Intensional changes

Modularizing crosscutting features

Peter Ebraert, Theo D'Hondt
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
{pebraert, tjdhondt}@vub.ac.be

Tim Molderez, Dirk Janssens
Universiteit Antwerpen
Middelheimlaan 1
2020 Antwerpen
{tim.molderez,dirk.janssens}@ua.ac.be

ABSTRACT

Feature-oriented programming (FOP) targets the encapsulation of software building blocks as features which better match the specification of requirements. As a result, programmers find it easier to design and compose different variations of their systems. Change-based FOP (CFOP) proposes to specify features as sets of first-class change objects which can add, modify or delete building blocks to or from a software system.

First, we show how CFOP supports the modularization of crosscutting functionality. Afterwards, we expose a weakness of CFOP which is a consequence from features holding *extensional* sets of changes. We elaborate on a solution for that weakness which is based on *intensional* changes: descriptions that can evaluate to an extension of changes.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: object-oriented programming, feature-oriented programming; D.2 [**Software Engineering**]: Reusable Software—*coding techniques, software verification, maintenance*

Keywords

Separation of crosscutting concerns, feature-oriented and change-oriented programming

1. CROSSCUTTING CONCERNS

Some functionality of system implementation, such as logging or error handling are notoriously difficult to implement in a modular way. The result is that code implementing such functionality is scattered over and tangled across a system. This leads to quality, productivity and maintenance problems. A functionality is said to be *crosscutting* if it cannot be cleanly separated into a separate module because it is affecting several other modules [8].

Feature-Oriented Programming (FOP) is a software development technique that can be used to modularize crosscutting functionality [14]. In FOP, software is modularized in features: modules which each each add one functionality to the system. The idea of FOP is to support the production of software variations by composing features.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

2. CHANGE-BASED FOP

In change-based FOP (CFOP) [5, 6], features are specified by a set of *changes* that have to be applied in order to implement the functionality the features provide. *Changes* model the operations (*addition*, *modification* and *deletion*) on all kinds of software building blocks (for example classes, methods, fields and statements). *Changes* are instrumented with explicit dependencies which provide information about the validity of different feature compositions. In [7], *flexible* features were introduced as an appropriate concept for modeling crosscutting features. A *flexible* feature is specified by an *extensional* set of changes that does not have to be applied as a whole in order to add the feature to a composition. An evaluation of CFOP and its flexible features showed that flexible features provide more flexibility to composing different variations of a software system [7].

In this paper, we expose a weakness of CFOP which is a consequence from features only being specified by *extensional* sets of changes. We elaborate on a solution for that weakness which requires a new kind of change: An *intensional* change is a descriptive change that can evaluate to an extension of changes.

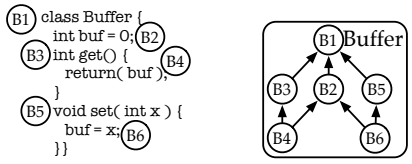

Figure 1: Buffer source code and change objects

Figure 1 shows the source code (on the left) and the changes (on the right) of a `Buffer`. The change objects are identified by a unique number: `B1` is a change that adds a class `Buffer`, `B4` is a change that adds an access of the instance variable `buf`. In CFOP, the dependencies between change objects are also maintained: `B4` depends on the change that adds the method to which `buf` is added (`B3`) and on the change that adds the instance variable that it accesses (`B2`). All changes on which a change `c` depends on are called the *parents* of `c`.

We extend the buffer with a `Restore`, a `Logging` and a

`Stats` feature which respectively add the functionalities of restoring the value of the buffer, logging the values of all instance variables whenever a method of the buffer is executed and maintaining statistics about the buffer. Figure 2 presents the code of the adapted application. From left to right, the features `Restore`, `Logging` and `Stats` are implemented. The corresponding change objects are presented just below.

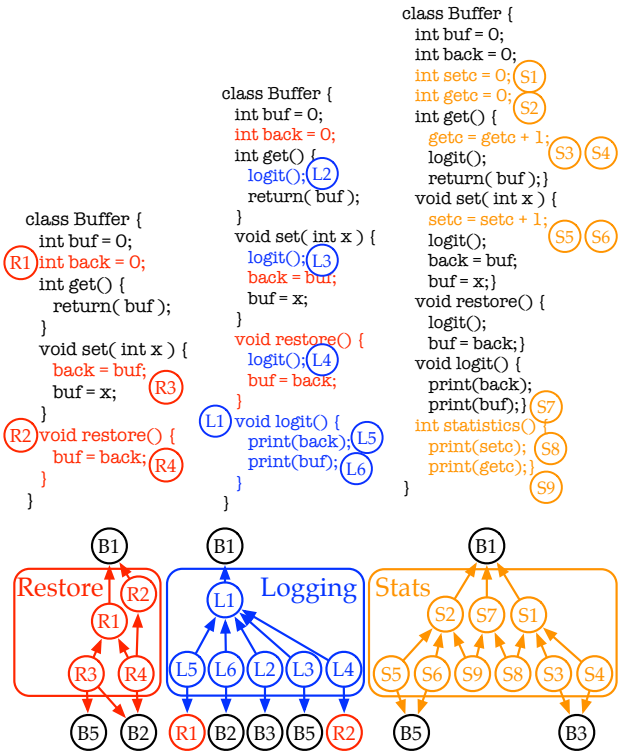

Figure 2: Source code and change objects of Restore, Logging and Stats

2.1 Feature composition

Feature composition is the mechanism that allows the creation of a software variation based on the corresponding required functionalities. In CFOP, a composition `C` is valid if all parent changes of the change objects of `C` are part of `C`. Hence, an invalid composition is the result of composing features that contain a change of which at least one parent change does not reside in the composition. Consider a composition of $Buffer$ and $Logging$. This can be produced by applying the changes of `Buffer` ($B1$,$B2$,$B3$,$B4$,$B5$ and $B6$) and the changes of `Logging` ($L1$,$L2$,$L3$,$L4$,$L5$ and $L6$). The left part of Figure 3 shows that such composition would be invalid, as the parents of $L3$ and $L5$ are not present in the composition. As `Logging` is a flexible feature – denoted by the dashed line surrounding its change objects in Figure 2 – not all its changes need to be included in a composition to make it valid. Consequently, $L3$ and $L5$ can be omitted from the composition in order to make it valid (the right part of Figure 3).

In CFOP, feature composition is similar to static weaving as exhibited by several AOP approaches [11, 10]. The difference is that change application operates on change objects, whereas static weaving typically operates on (byte) code. In [7], we explain that the state-of-the-art approaches to FOP (E.g. Mixin-layers [16], AHEAD [2], Lifting Functions [14], Composition Filters [3], FeatureC++ [1], AOP approaches [11, 15, 10]) lack expressiveness and hinder the reusability of feature modules. Moreover, CFOP is the sole technique that stimulates a bottom-up approach to software development, where modularization does not necessarily need to be determined up front, but modules may be defined as sets of change objects at any point.

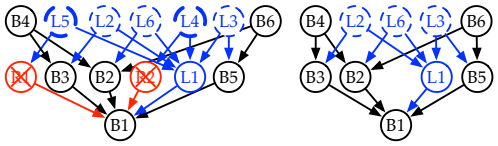

Figure 3: Composition of flexible features

2.2 Extensional changes

While flexible features already help in modeling crosscutting functionality, they still suffer from an inconvenience with respect to their maintenance. That is a consequence from them being specified by an extensional list of changes. In the remainder of this paper, we elaborate on this issue.

Consider the composition of the buffer that has all the available functionality. The `Stats` feature introduces two new instance variables (`setc` and `getc`) and adds a new method (`Stats`) to the `Buffer` class. This behaviour, however, is never logged by the crosscutting `Logging` feature, which is supposed to print the values off all the instance variables of class `Buffer` whenever a method of `Buffer` is invoked. This is a consequence from the extensional nature of the `Logging` specification. Instead of being specified by means of quantification, such as "print the values of *all* instance variable on the invocation of *any* method of class `Buffer`", it is specified by the finite set $L1, L2, L3, L4, L5, L6$ that introduces the logging behavior for buffer variations with or without `Restore` but not for variations including `Stats`. In order to overcome this issue, we propose a new kind of changes – *intensional changes*. The following section elaborates on this novel kind of change.

3. INTENSIONAL CHANGES

Our model defines a feature as a set of changes. There are two ways of specifying such a set: *extensionally* or *intensionally*. A set can be defined extensionally by explicitly enumerating all elements of the set. For example, we can define the set E of all even numbers extensionally as follows:

$$E = \{0, 2, 4, 6, 8, ...\}$$

However, the same set can also be specified intensionally by means of a description:

$$E = \{2x \mid x \in N\}$$

The same applies for sets of changes. The `Logging` feature can be specified by an extensional set of changes $F_{Logging}$ which can be applied on any variation of `Buffer` in order to add the logging functionality:

$$F_{Logging} = \{L1, L2, L3, L4, L5, L6, L7, L8, L9\}$$

An intensional description of the same `Logging` feature that adds logging to a class C could be: $F_{Logging} =$

{Add *logit* method to C}∪
{add statement print(v) in *logit* for every inst.var. v in C}∪
{add an invocation to *logit* in every method of C}

In order to be able to express a change set C in an intensional way, two things are required: a *change-cut language* and a *change-cut model*. A *change-cut language* is a language that can be used to specify change-cuts: a quantification or query that evaluates to an enumeration of changes. A *change-cut model* is a model that defines a change-cut language and a means of specifying the actions for a change cut. The following describes the language and the model.

3.1 Change-cut language

When an intensional change c needs to be applied to a program p in order to add to p the functionality c implements, the specification of c has to be evaluated to find out the exact changes that are required in the specific situation (the enumeration of changes described by c). In the above examples, we used natural language to describe the intensional specification of a change. It is clear that, in order to automate the evaluation process, a more formal language is needed. In this section we establish such a language based on a tuple calculus. We start by explaining that the building blocks of our approach consist of tuples of changes.

3.1.1 Building blocks

The population of a change set is represented by means of n-tuples. Each $n-tuple$ consists of n artifacts which belong together. For instance, an example of a 2-tuple is:

$$(\text{“}B1\text{”},\ 112)$$

These tuples represent for example an association of the id of a change with the timestamp of that change. In classic set-theory, the order in which the artifacts appear in a tuple is important. For instance, the tuple (“$B1$”, 112) is different from the tuple (112, “$B1$”). In order to improve readability of the tuples, we opt to use *named* n-tuples which contain tags mapping a certain attribute of a tuple to a value, similar to the way the population of databases are often described. In this notation, we express our example 2-tuple as follows:

$$(\textbf{id:}\text{“}B1''\text{, }\textbf{timestamp:}112)$$

where *id* and *timestamp* are the *attributes* of the tuple and $B1$ and 112 are the *values* of that tuple. Using this notation, the order of the artifacts in the tuples is no longer important. For instance, the tuples (**id:**$B1$, **timestamp:**112) and (**timestamp:**112, **id:**$B1$) are considered to be identical.

We define C_p to be the set of all changes that produce a program p whenever they are applied. The changes in this set are 6-tuples that follow the following pattern:

$$(id, type, parameterList, timestamp, user, intent)$$

where *id* is the attribute specifying the unique identifier of the change object, *type* is the attribute denoting the kind of the change, *parameterList* is the attribute containing a list of parameters specifying the change, *timestamp* is the attribute that specifies the time at which this change is instantiated, *user* is the attribute that contains the information of which user instantiated this change and *intent* is the attribute that describes the intent of the change.

3.1.2 Atoms

We assume a finite set C_p of tuples and an infinite set of tuple attributes $attrib(C_p)$, for the construction of formulas on the set of tuples. $Changetypes$ is the set of all different types of changes (as described in [5]). We then define the set of atomic formulas $A[C_p]$with the following rules:

1. if c_1 and c_2 in C_p, a and b in $attrib(C_p)$ then the formula “$c_1.a = c_2.b$” is in $A[C_p]$,
2. if c in C_p, a in $attrib(C_p)$ and k denotes a value then the formula “$c.a = k$” is in $A[C_p]$, and
3. if c in C_p and r in $Changetypes$ then the formula “$r(c)$” is in $A[C_p]$.

Examples of atoms include:

$$(c_1.user = c_2.user)$$
$$(c_1.user = \text{“}Gul'')$$
$$AddMethod(c)$$

The first example means that tuple c_1 has a “user” attribute and c_2 has a “user” attribute with the same value. The second example means that tuple c_1 has a “user” attribute and its value is “Gul”. The last example means that tuple c is of the $AddMethod$ type. The formal semantics of such atoms is defined given a change set C_p and a tuple attribute binding $val : C_p \times attrib(C_p) \rightarrow Object$ that maps tuple attributes to tuple values over the domain in C_p:

1. “$c_1.a = c_2.b$” is true iff $val(c_1, a) = val(c_2, b)$
2. “$c.a = k$” is true iff $val(c, a) = k$
3. “$r(c)$” is true iff the type atrribute of c is r.

Atomic formulas denote a condition for a change object. The next step in defining the change cut language is the construction of more complex formulas which consist of *composed* and *quantified* atomic formulas as we explain below.

3.1.3 Formulas

In our change cut language, the atoms can be combined into formulas, with the logical operators ∧ (and), ∨ (or) and ¬ (not), and we can use the existential quantifier (∃) and the universal quantifier (∀) to quantify variables or relations. We define the complete set of formulas $F[C_p]$ inductively with the following rules:

1. every atom in $A[C_p]$ is also in $F[C_p]$,
2. if f_1 and f_2 are in $F[C_p]$ then the formula “$f_1 \wedge f_2$” is also in $F[C_p]$,
3. if f_1 and f_2 are in $F[C_p]$ then the formula “$f_1 \vee f_2$” is also in $F[C_p]$,
4. if f is in $F[C_p]$ then the formula “$\neg f$” is also in $F[C_p]$,
5. if c in C_p and f a formula in $F[C_p]$ then the formula “$\exists c : f$” is also in $F[C_p]$,
6. if c in C_p and f a formula in $F[C_p]$ then the formula “$\exists! c : f$” is also in $F[C_p]$,
7. if c in C_p and f a formula in $F[C_p]$ then the formula “$\nexists c : f$” is also in $F[C_p]$, and
8. if c in C_p and f a formula in $F[C_p]$ then the formula “$\forall c : f$” is also in $F[C_p]$.

Examples of formulas are:

$$(c.user = \text{“}Gul\text{”} \vee c.user = \text{“}Sabine\text{”})$$
$$(AddMethod(c) \wedge c.user = \text{“}Gul\text{”}$$
$$\forall c : (AddMethod(c) \wedge c.user = \text{“}Gul\text{”} \wedge p.intent = \text{“}Buffer\text{”})$$

The first example binds c to all changes of C_p instantiated by Gul or Sabine. The second example binds c to all the changes of C_p that are of the $AddMethod$ type that are instantiated by Gul. The final example returns true if all changes of C_p are of the $AddMethod$ type, instantiated by

Gul in an intent to create a buffer. Note that we omit brackets if this does not cause ambiguity about the semantics of the formula. We assume that the quantifiers quantify over the set of all tuples of the change set C_p. This leads to the following formal semantics for formulas:

1. see the formal semantics of the atomic formulas,
2. "$f_1 \wedge f_2$ " is true iff "f_1" is true and "f_2" is true,
3. "$f_1 \vee f_2$" is true iff "f_1" or "f_2" or both are true,
4. "$\neg f$" is true iff "f" is not true,
5. "$\exists c : f$" is true iff there is a tuple c in C_p for which the formula "f" is true,
6. "$\exists! c : f$" is true iff there is exactly one tuple c in C_p for which the formula "f" is true,
7. "$\nexists c : f$" is true iff there is no tuple c in C_p for which formula "f" is true, and
8. "$\forall c : f$" is true iff for all tuples c in C_p "f" is true.

Formulas denote the condition for a change cut and consist of compositions and/or quantifications of formulas. Next, we explain how formulas can be used to define change cuts.

3.1.4 Change cuts

Finally, we define a change cut expression (in short a change cut) for a given change set C_p as: $\{c : f(c)\}$ where c is a tuple in C_p and $f(c)$ a formula in $F[C_p]$. The result of such a query for a given change set C_p that specifies a program p is the set of all tuples c of C_p such that f is true. Examples of change cut expressions include:

$$\{c : \forall c : \left(\begin{array}{l} AddMethod(c) \wedge AddClass(d) \wedge \\ c\ parameterList\ class = d\ parameterList\ class \end{array} \right)\}$$

$$\{c : \forall c : \left(\begin{array}{l} AddAttribute(c) \wedge AddClass(d) \wedge \\ c\ parameterList\ class = d\ parameterList\ class \end{array} \right)\}$$

which respectively return all the changes that add a method to a class and all changes that add an attribute to a class. Now that we are capable of expressing change cuts, all that remains in order to enable expressing intensional changes are the actions that need to be taken for the tuples a change cut specifies. This is the subject of the following paragraph.

3.1.5 Actions

The purpose of intensional changes is to allow a developer to *describe* which changes have to be added (in an intensional way) instead of *enumerating* them (in an extensional way). As an example, we consider again the `Logging` feature which we want to express in an intensional way.

Now that we have established a formal language for expressing change cuts, we only need a way of specifying the actions that need to be taken for all the tuples of a change cut. These actions consist of the addition of a change (or change set) to the change set.

We define an action as the specification of the addition of a new tuple (representing a change instance) to the tuple set C_p. We differentiate between actions that add that tuple just *before* and just *after* a change:

1. c_{new} before c
2. c_{new} after c

where c_{new} is a 6-tuple:

$$(id, type, parameterList, timestamp, user, intent),$$

in which all the attributes except for *id* and *timestamp* must be bound to values. Those values can be the value of another attribute of the action, or a constant. The *id* is assigned to the change tuple when the action is evaluated. It is ensured to be a unique identifier. The *timestamp* of the new change tuples is determined by the keyword (*before* or *after*) and by the *timestamp* of the change c. In case the action specifies the new change to come after c, the new *timestamp* is set to a *timestamp* which is just after the one of c. In case the action specifies the new change to come before c, the new *timestamp* is set to a *timestamp* which is just before the one of c. We elaborate on how the uniqueness of the *id* is ensured and how the timestamps are calculated in Section 3.3. As an example of an action, we present

$$(?id2, AddStatement, (Buffer, false, get, \text{“}logit\text{”}),\\ ?timestamp2, \text{“}Gul\text{”}, \text{“}Logging\text{”})\ \textbf{after}\\ (B3, AddMethod, (Buffer, false, get), 235, \text{“}Gul\text{”}, \text{“}Logging\text{”})$$

which adds a statement to the *get* method of the `Buffer` class just after the method *get* is added to the `Buffer` class. We conclude this subsection with the definition of an intensional change.

3.1.6 Intensional change definition

An intensional change consists of two parts: a set of actions A and a change cut:

$$\{c : A \mid f(c)\}$$

where the values of tuples specified by the change cut $f(c)$ can be used as values for the attributes of tuples in the actions A. We exemplify this by specifying the `Logging` feature by means of intensional changes:

$$\{(?id, AddMethod, (Buffer, false, logit()),\\ ?timestamp, \text{“}Gul\text{”}, \text{“}Logging\text{”})\}\cup\\ \{c : \{(?id, AddStatement, (Buffer, false, logit,\\ \text{“}print(c.parameterList.attribute)\text{”}),\\ ?timestamp, \text{“}Gul\text{”}, \text{“}Logging\text{”})\\ \textbf{after}\ c\} \mid\\ \{\forall c : (AddAttribute(c) \wedge c.parameterList.class = Buffer)\}\}\cup\\ \{c : \{(?id, AddStatement, (Buffer, false, ?m, \text{“}logit()\text{”}),\\ ?timestamp, \text{“}Gul\text{”}, \text{“}Logging\text{”})\\ \textbf{after}\ c\} \mid\\ \{\forall c : (AddMetod(c) \wedge c.method = ?m \wedge\\ c.parameterList.class = Buffer)\}\}$$

The definition consists of three parts which need to be evaluated with respect to a change set in order to produce the extension they describe. The following subsection elaborates on the evaluation of intensional changes.

3.2 Intensional change evaluation

An intensional change is a change that can be applied to a software program p in order to apply the changes described by that intensional change. This application consists of two phases. First, the intensional change needs to be evaluated with respect to the change set that specifies p. This step produces an enumeration of changes which are all applied on p in the second step.

Consider $\{Buffer, Restore, Stats, Logging\}$, a software composition which consists of a buffer with the features `Buffer`, `Restore`, `Stats` and `Logging`. The application of the `Logging` feature as it is specified by means of intensional changes evaluates to the extension of Listing 1 while the application on the same `Logging` feature in a composition $\{Buffer, Stats, Logging\}$ evaluates to the extension of Listing 2. The order in which the features are specified in a composition determines the outcome of the evaluation of intensional changes. This is due to the growing change set on which the intensional changes are evaluated.

```
(L1,AddMethod,(Buffer,false,logit),235465, "Gul", "Logging")
(L7,AddStatement,(Buffer,false,logit,"print(setc)"), 247537.1, "Gul", "Logging")
(L8,AddStatement,(Buffer,false,logit,"print(getc)"), 247737.1, "Gul", "Logging")
(L5,AddStatement,(Buffer,false,logit,"print(back)"), 236537.1, "Gul", "Logging")
(L6,AddStatement,(Buffer,false,logit,"print(buf)"), 235789.1, "Gul", "Logging")
(L2,AddStatement,(Buffer,false,get,"logit()"), 235935.1, "Gul", "Logging")
(L3,AddStatement,(Buffer,false,set,"logit()"), 237537.1, "Gul", "Logging")
(L4,AddStatement,(Buffer,false,restore,"logit()"), 236137.1, "Gul", "Logging")
(L9,AddStatement,(Buffer,false,statistics,"logit()"), 247937.1, "Gul","Logging")
```

Listing 1: `Logging` **extension on** $\{Buffer, Rest., Stats\}$

```
(L1,AddMethod,(Buffer,false,logit), 235465, "Gul", "Logging")
(L5,AddStatement,(Buffer,false,logit,"print(back)"), 236537.1, "Gul", "Logging")
(L6,AddStatement,(Buffer,false,logit,"print(buf)"), 235789.1, "Gul", "Logging")
(L2,AddStatement,(Buffer,false,get,"logit()"), 235935.1, "Gul", "Logging")
(L3,AddStatement,(Buffer,false,set,"logit()"), 237537.1, "Gul", "Logging")
(L4,AddStatement,(Buffer,false,restore, "logit()"),236137.1, "Gul", "Logging")
```

Listing 2: `Logging` **extension on** $\{Buffer, Restore\}$

Listings 1 and 2 present the outcome of the first step of the evaluation process. The second step consists of applying the changes that correspond to the tuples from those listings. In order to do that, a change object is instantiated for every tuple. The first tuple, for instance, results in an *AddMethod* change object which adds the `logit` method to `Buffer` class.

For the sake of clarity, we specified the `Logging` feature in a very naive way as the specification does not consider the possibility that attributes or methods were modified or removed by other change objects in the change set. Such naive specification could produce change objects with subjects that no longer exist in the software product at the time the change is applied. As the composition algorithm takes into account the structural dependencies between the change objects, this never produces products with invalid structure. The semantics, however, can indeed be influenced by this naive specification. It is up to the developer to alter the intensional change in such a situation. We now discuss the implementation of the change cut language and model of intensional changes.

3.3 Implementation

First, we extend the change model of CFOP [5] with the notion of intensional changes. Figure 4 shows the extended change model, in which the `Intensional Change` class is presented as a subclass of `Change`.

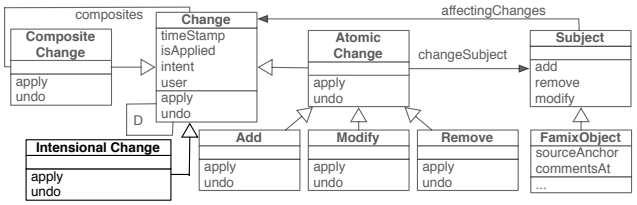

Figure 4: Extended change model

Second, we implement the language we defined for specifying intensional changes. For this implementation we use SOUL, an implementation of a Prolog-like declarative language on top of Smalltalk. Declarative programming allows a developer to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to accomplish it [17]. Another key feature of SOUL is that it can be used to reason about software programs specified in different programming languages. A third key feature of SOUL is that it supports symbiosis with the underlying Smalltalk or Java environment. The symbiosis allows one to write Smalltalk or Java code within the declarative language which gets executed whenever that declaration is asserted. These features make SOUL an ideal candidate for reasoning about and creating change objects, and thus for specifying intensional changes.

Logic languages such as SOUL have been used as a means of implementing pointcut languages in AOP approaches as well [10, 15]. This is no surprise, considering the similar purpose of a pointcut language and a change-cut language, as mentioned earlier.

SOUL consists of a logic kernel together with an underlying library of basic logic primitives for every programming language it can reason about. Two such libraries exist, namely the Library for Code Reasoning (LiCoR) [13] for reasoning about Smalltalk code and Irish [9] for reasoning about Java code. The logic kernel itself consists of a library of predicates that serve for basic reasoning. The predicates in that library can be used by the developers for creating new predicates in SOUL. Predicates can be grouped in layers, for the purpose of classification. We create a basic layer, which contains predicates for every change kind:

$change(?c)$ **if**
$member(?c, [ChangeLoggercurrentComposition\ changes])$

This predicate states that something is a change if it is a member of the collection of changes. Note that this predicate uses the symbiotic properties of SOUL to obtain the collection of changes. The evaluation of the query $change(?c)$ makes sure that every change instance is bound once to $?c$. For every kind of attribute of a change, we include a predicate of the shape:

$user(?u, ?c)$ **if** $change(?c), equals(?u, [$`?c user`$])$

This predicate binds to $?u$ the value of the Smalltalk expression that requests a change $?c$ for its `user` attribute. Now we have explained the building blocks of the language (tuples of change objects), we elaborate on the atoms. An atom is a predicate such as $equals(?u, ?n)$ where $?n$ and ?u can be constants or variables that must have the same value whenever the predicate is evaluated. Formulas are implemented by chaining atoms. A comma represents the $\wedge$, an alternative definition of a predicate represents the $\vee$, the $not()$ represents the $\neg$. As for the $\forall$ and $\exists$ quantifiers, they are implemented by means of standard SOUL predicates $forall(?query, ?test)$ and $exists(?query, ?test)$.

Third, we implement the actions. An action is specified as a SOUL predicate that includes symbiotic Smalltalk code that is executed to instantiate the concerned changes. An example of an action for adding a method before a change $?c$ is presented here:

$AddMethodBefore(?ParameterList, ?user, ?intent, ?c, ?c_{new})$ **if**
$equals(?c_{new}, [$AddChange newMethod: $?ParameterList$
before:$?c$ by:$?user$ for:$?intent$])

A predicate like the one above is defined for each combination of change kind (add, remove or modify), the kind of subject (e.g. class, method, etc) and the time keyword

(before or after). All predicates of this kind are grouped in a separate SOUL layer called "actions". The uniqueness of the id of the generated change is ensured by the Smalltalk virtual machine, which assigns a unique object id to every object that is instantiated. As we use the object ID as the change ID, it is also unique. The timestamp is assigned in such a way that it allows partially ordering the changes based on the timestamp.

Finally, we present the implementation of an intensional change and how its evaluation is implemented. An intensional change is specified by a SOUL query that includes a set of predicates (representing the actions) and another set of predicates (representing the change cut). Take for instance the following SOUL query:

$$
\begin{array}{l}
AddStatementAfter((Buffer, false, ?m, \text{“}logit()\text{”}),\\
\quad \text{“}Gul\text{”}, \text{“}Logging\text{”}, ?c, ?c_{new}),\\
AddMethod(?c),\\
method(?m, ?c),\\
class(\texttt{Buffer}, ?c)
\end{array}
$$

the first predicate of which is an action and the three final predicates of which represent the change cut. This query represents the intensional change of the `Logging` feature. It creates an AddStatement change with a timestamp greater than the one of $?c$, for each change $?c$ that adds a method to the `Buffer` class.

Whenever an intensional change instance needs to be applied, its *apply* method needs to be called. That method is implemented by a call to the SOUL engine that executes the query of that change. This causes an enumeration of the described change set to be produced and added to the change set.

We have implemented both the change cut language and the change cut model by means of SOUL. We integrated both in ChEOPS – a proof-of-concept implementation [6]. Some small experiments were conducted to validate the usability of intensional changes in the context of FOP. The following section elaborates on those experiments.

4. FOTEXT

The goal of this section is to demonstrate that CFOP supports the modularization of crosscutting functionality and that intensional changes make such modules even more reusable in different compositions.

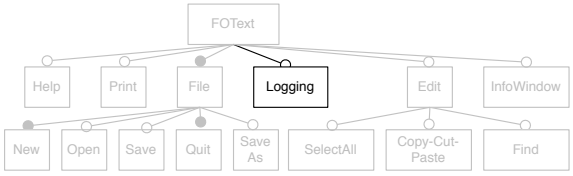

Figure 5: FODA diagram of FOText

Figure 5 presents the FODA diagram of the `FOText` application. In its original form [12], the `FOText` application contains only the grey features, on which we do not elaborate here. We extend `FOText` with the `Logging` feature, in order to demonstrate the power of flexible features and intensional changes. The `Logging` feature is a feature that adds logging behavior to the text editor. `Logging` is a crosscutting functionality and involves changes that depend on many `FOText` features. We specify it as flexible while the former fourteen features are specified as monolithic.

We implement `FOText` in a standard object-oriented way by using the VisualWorks for Smalltalk IDE that was instrumented with the ChEOPS tool to capture our development operations as first-class change entities by means of a *logging* technique. At the beginning of the development of a new feature, we inform the IDE of its name and type (*flexible* or *monolithic*). By doing that, ChEOPS is capable of *automatically classifying* changes in features and by keeping track of whether changes are optional or mandatory with respect to their parent feature.

For the sake of simplicity, we introduce an artificial feature, `Base`, which is the basic feature that needs to be included in every `FOText` variation. It consists of the changes that should always be included in a composition: `FOText`, `File`, `New` and `Quit`. It provides the main functionality: a basic word processor that provides a window to type text and a menu with two choices: `new` and `quit` – which are respectively introduced by the `New` and `Quit` features.

4.1 Feature composition

CFOP allows the composition of a program variation by specifying which functionality that variation should include. As in CFOP every functionality is linked to a set of changes – a feature, the specification of a set of functionalities can automatically be translated into a change composition that represents the variation. In order to obtain the variation, the corresponding change composition has to be applied. Some compositions, however, are not applicable due to unsatisfied dependencies. Thanks to the fine-grained level of feature specification, ChEOPS can check whether or not a composition is valid. In case it is not, it can assist in resolving the conflict by presenting the developer with fine-grained information about the conflict.

We now present two compositions in which we include `Logging`; a flexible feature implemented by means of intensional changes. The first composition consists of a variation of the viewer version of `FOText` (which is composed of the monolithic `Base` and `Open` features) and which has logging capabilities. Our tool informs us that this composition is valid (there were no unsatisfied dependencies) and depicts the change composition graph containing 181 change objects representing the `Logging` feature.

Now, let us produce a composition that contains all the features of `FOText` including `Logging`. ChEOPS again informs that this composition is valid and tells us that there is a total of 541 changes representing the `Logging` feature. The reason for the higher number is that it requires more changes to add the `Logging` functionality to a composition that contains more instance variables and methods as `Logging` is supposed to log the values of *all* instance variables whenever *any* method is invoked.

These compositions show how features that include intensional changes evaluate to different change sets depending on the context without having to adapt their specification. This is what one wants when implementing a crosscutting functionality. We conclude that a combination of flexible features and intensional changes can be used to modularize crosscutting functionality in reusable modules without giving up on flexibility when it comes to software composition.

Drawbacks of intensional changes are threefold. First they require an additional change cut language and evaluation

step, making them somewhat *more complex* than ordinary changes. They typically do not grow more complex, however, when systems grow bigger. Moreover, progress has recently been made with regards to rending query languages more accessible such that developers can apply these technologies to daily software engineering problems [4]. A second drawback is that intensional changes *complicate debugging*, as they evaluate differently in different compositions. A visualization of the impact of intensional changes might assist the developer when debugging intensional changes. Finally, while the *order* of the features within a composition was not important before the intensional changes were included in the model, it now has become important as it influences the way the intensional change is evaluated. This is also the case for AOP approaches [10, 11]

5. CONCLUSION

Some functionality, such as logging, is notoriously difficult to implement in a modular way. Such *crosscutting* functionality usually affects software building blocks scattered over the system. We show that in change-based feature-oriented programming (CFOP), the implementation of crosscutting functionality is actually not scattered over the software application because the changes are grouped in one change set.

While CFOP enables the modularization of crosscutting functionality, it still suffers from an inconvenience with respect to their maintenance. That is a consequence from the modules being specified by an extensional list of changes. In order to overcome this, we introduce the *intensional change*: a kind of change that can be applied on a software program p. This application consists of two phases. First, the intensional change is evaluated with respect to the change set that specifies p. That produces an enumeration of changes which is applied on p in the second step.

We present a formal language that can be used to specify intensional changes. It is based on a tuple calculus and implemented by means of SOUL, an implementation of a Prolog-like declarative language on top of Smalltalk. Finally, we evaluate intensional changes and present the advantages and drawbacks. Benefits include that intensional changes make features more reusable and feature composition more flexible. Drawbacks include an increase in complexity of the change specification and debugging process and the fact that the order in which the features are specified within a composition became important after the introduction of intensional changes, as those changes are evaluated with respect to a change set.

6. REFERENCES

[1] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

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

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, 2001.

[4] C. De Roover, J. Brichau, C. Noguera, T. D'Hondt, and L. Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM07)*, pages 92–102. ACM Press, 2007.

[5] P. Ebraert. First-class change objects for feature-oriented programming. In *15th Working Conference on Reverse Engineering proceedings*, pages 319–323. IEEE Computer Society, 2008.

[6] P. Ebraert and T. D'Hondt. Feature-oriented programming based on first-class changes. In *Proceedings of the 2nd Workshop on FAMIX and Moose in Reengineering*. IEEE Computer Society, 2008.

[7] P. Ebraert, J. Vallejos, Y. Vandewoude, Y. Berbers, and T. D'Hondt. Flexible features: Making feature modules more reusable. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing*, 2009.

[8] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001.

[9] J. Fabry and T. Mens. Language independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures,*, 30(1-2):21–33, April 2004.

[10] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectJ. In *Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science,*, volume 2072, pages 327 – 353. Springer Verlag, 2001. http://aspectj.org.

[12] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented software designs. In S. Reiff-Marganiec and M. Ryan, editors, *FIW*, pages 178–197. IOS Press, 2005.

[13] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 23(4):405–413, 2002.

[14] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241:419–434, 1997.

[15] T. W. (Rho). Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.

[16] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[17] R. Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.