include design information (such as the composition of features), but also fine-grained implementation details (such as which changes were applied to implement the features and the dependencies amongst them).

A change specification (such as in the one in Figure 3) translates almost immediately into an FD (like Figure 1). Each feature will be *and*-decomposed, since that is the only decomposition type of the change specification (the only source of variability being the optionality of a feature). The changes with their dependencies, however, also need to be represented as part of the FD. This can easily be done by considering every change object as a leaf feature of the FD. The dependencies between changes, however, crosscut the hierarchy and can therefore not be represented this way. Instead, we capture them in the FD constraints (the Φ). Finally, features and changes can be optional wrt. their parent. This translates immediately to optional features.

The outcome of the translation is a diagram such as the one in Figure 4. We call it a *change-based FODA diagram*. The particularity of such an FD is that it contains, unlike most FDs obtained from analysts, implementation details that were recorded as the code was written. The level of granularity is the statement, which is very low.

The change-based FDs provide a useful piece of documentation that links between implementation and design. Moreover, they can be used to reveal possible inconsistencies between the intended design and the actual implementation. Take for instance the generated FD from Figure 4. The righthand side of this figure shows the additional FD constraints (Φ) that were extracted from the dependencies between the changes. One of these constraints reveals that the optional change $M6$ depends on $L5$, and consequently that the implementation of the $MultipleRestore$ feature depends on the implementation of the *Logging* feature. As the original design of the buffer (from Figure 1) did not include such dependency, a discrepancy was found between

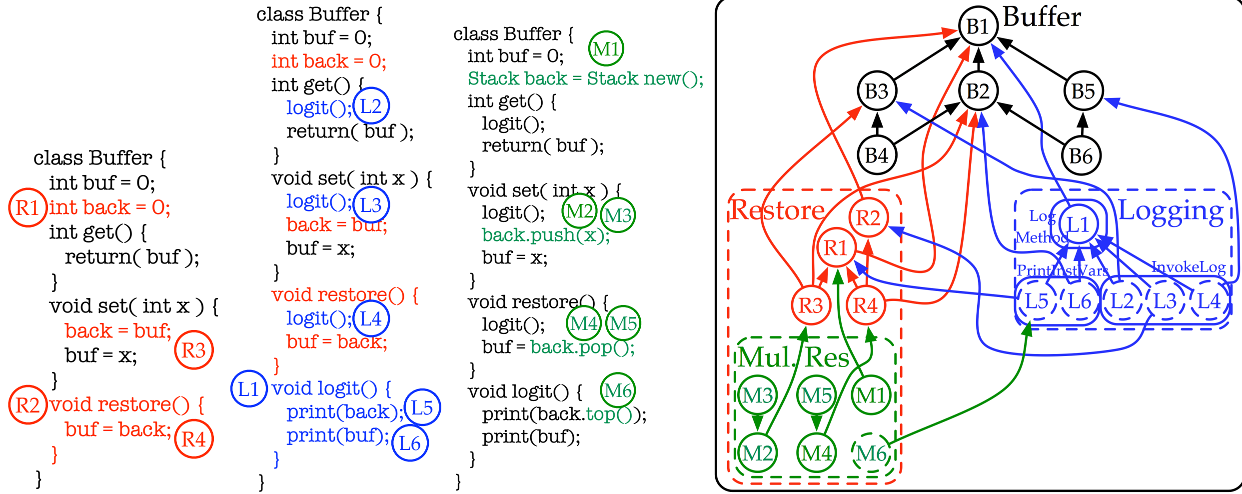

Figure 3: Left to right: source of ***Restore*****,** ***Logging*****, and** ***MultipleRestore*** **and the change specification**

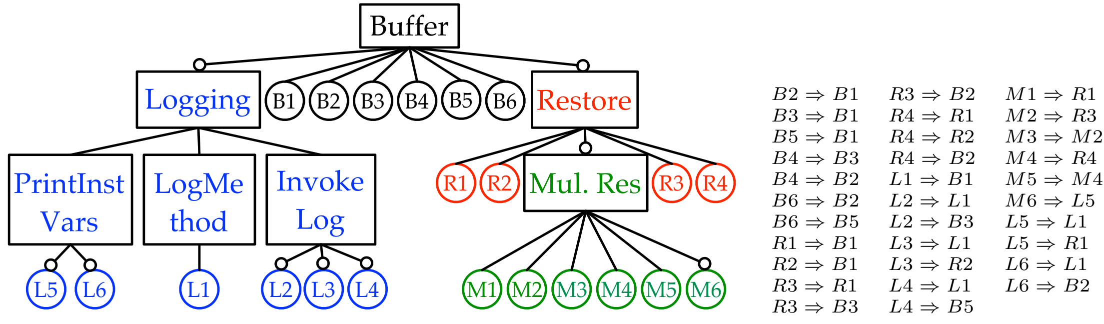

Figure 4: Change-based FODA diagram representing the Buffer change specification.

the design and the implementation. Note that there is even another "unforeseen" dependency that shows up in Figure 4, as *Logging* turns out to depend on *Restore*.

Intuitively, the "meaning" of the change-based FD (Figure 4) is the same as that of the original change specification (Figure 3) because it preserves all of its constraints, meaning that if a set of features (consisting of normal ones and those representing changes) satisfies the FD, it is also a legal change composition/feature set. This property, however, needs to be established formally, not only for the one example here, but also for the algorithm that does the translation. For that, we first establish a formal model of change-based FOP, then present the translation algorithm and finally, present a proof-of-concept.

3. A FORMAL MODEL FOR CHANGE-BASED FOP

In this section, we first provide a formal model of the intuitive notions of change-based FOP presented in Section 2.3 and define some basic properties such as *composability*.

3.1 Fundamental concepts

The main concept in change-based FOP is the *change object*, which encapsulates a development operation. A change can be applied to a software system in order to execute the development operation it encapsulates. Let C be the set of all change objects that make up the system. Another important concept is that of a *feature*, so let F denote the set of all features f_i in the system. As seen in Section 2.3, features consist of changes and can have sub-features.

First, consider the sub-feature relation. A feature f_i can consist of sub-features, which can each be mandatory (man) or optional (opt), as captured by the relation Sub

$$Sub \subseteq F \times F \times \{man, opt\}, \tag{1}$$

where the first element denotes the parent feature and the second the child. For convenience, we will note

$$\begin{array}{ll} f_1 \overset{man}{\rightarrow} f_2 & \text{if } (f_1, f_2, man) \in Sub \\ f_1 \overset{opt}{\rightarrow} f_2 & \text{if } (f_1, f_2, opt) \in Sub \\ f_1 \overset{?}{\rightarrow} f_2 & \text{if } (f_1, f_2, man) \in Sub \vee (f_1, f_2, opt) \in Sub. \end{array}$$

The relation Sub needs to satisfy two well-formedness constraints.

- A sub-feature is either mandatory or optional.

$$\forall f_1, f_2 \in F \bullet \neg(f_1 \overset{man}{\rightarrow} f_2 \wedge f_1 \overset{opt}{\rightarrow} f_2) \tag{2}$$

- The relation contains no cycles, and each feature has no more then one parent.

$$\{(f_1, f_2) | f_1 \overset{?}{\rightarrow} f_2\} \text{ forms a forest} \tag{3}$$

A feature generally consists of changes $c \in C$ which can also be mandatory (*man*) or optional (*opt*). This is formalized with the function $F4C$

$$F4C : C \rightarrow F \times \{man, opt\}, \quad (4)$$

which associates to each change its parent feature. For convenience, we will note

$$\begin{array}{ll} f \overset{man}{\rightarrow} c & \text{if } F4C(c) = (f, man) \\ f \overset{opt}{\rightarrow} c & \text{if } F4C(c) = (f, opt) \\ f \overset{?}{\rightarrow} c & \text{if } F4C(c) = (f, man) \vee F4C(c) = (f, opt). \end{array}$$

The dependencies between changes, finally, are denoted by the relation D,

$$D \subseteq C \times C, \quad (5)$$

which is required to be irreflexive (a change does not depend on itself), asymmetric (changes cannot be mutually dependent) and transitive. In other words, D is a strict partial order over C.

In addition to the well-formedness constraints on Sub, we require that each feature must have sub-features, changes or both.

$$\forall f \in F \bullet (\exists f' \in F \bullet f \overset{?}{\rightarrow} f') \vee (\exists c \in C \bullet f \overset{?}{\rightarrow} c) \quad (6)$$

Considered together, all these concepts make up a *change specification* as the following definition records.

Definition 3 (Change specification). *A* change specification *Cs is a 5-tuple $Cs = (C, F, Sub, F4C, D)$, where $C, F, Sub, F4C$ and D are as defined above.*

3.2 Properties

From the fundamental concepts, we can now define properties that may be required in certain circumstances. Let us first define what a *change composition* is.

Definition 4 (Change composition). *A* change composition *is a set of changes $H \subseteq C$ (with $H \neq \emptyset$) that may be applied to a base system.*

Definition 5 (Legal change com & feature set). *A* legal change composition *H is a change composition such that there exists a* legal feature set *$G \subseteq F$, which satisfies the following constraints*

- *If a feature is selected, its parent feature must be selected, too*

$$\forall f \in G \bullet r \overset{?}{\rightarrow} f \implies r \in G \quad (7)$$

- *If a feature with mandatory sub-features is selected, those need to be selected, too*

$$\forall r \in G \bullet r \overset{man}{\rightarrow} f \implies f \in G \quad (8)$$

- *Let $M = \{c | f \in G \wedge f \overset{man}{\rightarrow} c\}$, the set of mandatory changes and $O = \{c | f \in G \wedge f \overset{opt}{\rightarrow} c\}$, the set of optional changes. We need that*
  - *all changes that are mandatory wrt. the selected features are in: $M \subseteq H$*
  - *all changes stem from selected features: $H \setminus M \subseteq O$*
- *All dependencies are satisfied*

$$\forall c \in H \bullet \exists c' \bullet (c, c') \in D \implies c' \in H \quad (9)$$

By extension, we will say that such a G is a *legal feature set* for H wrt. Cs; or that the changes H are *composable*. Intuitively, G is the set of features in which the changes of H are contained (through $F4C$).

Definition 6 (Semantics of a change spec.). *The semantics of a change specification Cs, noted $[\![Cs]\!]$, is defined as the set of pairs (H, G) such that H is a legal change composition of Cs and G is a legal feature set for H wrt. Cs according to the above definition.*

4. AUTOMATIC GENERATION OF FDS

The goal of the present section is to define a way to automatically generate change-based FDs, so that the resulting FD has the "same meaning" as the change specification. Such a procedure has two main benefits with respect to filling the gap between requirement specification, design and implementation. First, the generated FDs provide links between implementation and design which bring along traceability between both phases. Second, they can be used to reveal possible inconsistencies between the intended design and the actual implementation. Additional benefits include the fact that, having a formal FD opens the way for automated queries and reasoning on the change specification through the use of an FD tool such as FAMA [7] or the one described in [17]. In particular, this allows a safe and efficient integration of change-based FOP with those tools. Let us first give an intuition of how such a translation might look like and what the "same meaning" means.

4.1 Translating the formalism

A general procedure to translate a change specification into an FD is given by Algorithm 1. As can be seen in the `Mapping root features` part, one thing that the previous example did not account for is the fact that a change specification does not necessarily have a root. An FD, however, needs to have one, which is why the algorithm starts by creating an artificial root r, and making each of the top level features an optional child of that root. This and other subtleties are made explicit in Algorithm 1.

The result from applying this algorithm to the change specification of Figure 3 is presented in Figure 5. It is more complex but semantically equivalent to the FD of Figure 4. Actually, the algorithm makes abundant use of dummy features, not only for the root, but also to express optionality. These dummy features are, however, not primitive, and will not appear in the semantics of the resulting FD.[2]

The following theorem formalizes an important property of this algorithm, which we intuitively referred to as the resulting FD having the "same meaning" as the original change specification. More formally, the algorithm *preserves the semantics* of the change specification.

Theorem 7 (Correctness of algorithm 1). *Let $cs2fd$ denote the translation function described by algorithm 1. Then for each change specification Cs, $flatten([\![Cs]\!]) = [\![cs2fd(Cs)]\!]$ where $flatten(Set) = \{a \cup b | (a, b) \in Set\}$.*

Given algorithm 1 and Theorem 7, it is possible to automatically translate an automatically generated change specification Cs into an FD d whose legal products are exactly

[2]Note that dummy features only appear in the translation; they are thus 'hidden' from the user.

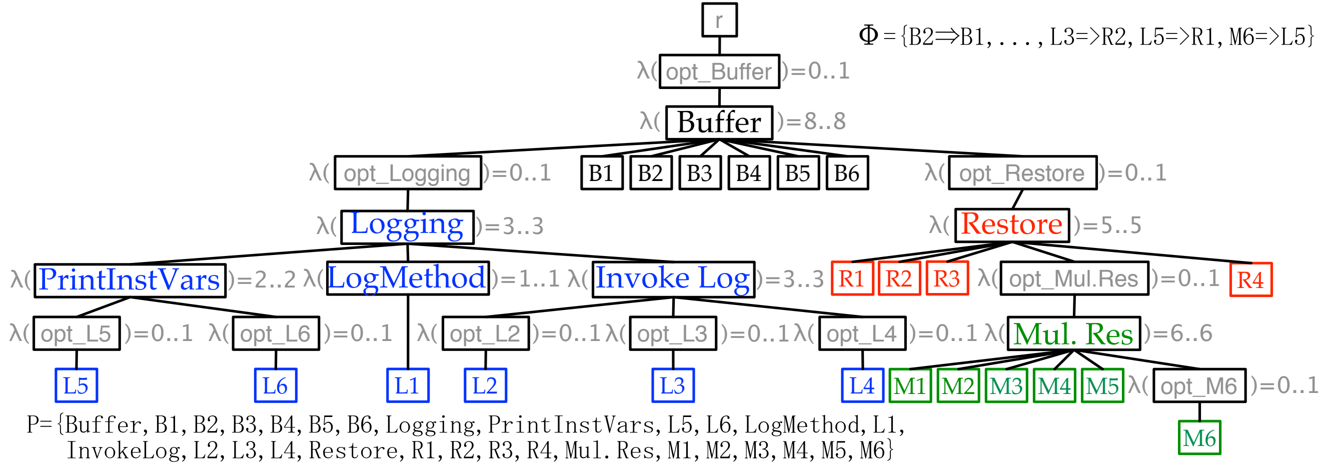

Figure 5: Buffer change-based FODA Diagram resulting from the translation algorithm

Input: A change specification $Cs = (C, F, Sub, F4C, D)$
Output: an FD $d = (N,\ P,\ r,\ \lambda,\ DE,\ \Phi)$

`% Initialisations`

$r \leftarrow$ a new fresh node;
$P \leftarrow C \cup F$;
$N \leftarrow P \cup \{r\}$;
$(\lambda, DE, \Phi) \leftarrow (\emptyset, \emptyset, \emptyset)$;

`% Mapping root features`

Let $roots \leftarrow \{f \mid f \in F \wedge \neg\exists f' \cdot f' \overset{?}{\rightarrow} f\}$;
Let $i \leftarrow 0$;
foreach $n \in roots$ **do**
  $f \leftarrow$ a new fresh node;
  $N \leftarrow N \cup \{f\}$;
  $\lambda \leftarrow \lambda \cup \{f \mapsto card_1[0..1]\}$;
  $DE \leftarrow DE \cup \{(r, f), (f, n)\}$;
  $i \leftarrow i + 1$;
end
$\lambda \leftarrow \lambda \cup \{r \mapsto card_i[i..i]\}$;

`%Mapping non-root features & changes`

foreach $f \in F$ **do**
  $i \leftarrow 0$;
  foreach
  $n \in \{f' \mid (f, f', x) \in Sub\} \cup \{c \mid F4C(c) = (f, x)\}$ **do**
    if $x{=}man$ **then**
      $DE \leftarrow DE \cup \{(f, n)\}$;
    end
    else
      Let $z \leftarrow$ a new fresh node;
      $N \leftarrow N \cup \{z\}$;
      $\lambda \leftarrow \lambda \cup \{z \mapsto card_1[0..1]\}$;
      $DE \leftarrow DE \cup \{(f, z), (z, n)\}$;
    end
    $i \leftarrow i + 1$;
  end
  $\lambda \leftarrow \lambda \cup \{f \mapsto card_i[i..i]\}$;
end

`% Mapping change dependencies`

foreach $(c, c') \in D$ **do**
  $\Phi \leftarrow \Phi \cup \{\text{“}c \Rightarrow c'\text{”}\}$;
end

Algorithm 1: Transforming a Cs to an FD

the legal change composition/feature set pairs of the change specification. This implies that we can generate d in a bottom-up way – starting from the implementation. Let us now present a proof-of-concept implementation that is capable of producing change specifications that adhere to Definition 3.

5. PROOF-OF-CONCEPT

ChEOPS (Change and Evolution-Oriented Programming Support) is an IDE plugin for VisualWorks, which was created as a proof-of-concept implementation of change-based FOP. It supports change-oriented programming but also has the capability of logging developers producing code in the standard object-oriented way. Behind the scenes, ChEOPS produces fine-grained first-class change objects that represent the development actions taken by the developer. In this section, we first explain its architecture and then show that it maps to the formal model of Section 3.

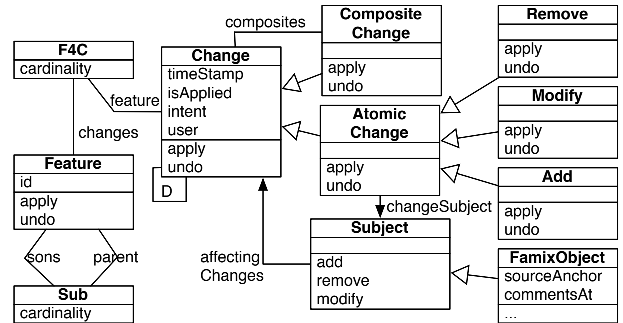

Figure 6: ChEOPS - Core Model

The UML class diagram of the tool's core is presented in Figure 6. We identify three possible actions a developer can take to produce software systems: the addition, the removal and the modification of code. We model them with the classes `Add`, `Remove` and `Modify` respectively. The `Atomic Change` class plays the role of the abstract Command class in the Command design pattern. A `Composite Change` is

composed of Changes (which can in their turn be of any change kind), that have to be applied as a transaction.

The dependency relation between changes is modeled by D: a circular many-to-many relationship on Change. Features are modeled by the Feature class. A feature has a unique name, a set of changes of which it consists and possibly a parent and son features. The relation between features and the changes of which they consist is modeled by the $F4C$ relation, which has a cardinality that denotes whether the change is mandatory or optional with respect to the feature. The relation between features is modeled by the Sub relation, that has a parent, a son and a cardinality that denotes whether the son is mandatory or not with respect to its parent.

The Subject of a change is a building block of the programming language used to develop the software system. The different building blocks of a programming language are specified by the meta-model of that programming language. As a meta-model, we choose the *FAMOOS Information Exchange* model because (1) it allows the expression of building blocks up to the statement level and (2) it provides a generic model to which most class-based programming languages (e.g. Java, C++, Ada, Smalltalk) adhere [10, 23].

5.1 ChEOPS adheres to the formal model

The principal functionality of ChEOPS is to capture the development actions performed within VisualWorks and to reify them into instances of the Change class. The set of all those instances corresponds to the set C of the formal model. Once changes are collected, ChEOPS takes changes that belong together and uses them to generate an instance of the Feature class. The set of all those instances maps to the set F of the formal model, and the grouping of changes into features to the $F4C$ function (Equation 4). The grouping relation is a function, since ChEOPS makes sure that every change belongs to exactly one feature. In the current version, ChEOPS actually implements a slightly stricter $F4C$, since it requires that the changes of a feature are either *all* optional or *all* mandatory, namely $F4C$ with property $\forall c, c' \in C \bullet F4C(c) = (f, x) \wedge F4C(c') = (f, y) \Rightarrow x = y$. The benefit of this is that a developer does not need to specify for every change whether it is mandatory or optional, but only is required to do so on the level of features.

ChEOPS has an interface that allows a developer to group features into more high-level features. The grouping relation actually maps to Sub (Equation 1) of the formal model, in which a parent feature is related to a child feature, which is either mandatory or optional with respect to its parent. The Sub relation implemented by ChEOPS hence satisfies the Sub relation of the formal model. Moreover, it satisfies properties 2 and 3. Property 2 holds since the particular implementation of Sub implies that every feature will either be optional or mandatory wrt. its parent. Property 3 holds because ChEOPS ensures that the relation Sub over F only contains trees. For that, it imposes two restrictions on the grouping of features. First, a feature can never be part of more than one other feature. Second, a feature can never be included into a feature that it already consists of.

The dependencies between changes are imposed by the meta-object protocol of the programming language used in the IDE. ChEOPS is capable of identifying all kinds of dependencies for dynamically typed programming languages that adhere to the FAMIX model [23], and records them while changes are applied. The dependency relation hence recorded can be mapped to D (Equation 5) of the formal model. It satisfies the properties required for D, since it is: *irreflexive* (a change can never depend on itself as that would mean that it would never be applicable in the IDE), *asymmetric* (if a change c_1 depends on c_2, c_2 never depends on c_1 as that would mean that both c_1 and c_2 would never be applicable) and *transitive* (if a change c_1 depends on c_2 and c_2 depends on c_3, c_1 always depends on c_3. If c_1 can only be applied if c_2 is applied and c_2 can only be applied if c_3 is applied, we can indeed say that c_1 can only be applied if c_3 is applied).

Finally, ChEOPS adheres to Equation 6 as it only allows to create a feature by grouping changes and/or features, thus every feature in ChEOPS necessarily consists of subfeatures, changes, or both. From this, we can conclude that ChEOPS completely adheres to the formalisms explained in Section 3.1 and that we can safely say that each change specification created with ChEOPS adheres to Definition 3. Moreover, by using translation Algorithm 1, the generated change specifications can be automatically translated into a semantically equivalent change-based FODA diagram.

5.2 Change-based FODA bridge between design and implementation

In order to validate that change-based FODA diagrams (1) provide links between implementation and design and (2) can be used to reveal possible inconsistencies between the intended design and the actual implementation we refer back to Section 2.4. While the many similarities between the original FODA diagram of Figure 1 and the generated change-based-FODA diagram of Figure 4 support the former claim, the unveiled discrepancies between both ($MultipleRestore$ depending on *Logging* and *Logging* depending on *Restore*) support the latter. Consequently, we conclude that change-based FODA diagrams support bridging between the design and the implementation of feature-oriented software.

6. CONCLUSION AND FUTURE WORK

Feature-oriented programming is a style of programming that targets the separation of concerns. In feature-oriented programming, every concern is modularized as a separate *feature*: a first-class entity that forms the basic building block of a software system. In feature-oriented programming, software is developed by means of a development process that encompasses different phases: the specification of the requirements, design, implementation, verification and maintenance. Changes during one of these steps may have side effects on the others. One of the basic side effects of this development process is the emergence of a gap between the different phases.

In this paper, we make an effort to fill this gap with two contributions. First we propose a formal model of change-based feature-oriented programming (a bottom-up approach to feature-oriented programming), define properties such as composability, and then show that ChEOPS – an implementation of change-based feature-oriented programming – adheres to this model. Second, we map the model to the well-understood notion of FODA Diagrams (FDs), which has become one of the standard modeling languages for designing variability in software product line engineering. The mapping – provided in form of a translation algorithm – allows us to automatically generate FDs starting from the

implementation. The generated FDs contain both coarse-grained design information and fine-grained implementation details. They can be used as a documentation artifact, to bridge between the design and implementation and to reveal inconsistencies between both.

Currently, the dependencies between the change objects reflect low-level constraints only. The new connection between FDs and the change objects also allows to express dependencies and constraints on the application level. One track of future work may consist in how to carry back such dependencies and constraints to the change object level. This can further support the bridging of the gap between design and implementation. Another track of future work consists in extending the range of applications of FDs. In this paper, we only used FDs to bridge between the design and implementation. The state-of-the-art work on FDs, however, includes many more applications such as visual modeling support, specification of metrics, program understanding, etc. which we plan to investigate in order to find out how they can be helpful. A third track encompasses the development of an algorithm that can automatically detect inconsistencies between FDs created in the design and a change-based FDs generated from the implementation. Moreover, such an algorithm can also be used to measure the coupling – in terms of dependencies – that exists between the feature modules.

7. REFERENCES

[1] M. Amor, L. Fuentes, and A. Vallecillo. Bridging the gap between agent-oriented design and implementation using mda. In *In Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004), LNCS 3382*, pages 93–108, 2004.

[2] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. An algebra for feature-oriented software development. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*. Springer-Verlag, 2007.

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Transactions on Software Engineering*, 30(6):355–371, 2004.

[5] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th Int. Software Product Line Conference (SPLC)*, pages 7–20, 2005.

[6] D. S. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *GTTSE*, pages 3–35, 2006.

[7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Proceedings of the 17th International Conference (CAiSE'05) LNCS, Advanced Information Systems Engineering.*, 3520:491–503, 2005.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, 2000.

[9] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[10] S. Ducasse and S. Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, 1999.

[11] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 3–24, New York, NY, USA, 2007. ACM.

[12] P. Ebraert, E. Van Paesschen, and T. D'Hondt. Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel, 2007.

[13] N. Eén and N. Sörensson. An extensible sat-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[14] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 189–201. Springer Berlin / Heidelberg, 2002.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, November 1990.

[16] J. Marco and X. Franch. Bridging the gap between design and implementation of component libraries. Technical Report LSI-00-79-R, Universitat Politecnica de Catalunya, 2000.

[17] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India, October 2007.

[18] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[19] F. Paterno. Model-based design of interactive applications. *Intelligence*, 11(4):26–38, 2000.

[20] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[21] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, pages 93–109, 2007.

[22] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks (2006), special issue on feature interactions in emerging application domains*, page 38, 2006.

[23] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.