Feature Diagrams for Change-Oriented Programming

Peter EBRAERT [a] Andreas CLASSEN [b,1] Patrick HEYMANS [b] Theo D'HONDT [a]

[a] *Computer Science Department, Vrije Universiteit Brussel, 1050 Brussels, Belgium, {pebraert,tjdhondt}@vub.ac.be*

[b] *PReCISE Research Centre, Faculty of Computer Science, University of Namur, 5000 Namur, Belgium, {acs,phe}@info.fundp.ac.be*

Abstract. The idea of feature-oriented programming is to map requirements to features, concepts that can be composed to form a software product. Change-oriented programming (ChOP), in which features are seen as sets of changes that can be applied to a base program, has recently been proposed as an approach to FOP. Changes are recorded as the programmer works and can encapsulate any developer action, including the removing of code.

Before changes can be combined to form a product, it has to be verified that there are no harmful interactions between selected changes. There exists, however, no formal model of the current approach that may serve as a reference specification for ChOP implementations. In an effort to fill this gap, we propose a formal model of ChOP, which, as we will show, maps to the well-understood notion of feature diagram. Thanks to this, we can reuse a number of results in feature diagram research and apply them to ChOP.

Keywords. Formal Methods, Feature-Oriented Programming, Feature Diagrams, Software Product Lines

1. Introduction

Software Product Line Engineering (SPLE) is a software engineering paradigm that institutionalises reuse throughout software development. Central to SPLE is the management of *variability*, i.e. "*the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts*" [1].

Several methods for implementing variability in SPLE exist [2]. Traditional approaches sometimes lack modularity and reusability [3] or require a significant amount of manual labour [4]. *Feature-Oriented Programming* (FOP) overcomes these problems by using *features* as first-class abstractions during the development process [5,6,7]. A feature closely maps to a variant of a variable requirement and variability can be implemented mere by selecting or deselecting features. In *Change-Oriented Programming* (ChOP), a specific approach to FOP, each feature is represented by a set of *changes* applied to a base system [8,9]. Feature composition, in this case, becomes change composition. One of the challenges in ChOP is to make sure that a composition of several

[1] FNRS Research Fellow.

features, viz. changes, is free of harmful interactions. However, there is currently no formal framework for ChOP, that would allow to define this kind of property and serve as a reference for ChOP implementations.

In an effort to fill this gap, we propose a formal model of ChOP. First, we formalise its concepts and properties using common set theory. Then, we show how our implementation of ChOP conforms to the formal model. Furthermore, we define properties such as *composability* of a set of changes (i.e. the fact that they are free of harmful interactions), that need to be checked before they can be composed.

An interesting observation is that the model is quite close to Feature Diagrams (FD), a SPLE notation used to model the variability and manage the interactions of an application at an – up to now – relatively high level of granularity [10,11,12]. The main purpose of FDs it to model which combinations of features are allowed and disallowed in a SPL. By mapping the formal model of ChOP to FDs, we are able to reuse their well-studied semantics as well as existing analysis tools. We prove that *composability* of features in ChOP is equivalent to checking whether the product they form satisfies the FD.

The paper is structured as follows. Section 2 recalls the concepts of FOP and ChOP, while Section 3 remembers the definition of FDs. Section 4 introduces a formal model for ChOP, which is mapped to FDs in Section 5. The paper is concluded in Section 6 with a discussion on future work.

2. Feature and change-oriented programming

FOP is a programming paradigm that targets the separation of concerns [13]. In FOP, every concern is modularised as a separate *feature*: a first-class entity that forms the basic building block of a software system [6]. Features satisfy intuitive user-formulated requirements on the software system and can be composed to form different variations of the same system [14].

Most state-of-the-art approaches to FOP specify a feature as a set of building blocks. An alternative, already pointed out by Batory in [6], is to see a feature as a function that modifies a program, so that feature composition becomes function composition. Indeed, a feature would be seen as a function $prog \rightarrow prog$ 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 toolchain [7] was recently formalised this way [5].

In [8] and [9] we proposed ChOP, a novel incarnation of FOP, that uses *change* as the main development entity.

2.1. Change-oriented programming

The central idea of ChOP is that a system can be specified by a set of first-class *change objects*, which represent the development actions that were (or have to be) taken in order to produce it. There are several ways of obtaining these change objects. A classic way is to take two finished versions of a software system and to execute a Unix `diff` command on their respective abstract syntax trees [15], revealing the changes. This approach,

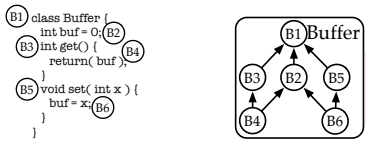


Figure 1. Buffer: (left) source (right) change objects

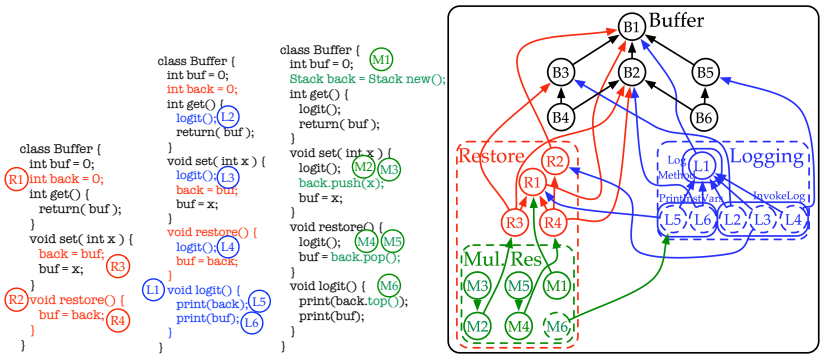


Figure 2. Left to right: source of $Restore$, $Logging$, and $MultipleRestore$ and the change specification

however, only works *a posteriori*, and at a high level of granularity (the version level). A more subtle alternative is to log the developer's actions as he is performing the changes. This can be done, for instance, by taping into the IDE, and extending it with a mechanism that captures the developer actions and stores them as change objects. In [8,9], we opted for the latter approach since it provides a more complete overview of all development actions [16].

Figure 1 shows the source code of a $Buffer$ class on the left, and a diagram on the right. As mentioned before, a *feature* in ChOP is a set of change objects, representing the incremental development actions taken to implement it. 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 *structural 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 structural dependencies stem from the programming language used, and reflect its grammar (e.g. a statement must be inside a method) and semantics (e.g. a variable must exist in the scope in which it is used). A diagram of features, changes and structural dependencies is called a *change specification*.

Continuing with the illustration, Figure 2 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 upper half of Figure 2 shows the annotated code, while the lower half shows the change specification. As can be seen in the change specification, features can be *nested* by drawing them inside another. In

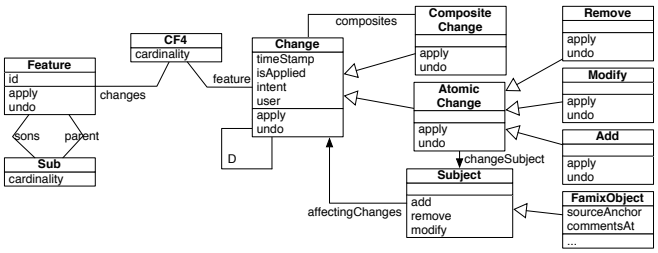


Figure 3. ChEOPS - Core Model

addition, sub-features and changes can be optional or mandatory, where optionality is denoted by a dashed border. 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.

2.2. Proof-of-concept implementation

ChEOPS *(Change and evolution-oriented programming support)* is an IDE plugin for VisualWorks, which was created as a proof-of-concept implementation of ChOP. ChEOPS fully supports ChOP 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. The UML class diagram of the tool's core is presented in Figure 3.

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 structural dependency relation between changes is modelled by D: a circular many-to-many relationship on `Change`. Features are modelled by the `Feature` class. `Feature` is a class that has a unique name, a set of changes of which it consists and possibly a parent and sons 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

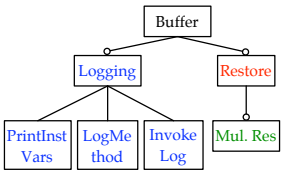


Figure 4. Buffer FD

choose the *FAMOOS Information Exchange* model because (a) it allows the expression of building blocks up to the statement level and (b) it provides a generic model to which most class-based programming languages (e.g. Java, C++, Ada, Smalltalk) adhere [17, 18].

3. Feature Diagrams

FDs were introduced by Kang et al. as part of the FODA method [10], and have become one of the standard modelling languages for variability in SPLE [1]. An example FD, based on the buffer example, is shown in Figure 4.

Basically, an FD is a hierarchy of features, where the hierarchy relation denotes decomposition. 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. Some authors also consider optional features, generally represented with a hollow circle above them, which need not 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 [11], that further restrain the set of allowed configurations.

A number of FD dialects have appeared in the literature since their original proposal [12]. In this paper, we use the visual syntax of Czarnecki and Eisenecker [19], and the formal semantics of Schobbens et al. [12] which we recall in the following definitions.

Definition 1 (FD Abstract Syntax)**.** *an FD d is a 5-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 $n \rightarrow n'$ sometimes instead of $(n, n') \in DE$.*
- *$\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$ 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 conjunction of propositional logic formulae on features, expressing additional constraints on the diagram.*

Furthermore, each d *must satisfy the following well-formedness rules:*

- r *is the root* $\forall n \in N(\nexists n' \in N \bullet n \rightarrow n') \Leftrightarrow n = r$,
- *DE is acyclic* $\nexists n_1, .., n_k \in N \bullet n_1 \rightarrow .. \rightarrow n_k \rightarrow n_1$,
- *Terminal 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 $FC \in \mathcal{P}\mathcal{P}N$ *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:* $f \in c \wedge \lambda(f) = \langle m..n \rangle \Rightarrow m \leq |\{g \in c | f \rightarrow g\}| \leq n$*;*
- *include each selected feature's parent:* $g \in c \wedge f \rightarrow g \Rightarrow f \in c$*;*
- *satisfy the additional constraints:* $c \models \Phi$.

The semantics of the diagram in Figure 4 is the set

$$
\begin{array}{l}
\{\{Buffer\}, \\
\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog\}, \\
\ \{Buffer, Restore\}, \\
\ \{Buffer, Restore, Mul.Res\}, \\
\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog, \\
\quad Restore\}, \\
\ \{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog, \\
\quad Restore, Mul.Res\}\}
\end{array}
$$

For the remainder of the paper, unless otherwise stated, we always assume d to denote an FD, and $(N, r, \lambda, DE, \Phi)$ to denote the respective elements of its abstract syntax.

4. A formal model for ChOP

In this section, we first provide a formal model of the intuitive notions of ChOP presented in Section 2.1 and define some basic properties such as *composability*. We then show that our tool implementation satisfies the formal model.

4.1. Fundamental concepts

The principal concept in ChOP 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 2.1, 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

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

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.

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

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

$$F4C : C \rightarrow F \times \{man, opt\}, \tag{4}$$

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

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

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

$$D \subseteq C \times C, \tag{5}$$

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.

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

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 $(C, F, Sub, F4C, D)$, where $C, F, Sub, F4C$ and D are as defined above.*

4.2. 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 $H \subseteq C$ that may be applied to a base system.*

Definition 5 (Legal change composition, legal 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:*

$$\forall f \in G \bullet r \xrightarrow{?} f \implies r \in G \tag{7}$$

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

$$\forall r \in G \bullet r \overset{man}{\rightarrow} f \implies f \in G \tag{8}$$

- *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 structural dependencies are satisfied*

$$\forall c \in H \bullet \exists c' \bullet (c, c') \in D \implies c' \in H \tag{9}$$

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*, free of harmful interactions.

Definition 6 (Semantics of a change specification)**.** *The* semantics of a change specification *Cs, noted $[\![Cs]\!]$, is defined as the set of couples (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.*

The following theorem establishes that every change specification has at least one legal change composition, i.e. $[\![Cs]\!] \neq \emptyset$ for every change specification Cs. The proofs for this and subsequent theorems can be found in Appendix A.

Theorem 7 (Legal composition existence)**.** *Given a change specification, there is at least one legal change composition & feature set.* $Cs = (C, F, Sub, F4C, D) \bullet \exists (H, G) \in [\![Cs]\!]$

Given a legal feature set G, there might be several legal change compositions. And similarly, given a legal change composition H, there can be several legal feature sets such that $(H, G) \in [\![Cs]\!]$. For proving this, simply consider the following example. Let

us assume we have a Cs with only $f \overset{opt}{\rightarrow} f'$, $f \overset{man}{\rightarrow} c$ and $f' \overset{opt}{\rightarrow} c'$. In this case, we have $[\![Cs]\!] = \{(\{c\}, \{f\}), (\{c\}, \{f, f'\}), (\{c, c'\}, \{f, f'\})\}$.

Consequently, we will define the notions of minimal and maximal change compositions and prove their unicity.

Definition 8 (Minimal/maximal change composition)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, H is said to be* minimal *(resp.* maximal*) if $\nexists H' \cdot H' \subset$ (resp. $\supset$) $H \wedge (H', G) \in [\![Cs]\!]$.*

In the small example we just gave, $\{c\}$ is both the minimal and maximal change composition wrt $\{f\}$, whereas wrt $\{f, f'\}$, we have a minimal change composition $\{c\}$ and a maximal change composition $\{c, c'\}$. The following theorem proves the unicity of the minimal and maximal change composition in the general case.

Theorem 9 (Unicity of minimal and maximal change compositions)**.** *A minimal change composition is unique. And so is a maximal change composition.*

Similarly, we can define the notions of minimal and maximal feature set.

Definition 10 (Minimal/maximal feature set)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, G is said to be* minimal *(resp.* maximal*) if $\nexists G' \cdot G' \subset$ (resp. $\supset$) $G \wedge (H, G') \in [\![Cs]\!]$.*

Theorem 11 (Unicity of minimal feature sets)**.** *A minimal feature set is unique.*

Theorem 12 (Unicity of maximal feature sets)**.** *A maximal feature set is unique.*

4.3. ChEOPS

Let us show how ChEOPS, the tool introduced in Section 2.2, is capable of producing change specifications that adhere to Definition 3.

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 puts them into 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 $\forall c, c' \in C \bullet F4C(c) = (f, x) \wedge F4C(c') = (f, y) \Rightarrow x = y$.

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 restric-

tions 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 in a feature that it already consists of.

The structural 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 (hierarchical, accessive, invocative and creational dependencies) for dynamically typed programming languages that adhere to the FAMIX model [18], and records them while changes are applied. The structural 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 of c_3 is applied).

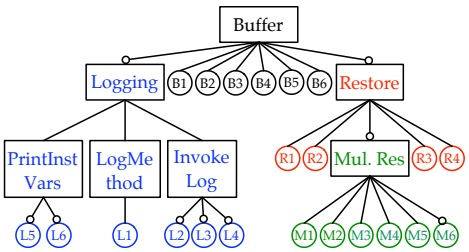
Finally, ChEOPS adheres to Equation 6 as it only allows to create a feature by grouping changes and/or features, thus every feature in ChEOPS necessarily consists of subfeatures, changes or both. From this, we can conclude that ChEOPS completely adheres to the formalisms explained in Section 4.1 and that we can safely say that each change specification created with ChEOPS adheres to Definition 3.

Retrospectively, we realised that given a set of features, ChEOPS currently always produces a maximal change composition. This strategy makes sense, since it produces the system with the most complete implementation of the corresponding feature set. Other strategies are currently being considered for implementation. Namely, allowing the user to create the minimal change composition for a set of features might be interesting in the case where code size needs to be minimised, as it returns the most basic implementation of a feature set. Yet another strategy could be a mix of both in which the optional changes that add code are included and the optional changes that remove code are omitted. This, however, remains a topic for future work.

5. From ChOP to FDs

The goal of the present section is to define a way to systematically translate a change specification into an FD, so that the resulting FD has the 'same meaning' as the change specification. Such a procedure has two main benefits. First, generating and then visualising an FD can provide an alternative representation of the change specification, which is in many cases more readable. Second, and more importantly, 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 [20] or the one described in [21]. In particular, since ChEOPS is a valid implementation of our formal ChOP model, this allows a safe and efficient integration of ChEOPS with those tools. Let us first give an intuition of how such a translation might look like and what 'the same meaning' means.

Figure 4 shows an FD inspired by the buffer example. Even though this diagram is based on the *description* of the buffer example rather than the corresponding change specification (Figure 2), it illustrates part of the translation. Indeed, the change specification is made up of features and changes, as well as the relations between them. The feature hierarchy of ChOP translates almost immediately into an FD (like Figure 4). The



$B2 \Rightarrow B1$ $R3 \Rightarrow B2$ $M1 \Rightarrow R1$
$B3 \Rightarrow B1$ $R4 \Rightarrow R1$ $M2 \Rightarrow R3$
$B5 \Rightarrow B1$ $R4 \Rightarrow R2$ $M3 \Rightarrow M2$
$B4 \Rightarrow B3$ $R4 \Rightarrow B2$ $M4 \Rightarrow R4$
$B4 \Rightarrow B2$ $L1 \Rightarrow B1$ $M5 \Rightarrow M4$
$B6 \Rightarrow B2$ $L2 \Rightarrow L1$ $M6 \Rightarrow L5$
$B6 \Rightarrow B5$ $L2 \Rightarrow B3$ $L5 \Rightarrow L1$
$R1 \Rightarrow B1$ $L3 \Rightarrow L1$ $L5 \Rightarrow R1$
$R2 \Rightarrow B1$ $L3 \Rightarrow R2$ $L6 \Rightarrow L1$
$R3 \Rightarrow R1$ $L4 \Rightarrow L1$ $L6 \Rightarrow B2$
$R3 \Rightarrow B3$ $L4 \Rightarrow B5$

Figure 5. Tentative FD representing the Buffer change specification.

decomposition type of each feature will be an and, 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 structural dependencies, however, also need to be represented as part of the FD. This can easily be done by considering every change object as a feature as a leaf feature of the FD. The structural dependencies between changes, however, crosscut the hierarchy and can therefore not be represented this way. Instead, we capture them by additional FD constraints (the Φ). Finally, features and changes can be optional wrt. their parent. This translates immediately to optional features.

The result of this translation is the diagram of Figure 5. Intuitively, its 'meaning' is the same as that of the change specification in Figure 2 because it preserves all of its constraints, meaning that if a set of features and feature-become changes satisfies the FD, it is also a legal change composition/feature set.

This property, however, needs to be established formally, and for the algorithm that does the translation, rather than for the one example here. Similarly, the properties we want to capture when analysing the FD, and what they mean for the change specification need to be specified formally. This is the goal of the next two sections.

The particularity of an FD obtained from a change specification 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. In realistic cases, the resulting FD will be enormous. To date, we do not have empirical results, but given current industrial-strength SAT solvers,[2] we are confident that this is not a critical limitation.

5.1. 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 algorithms starts by creating an artificial root r, and making each of the top level features an optional child of that root.

The result from applying this algorithm to the change specification of Figure 2 is presented in Figure 6. It is more complex but semantically equivalent to the FD of Fig-

[2] See, for instance, www.cril.univ-artois.fr/SAT07.

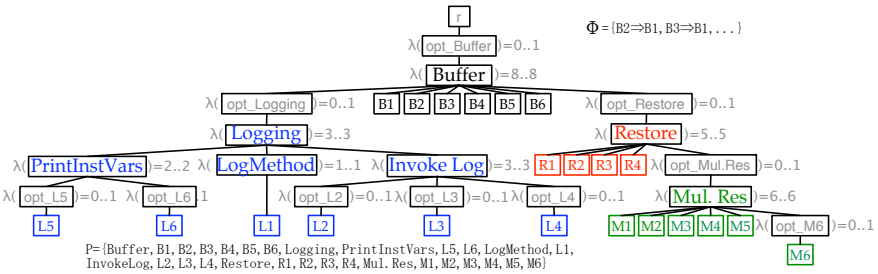


Figure 6. Buffer FD resulting from the translation algorithm

ure 5. 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.

The following theorem formalises 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. The proof is straightforward and was omitted from the paper.

Theorem 13 (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* $flatten(Set) = \{a \cup b | (a, b) \in Set\}$.

5.2. Applications

Given algorithm 1 and Theorem 13, it is possible to translate a change specification Cs into an FD d whose legal products are exactly the legal change composition/feature set pairs of the change specification. This means that we can analyse d instead of Cs but interpret the results in terms of Cs. Here we present several analysis methods for FDs and show how they can be useful in the context of ChOP.

A first application of FDs was already hinted at in the previous sections. Indeed, given a change composition/feature set pair, it is legal iff it is a product of the FD. Which means that we can use FD tools to check the validity of change composition/feature set pairs. Formally, given a change specification Cs, $H \subseteq C$ and $G \subseteq F$, then

$$(H, G) \text{ is legal iff } (H \cup G) \in [\![cs2fd(Cs)]\!].$$

Furthermore, we can use the FD to determine the feature compositions that are legal wrt. a change composition, i.e.

$$fcomp(H) = \{P \cap F | P \in [\![cs2fd(Cs)]\!] \wedge H = P \cap C\},$$

where $H \subseteq C$, or the other way around, with $G \subseteq F$

$$ccomp(G) = \{P \cap C | P \in [\![cs2fd(Cs)]\!] \wedge G = P \cap F\}.$$

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 $roots \leftarrow \{f \mid f \in F \wedge \neg\exists f' \cdot f' \xrightarrow{?} f\}$;
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

If $fcomp(H)$ or $ccomp(G)$ return an empty set, we know that H is an illegal change composition, respectively C an illegal feature set. Otherwise, the results of $fcomp(H)$ (resp. $ccomp(G)$) can easily be used to determine the minimal/maximal feature set (resp. change composition), it suffices to take the element of $fcomp(H)$ (resp. $ccomp(G)$) with minimal/maximal cardinality.

A common analysis means for FDs are metrics defined on the FD [20]. For instance, as FDs are generally used to express the variability of a SPL, the number of products of an FD $|[\![d]\!]|$ is a measure for the variability of the SPL. If the FD was obtained from a change specification, it measures the variability of the change specification. This kind of metric is already implemented in FD tools such as FAMA [20]. Obtaining the same information based on only the data in Figure 2 would require a new algorithm and would consequently imply more work.

A similar metric would be to determine in how many ways a feature set $G \subseteq F$ can be implemented, and how many changes are needed (at most/least) to implement it. This can be done by calculating $fcomp(H)$ and determining the minimal/maximal cardinality of its elements. A configuration tool, i.e. a tool that lets a user choose which features to include, could then show this kind of statistics while performing the choices.

If additional information about changes is available, such as the lines of code added or additional memory consumption, it can be added to the FD in the form of feature attributes. The configuration tool could then show more comprehensive statistics about the user's current feature selection. Instead of seeing merely how many changes it will require, the user will be able to see to what extend the resulting application will grow in code size or memory consumption. Instead of showing metrics to the user, the configuration tool could also choose the change composition itself, for instance by selecting the one that is optimal wrt. an objective function (e.g. minimise memory consumption) [20]. The advantage here is that the attribute values would be automatically derived from the actual changes in the code without the need for human intervention.

Another application of FDs is to determine which features are always present (called *commonality*), $comm(d) = \bigcap [\![d]\!]$, and which are never present (called *dead features*), $dead(d) = P \setminus \bigcup [\![d]\!]$. If we project these results to the sets of changes $comm(cs2fd(Cs)) \cap C$ and $dead(cs2fd(Cs)) \cap C$, we obtain the set of changes which are always/never present in the system. With the current model of ChOP, however, these indicators are not very useful. As shown in the proof of Theorem 7, the change composition/feature set consisting of all changes and features is always legal, hence $dead(cs2fd(Cs)) = \emptyset$. Furthermore, through algorithm 1, each top-level feature of the change composition becomes optional, hence $comm(cs2fd(Cs)) = \emptyset$.

A more subtle and relevant approach would be to consider the number of occurrences of a change or feature among the set of legal compositions. A change with a high frequency ('almost common', one could say) could suggest a refactoring to make the corresponding code efficient, whereas no effort should be put in code that is 'almost dead'. Formally, a tool should indicate, given a change $c \in C$, whether

$$\frac{|\{p \in [\![cs2fd(Cs)]\!] \mid c \in p\}|}{|[\![cs2fd(Cs)]\!]|} \quad > \quad k_1 \;\; (resp. < k_2)$$

where k_1 and k_2 are some fixed thresholds.

6. Conclusion and future work

ChOP, in which features are seen as sets of changes that can be applied to a base program, has recently been proposed as an approach to feature-oriented programming. In ChOP, changes are recorded as the programmer works and can encapsulate any developer action, including the removing of code. Before changes can be combined to form a product, it has to be made sure that they are free of harmful interactions. There exists, however, no formal model of the current approach that would allow to define properties such as this independently from an actual implementation.

In this paper, we make an effort to fill this gap with two contributions. First we propose a formal model of change-oriented programming, define properties such as composability, and then show that ChEOPS, the proof-of-concept implementation of ChOP, adheres to this model. Second, we map the model to the well-understood notion of FDs, which has become one of the standard modelling languages for variability in SPLE. The mapping, provided in form of a translation algorithm, allows us to reuse a number of results in FD research and apply them to ChOP.

The elaboration of the formal ChOP model lead us to take a high-level look at ChOP, independent from an implementation, which allowed us to identify new interesting properties and to make some generalisations. Therefore, an immediate topic for future work is to improve ChEOPS based on the feedback gathered while developing the formal model. For instance, the current version is not expressive enough to cover the whole formal model, given that the relations between a feature and its changes ($F4C$) and between features (Sub) are implemented with the restriction that the optionality of a sub-feature (change) wrt. its parent is an attribute of the parent, and not of the relation. The design and implementation of ChEOPS will be refactored to overcome this issue.

Another example of future work induced by the formal model are the different strategies that can be used to produce legal change compositions. At the time of writing, we only formalised two of them: produce maximal (respective minimal) legal change compositions. We also sketched other conceivable strategies, which would also take into account the nature of a change (addition, deletion, modification). These strategies need to be formalised and can then also be implemented in ChEOPS.

A final track of future work consists in extending the range of applications of FDs to ChOP. In this paper, we only used FDs to validate change compositions and to express basic properties of change specifications. The state-of-the-art work on FDs, however, includes many more applications such as visual modelling support, specification of metrics, program understanding, etc. which we plan to investigate in order to find out how they can be helpful in ChOP.

Acknowledgements

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy under the MoVES project and the FNRS.

References

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

[2] Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software – Practice and Experience **35**(8) (2005) 705–754
[3] Nakkrasae, S., Sophatsathit, P.: A formal approach for specification and classification of software components. In: SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, New York, NY, USA, ACM (2002) 773–780
[4] Ed Jung, Chetan Kapoor, D.B.: Automatic code generation for actuator interfacing from a declarative specification. In: International Conference on Intelligent Robots and Systems. (IROS 2005). 2005 IEEE/RSJ. (2005) 2839 – 2844
[5] Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An algebra for feature-oriented software development. In: Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST). Number MIP-0706. Springer-Verlag (2007)
[6] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. Transactions on Software Engineering (2004)
[7] Batory, D.S.: A tutorial on feature oriented programming and the ahead tool suite. In: GTTSE. (2006) 3–35
[8] Ebraert, P., Van Paesschen, E., D'Hondt, T.: Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel (2007)
[9] Ebraert, P., Vallejos, J., Costanza, P., Van Paesschen, E., D'Hondt, T.: Change-oriented software engineering. In: ICDL '07: Proceedings of the 2007 international conference on Dynamic languages, New York, NY, USA, ACM (2007) 3–24
[10] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University (November 1990)
[11] Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Proceedings of the 9th Int. Software Product Line Conference (SPLC). (2005) 7–20
[12] Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Computer Networks (2006), special issue on feature interactions in emerging application domains (2006) 38
[13] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Comm. ACM **15**(12) (1972) 1053–1058
[14] Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. **1**(4) (1992) 355–398
[15] Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th International Conference on Automated Software Engineering. (2005)
[16] Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (2007) 93–109
[17] Ducasse, S., Demeyer, S.: The FAMOOS Object-Oriented Reengineering Handbook. University of Bern (1999)
[18] Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. PhD thesis, University of Bern (2001)
[19] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston (2000)
[20] Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated Reasoning on Feature Models. Proceedings of the 17th International Conference (CAiSE'05) LNCS, Advanced Information Systems Engineering. **3520** (2005) 491–503
[21] Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: 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), New Delhi, India (October 2007) 243–253

A. Appendix

Proof of Theorem 7. We prove that (C, F) is a valid change composition ($(C, F) \in [\![Cs]\!]$) because it complies to the constraints of Definition 5. Constraints 7, 8 and 9 are satisfied trivially by the equations 1 and 5. Given M and O as defined in Definition 4, we need to show that $M \subseteq H$ and that $H \setminus M \subseteq O$. Because a change is either optional or mandatory to a feature, we know that $F = O \cup M$ and that $O \cap M = \emptyset$. From that we know that both properties hold. □

Proof of Theorem 9. Let us consider building a change composition as follows. We include in H all mandatory changes wrt G and only those optional changes required to satisfy the dependencies steming from mandatory changes. Wrt Definition 5, this means we build:

$$H = M \cup (O \cap \{c \mid c' \in M \wedge (c', c) \in D^+\})$$

where D^+ is the transitive closure of D. Such an H is legal, unique and minimal since we cannot remove any optional change from it without violating a dependency. The unicity of the maximal change composition is proved by considering $H = M \cup O$ which makes H legal, unique and maximal. □

Proof of Theorem 11. Let us assume we have a legal change composition H. All legal G associated to it satisfies $\forall c \in H \cdot G \subseteq \{f \mid f \xrightarrow{?} c\}$ since all changes need to be justified by a feature. Since H is fixed, one can only add features to G that do not require more changes being added to H.

Minimally, to make G legal, only those features that help satisfy Sub should be added. This means that for each $c \in H$ with $f \xrightarrow{?} c$, we need to include in G (1) the feature f, (2) the set Anc_f of all its ancestors, (3) the mandatory descendants of f, noted $Desc_f^{man}$, and (4) the mandatory descendants of the features in Anc_f. Because the resulting feature set should be legal wrt H, those features do not require more changes. We now define this formally.

In what follows, we will use the notation $f \xrightarrow{man+} f'$ to mean $\exists f_1, f_2 \ldots f_n \cdot f \xrightarrow{man} f_1 \wedge f_1 \xrightarrow{man} f_2 \wedge \ldots \wedge f_{n-1} \xrightarrow{man} f_n \wedge f_n \xrightarrow{man} f'$, and the notation $f \xrightarrow{?+} f'$ to mean $\exists f_1, f_2 \ldots f_n \cdot f \xrightarrow{?} f_1 \wedge f_1 \xrightarrow{?} f_2 \wedge \ldots \wedge f_{n-1} \xrightarrow{?} f_n \wedge f_n \xrightarrow{?} f'$.

If we define $Anc_f = \{f' \mid f' \xrightarrow{?+} f\}$ and $Desc_f^{man} = \{f' \mid f \xrightarrow{man+} f'\}$, then the unique, legal and minimal feature set G_2 can be constructed as follows:

- $G_0 = \{f \mid f \xrightarrow{?} c \wedge c \in H\}$
- $G_1 = G_0 \cup \bigcup_{f \in G_0} Anc_f$
- $G_2 = G_1 \cup \bigcup_{f \in G_1} Desc_f^{man}$

□

Proof of Theorem 12. Starting from G_2 built according to the previous proof, we can now possibly add (1) root features that were not already in G_2 plus some of their descendants, and (2) optional descendants of features already in G_2. If we want to keep a change composition that is legal wrt H, the selected additional features cannot require to add any change that is not already in H. Those requirements are fulfilled by the algorithms 2 and 3 which provide a unique legal and maximal set G_3. □

Input: A minimal feature set G_2
Output: A maximal feature set G_3

$candidates \leftarrow roots \setminus G_2 \cup \{f \mid f' \overset{opt}{\rightarrow} f \wedge f' \in G_2\}$;
$added \leftarrow addCandidates(candidates)$;
$G_3 \leftarrow G_2$;
while $added \neq \emptyset$ **do**
  $G_3 \leftarrow G_3 \cup added$;
  $candidates \leftarrow \{f \mid f' \overset{opt}{\rightarrow} f \wedge f' \in added\}$;
  $added \leftarrow addCandidates(candidates)$;
end

Algorithm 2: Constructing a maximal feature set

$added \leftarrow \emptyset$;
foreach $f \in candidates$ **do**
  if $\forall f' \in Desc_f^{man} \cdot \exists c \in C \cdot f' \overset{man}{\rightarrow} c$ **then**
    $added \leftarrow added \cup \{f\} \cup Desc_f^{man}$
  end
  return $added$;
end

Algorithm 3: $addCandidates$ subroutine