Flexible features

Making feature modules more reusable

Peter Ebraert[*]
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
pebraert@vub.ac.be

Jorge Vallejos[†]
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
jvallejo@vub.ac.be

Yves Vandewoude
Catholic University of Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
yves@stcham.org

ABSTRACT

A growing trend in software construction advocates 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. Feature-oriented programming (FOP) is the research domain that targets this trend. We argue that the state-of-the-art techniques for FOP have shortcomings because they specify a feature as a set of building blocks rather than a transition that has to be applied on a software system in order to add that feature's functionality to the system.

We propose to specify features as sets of first-class change objects which can add, modify or delete building blocks to or from a software system. We evaluate this approach by implementing a simple text editor in a feature-oriented way and use the implementation to produce four different program variations. This shows that our approach contributes to FOP on three levels: expressiveness, composition verification and bottom-up feature-oriented development.

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*

General Terms

Algorithms, management, design, verification

[*]Research funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen .

[†]Research funded by the Varibru research project initiated in the framework of the Brussels Impulse Programme for ICT supported by the Brussels Capital Region.

Keywords

Separation of crosscutting concerns, Feature-Oriented Programming, Change-Oriented Software Engineering

1. SOFTWARE SYSTEM VARIATIONS

A software product family is a set of variations of the same software system. Developing all variations of a software product family can be addressed in an ad-hoc way by implementing one big system that contains all possible variations and which behaves differently depending on its configuration. In a procedural or functional programming style, the resulting code, however, would contain an *IF-THEN* control statement at every place where the program chooses which variant to produce. This kind of implementation lacks *modularity* and *reusability* [13]. An object-oriented implementation would use *polymorphism* to implement the variation points. Then, each *IF-THEN* control statement is replaced by instantiating different subclasses of a class which are modeling the specification of each variation. This approach would still require a significant amount of manual labor [7]. The most important drawback of such approach, however, is that it suffers from a combinatorial explosion [11], as for every new funtionality, the number of variations is multiplied by two.

A better alternative is to modularize a software system based on the functionalities it provides. Modules which add a functionality to the system are called *features*. *Feature-Oriented Programming* (FOP) is the discipline that centralises features as the main development module. The idea of FOP is to produce software variations by composing features. We find that the state-of-the-art approaches to FOP (E.g. Mixin-layers [17], AHEAD [2], Lifting Functions [14], Composition Filters [3], FeatureC++ [1] and AOP approaches [10]) lack expressiveness and hinder the reusability of feature modules.

None of the state-of-the-art approaches allow features to express the *deletion* of software building blocks. Later in this paper, we present an example where that is required. The *granularity* the appraoches provide rarely reaches the statement level and in case it does, it is limited. AHEAD, Lifting Functions, Mixin-layers, FeatureC++ and most AOP approaches do allow the specification of a feature that adds a statement to an existing method by means of the `super` call. This construct, however, only allows the expression of a statement addition before and/or after the complete old method behavior. With those approaches, it is not possible to specify a feature that adds a statement between the statements of an existing method. With the exception of FeatureC++ and the AOP approaches, none of the above techniques provide means to specify such features: they require an alternative implementation of the crosscutting feature depending on the features present in the composition, hindering reusability. None of the approaches allow a *bottom-up approach* to FOP that does *not require specific language support*. We believe such an approach to FOP might increase its usability in an industrial context, where companies typically do not want to alleviate from their development methodologies and exerted programming language.

We believe that selecting appropriates building blocks to specify features does allow to overcome these issues of expressiveness and reusability. In the following two sections, we propose a change-based model for FOP that does so.

2. CHANGE-BASED FOP

Other researchers pointed out the use of encapsulating change as first-class entities. In [15] Robbes shows that the information from the change objects provides a lot more information about the evolution of a software system than the central code repositories. In [4] Denker shows that first-class changes can be used to define a scope for dynamic execution and that they can consequently be used to adapt running software systems. In this section, we first explain a model of first-class changes and then show how to specify and compose features as sets of first-class changes.

2.1 Change-Oriented Software Engineering

Change-Oriented Software Engineering (COSE) was first introduced in [6]. It centralizes *change* as the main development entity. In COSE, all operations a programmer performs while making a software system are captured in change objects. We illustrate COSE by an example: a `Buffer` base program that follows the *value object* pattern [8].

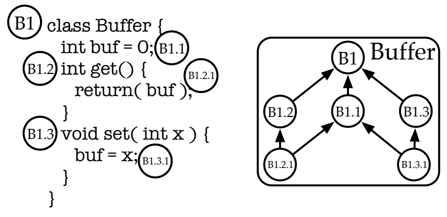
```
class Buffer {
  int buf = 0;
  int get() {
    return( buf );
  }
  void set( int x ) {
    buf = x;
  }
}
```

Figure 1: Source code (left) and change objects (right) of the Buffer

Figure 1 shows the Java code of the `Buffer` feature. It does not show what operations were performed in order to produce this code. Actually, first the `Buffer` class was added (B1). Afterwards, an instance variable called `buf` was added (B1.1). Finally, the methods `get` (B1.2) with body (B1.2.1) and `set` (B1.3) with body (B1.3.1) were added. In COSE, these development operations are captured as first-class change objects which are depicted in the right part of Figure 1.

Afterwards, we extend the application with a `Restore`, a `Logging` and a `Multiple Restore` 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 allowing the buffer to restore more than one value. Figure 2 presents the code of the adapted application. From left to right, the features Restore, Logging and Multiple Restore are implemented. The corresponding change objects are presented in Figure 3.

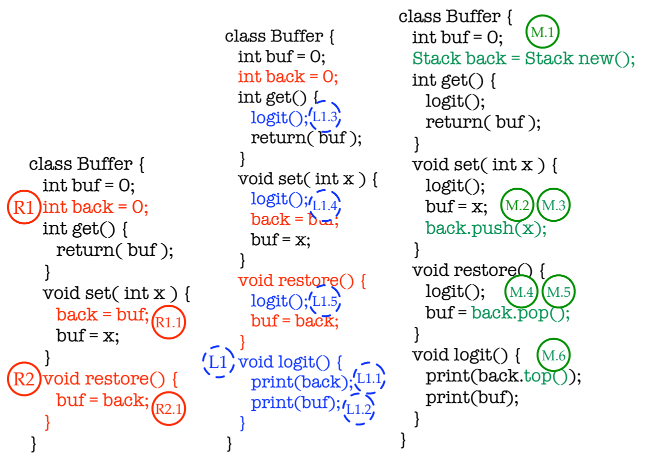
```
class Buffer {
  int buf = 0;
  int back = 0;
  int get() {
    return( buf );
  }
  void set( int x ) {
    back = buf;
    buf = x;
  }
  void restore() {
    buf = back;
  }
}
```
```
class Buffer {
  int buf = 0;
  int back = 0;
  int get() {
    logit();
    return( buf );
  }
  void set( int x ) {
    logit();
    back = buf;
    buf = x;
  }
  void restore() {
    logit();
    buf = back;
  }
  void logit() {
    print(back);
    print(buf);
  }
}
```
```
class Buffer {
  int buf = 0;
  Stack back = Stack new();
  int get() {
    logit();
    return( buf );
  }
  void set( int x ) {
    logit();
    buf = x;
    back.push(x);
  }
  void restore() {
    logit();
    buf = back.pop();
  }
  void logit() {
    print(back.top());
    print(buf);
  }
}
```

Figure 2: Source code of adding Restore (left), Logging (middle), Multiple Restore (right)

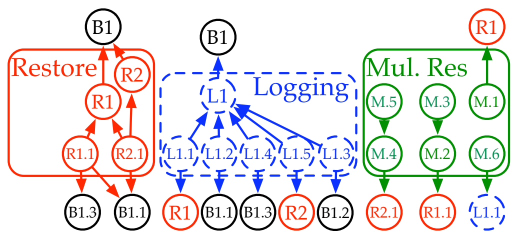

Figure 3: Changes of Restore (left), Logging (middle) and Multiple Restore (right)

2.2 Explicit dependency Management

Some features depend on others in order to be able to provide the functionality they implement. FODA diagrams [9] (E.g; Figure 5) provide information about feature interactions and express such dependencies. The dependency relation is a binary relation that is transitive, anti-symmetric and reflexive. COSE's change objects are also related by such a dependency relationship. All changes on which a change `c` depends on are called the *parents* of `c`. There are two types of change dependencies: *syntactic* and *semantic* dependencies [16].

Syntactic dependencies are enforced by the meta-model of the programming language. We find that programming language entities are related by the abstract syntax tree. We call the dependencies that arise from this relation *hierarchical* dependencies. We also distinguish *invocative* dependencies: dependencies between the change that adds a method invocation statement and the change that adds the invoked method. An *accessive* dependency is found between a change that adds an instance variable and the change that adds an access to that instance variable. Finally, a *creational* dependency exists between a change that removes or modifies an entity and the change that added the entity, since an entity can be removed only if it exists.

Consequently, a syntactic dependency is a dependency that is needed to ensure the compilation of a program. Examples of a syntactic dependency are: a change that adds a method *depends on* the change that created the class where the method is added, or the change that adds an invocation to a method *depends on* the change that added the method that is invoked.

Semantic dependencies come from the domain knowledge. Hence, the developer is responsable to establish these dependencies. One possible semantic dependency is the common intention of changes like the implementation of a feature. A semantic dependency is a relation where the dependent entity does not exhibit the desired behavior whenever the entity on which it depends is not present in the expected form. An example of a semantic dependence is when the addition of an invocation to method `m` only exhibits correct behavior if the body of method `m` is modified in a specific way.

The syntactic dependencies between change objects are included in the change model and presented by the arrows in figures 1 and 3. The semantic dependencies are denoted by grouping the change objects by their common intention: the feature they implement (denoted by the line surrounding the change objects).

2.3 Changes as feature building blocks

In our model, the change objects that represent the development operations that were carried out to implement one feature are grouped in a set of changes that represent that feature. The dependencies between the change objects are maintained within the change objects themselves to sensure that the changes are aware of and can be queried for their dependencies. Figures 1 and 3 present a view of the changes of the complete `Buffer` application.

Dependencies are not confined within a single feature, but can reach changes of other features as well. For example, in Figure 3, changes within the `Restore` feature depend on changes inside the `Base` feature which explicitely states that the `Restore` feature depends on `Base`. Hence, feature dependencies can be modeled by means of dependencies between changes. Note that a well-modularized program would contain only few cross-feature dependencies. The number of cross-feature dependencies might provide a representative metric to measure the coupling between the features of a software system.

Because the features of the `Buffer` example are incrementally implemented, some features do not only consist of additions but also modifications and deletions of statements, instance variable accesses or methods. For instance, the `Multiple Restore` feature is created by modifying a statement – to initialize `back` with an empty stack – and deleting the statement that assigned `buf` to `back` and adding a statement that invokes the `push` method of that stack. This shows that our model allows the expression of features that add, modify or remove very fine-grained building blocks like statements.

2.4 Feature Composition

The composition of features is the mechanism that allows the creation of a software variation based on the required corresponding functionalities. In our approach, 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. Listing 1 presents an algorithm that verifies the validity of a composition. It receives as input a list containing the features of the required composition and the change objects that specify the implementation of all the features. It returns a list that consist of two lists. The `successful` list contains the changes in the order in which they must be applied to produce the required composition. The `error` list consists of the changes that had to be omitted from the composition because of unsatisfied dependencies.

```
validateComposition: features
features do: [:feature |
  feature getChanges do: [:change |
    (change isIndependent
      || change allDependenciesSatisfied)
      ifTrue: [successful add: change]
      ifFalse:[self depthFirstStrategy:change]
  ]
]
^List with:successful with:error

depthFirstStrategy: aChange
aChange changesOnWhichIDepend do: [:parent |
 (successful includes: parent)
  ifFalse: [(parent feature = aChange feature)
    ifTrue: [parent isIndependent
     ifTrue: [successful add: parent]
     ifFalse: [depthFirstStrategy: parent]]
    ifFalse: [error add: aChange]]
]
successful includesAll:
 (aChange changesOnWhichIDepend)
  ifTrue: [successful add: aChange]
```

Listing 1: Feature composition algorithm

In the best case, the order of the time complexity of this algorithm is $O(n)$, where n is the number of changes in the composition. It occurs when all changes do not have dependencies and can be added to the list directly without calling the recursive method. In the worst case the order is $O(n * (n + e))$ which is the result of applying n times a depth-first search in a graph with n nodes and e edges. On average, the order is $O(n^2)$ since the number of edges and nodes are usually from the same order of magnitude.

The presented algorithm is naive and returns different results depending on the order in which the required features are specified in the `features` parameter. In fact, it only produces a valid composition if the dependencies of the changes of each feature are satisfied by the changes of its predecessors in the `features` parameter. This limitation is not only a technical issue, but is also a consequence from the way features are specified in our model. We come back to this issue in Section 5.

3. FLEXIBLE FEATURES

In our model a *crosscutting* functionality is implemented by a feature that introduces changes that depend on changes of more than one feature. A composition in which a feature is added without one of its dependent features is immediately rejected. A quick and dirty solution would be to provide a feature variation for each combination of the features on which it depends. This kind of solution would suffer from a combinatorial explosion, increase the coupling between features and decrease reusability. We advocate another solution which allows features to be partially deployed in a composition. A feature that implements a crosscutting functionality can be implemented as a set of changes that does not have to be applied as a whole for a composition to be valid. In contrast to a feature that has to be applied as a whole (*monolithic*) we call features that can be partially applied *flexible* features. When a flexible feature is deployed in a composition, the composition algorithm should decide which changes should be included to and which omitted from the composition.

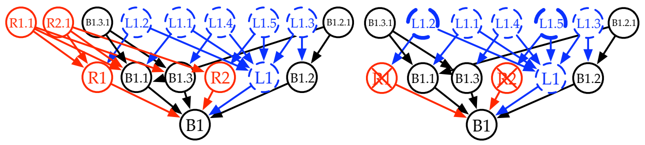

Figure 4: Compositions based on first-class changes

An example of such feature is the `Logging` feature in the `Buffer` example (denoted by the dashed line surrounding the change objects in Figure 4). The `Logging` feature consists of the addition of a method which implements its main functionality and several invocations that added to methods introduced by other features. We argue that in such a case, although the changes that add such invocations depend on the changes that added the methods to which the former change add their invocations, we should be able to omit the former change from the composition to make it valid. Figure 4 shows an example of a valid composition of `Base` and `Logging` (on the right). Since in this composition, the `Restore` feature is not present, the changes `L1.2` and `L1.5` from the `Logging` feature that depend on changes `R1` and `R.2` are omitted, allowing to produce a valid composition.

Use of *flexible* features is not confined to describing crosscutting functionality. For instance, a feature that implements a *facade* pattern [8], would add a class and a method for each complex service. It can be conveniently described by a *flexible* feature allowing to be composed with a set of features which not necessarily include all the services that the *facade* class references. In a composition, the *facade* class will provide only the methods which functionality is indeed present in the composition.

Specifying a feature as *flexible* has to do with the semantics of the feature and must be done manually by the developer. If a developer classifies a feature as *flexible*, this feature will be able to be included in all compositions. Composing a feature that was erroneously specified as *flexible* would yield a useless program. Consequently, programmers should understand the responsibility that comes with the power of flexible features. Note that the compilation of the resulting program can still be ensured by the syntactic dependencies between the change objects as we explain below.

3.1 Compositions of Flexible features

In order to incorporate *flexible* features in our composition model, the composition algorithm of Listing 1 is adapted. The `validateComposition` method is provided with an extra list for storing the changes that are omitted when deploying a *flexible* feature that contains changes with unsatisfied dependencies. The `depthFirstStrategy` method is also adapted in such a way that, whenever a change of a flexible feature has an unsatisfied dependency, it is omitted from the `successful` list and added to the `warning` list. Listing 2 presents the improved composition algorithm that can be used to compose flexible features. It has the same order of time complexity as the previous version.

```
validateComposition: features
features do: [:feature |
  feature getChanges do: [:change |
    (change isIndependent
      || change allDependenciesSatisfied)
      ifTrue: [successful add: change]
      ifFalse:[self depthFirstStrategy:change]
  ]
]
^List with:successful with:error with:warning

depthFirstStrategy: aChange
aChange changesOnWhichIDepend do:[:parent |
 (successful includes: parent)
  ifFalse: [(parent feature = aChange feature)
    ifTrue: [parent isIndependent
      ifTrue: [successful add: parent]
      ifFalse: [depthFirstStrategy: parent]]
    ifFalse: [aChange feature = #Flexible
      ifTrue: [warning add: aChange]
      ifFalse: [error add: aChange]
    ]]
].
successful includesAll:
 (aChange changesOnWhichIDepend)
  ifTrue: [successful add: aChange].
 error size > 0
  ifTrue: [warning removeAll]
```

Listing 2: Improved feature composition algorithm

4. EVALUATION

In this section, we evaluate our model by implementing `FOText`: a Feature-oriented implementation of a *word processor*. We expect our model to allow the expression of additions, modifications and deletions of building blocks up to the level of a single statement. We expect our model to allow a customised deployment of flexible features without braking the validity of a composition. Finally, we assess that our approach to FOP indeed does not require additional language support and as such allows bottom-up FOP development.

4.1 FOText design

The FOText application provides a graphical interface in which users may type and edit texts. It also provides a menu – launched by the right mouse button – that allows the execution of the editing functions that are provided by FOText. FOText adheres to the *Model-View-Controller* design pattern [8].

Figure 5 presents the different features of the FOText application and the relations amongst them. Features such as: `New`, `Open`, `Save`, `SaveAs`, `Print`, `Copy-Cut-Paste`, `Find`, `SelectAll` and `Help` are self explaining. The `Compress` fea-

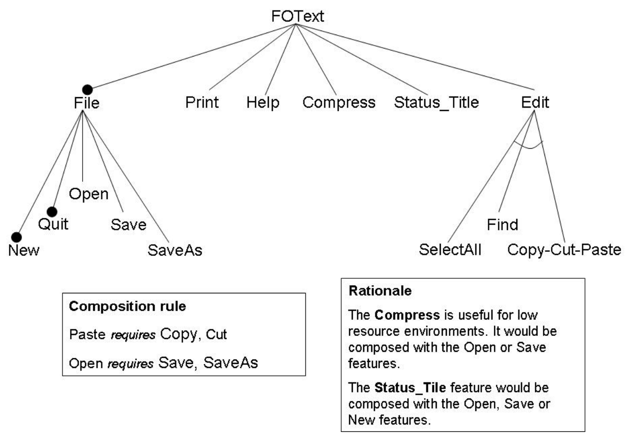
FOText
File
Print
Help
Compress
Status_Title
Edit
Open
Quit
Save
New
SaveAs
Find
SelectAll
Copy-Cut-Paste
Composition rule
Paste *requires* Copy, Cut
Open *requires* Save, SaveAs
Rationale
The **Compress** is useful for low resource environments. It would be composed with the Open or Save features.
The **Status_Tile** feature would be composed with the Open, Save or New features.

Figure 5: FODA diagram of FOText

ture provides the the ability to compress the text files before they are saved, and decompresses them before they are opened. The `Status Title` feature displays the name of the opened file and the name of the file that is being saved in the title bar of the FOText window. It also clears the window title bar when the user starts a `new` file. We specify the latter two as flexible while the former nine are specified as monolithic.

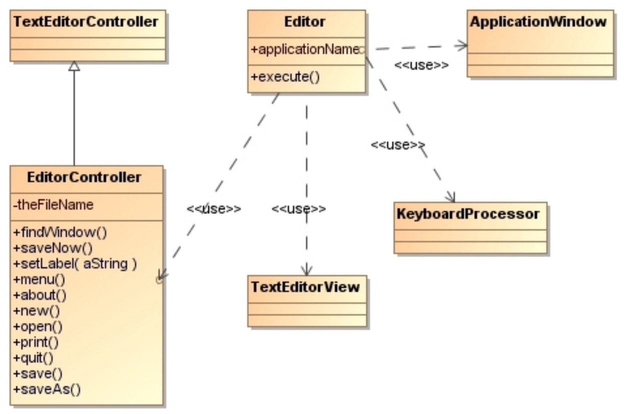

Figure 6: Class diagram of FOText

The UML class diagram of the complete FOText application is presented in Figure 6. The main class `Editor` has a method `execute` that produces an instance of the class `ApplicationWindow`. It provides the window to display and edit text. The `execute` method also creates an instance of the class `TextEditorView` which is linked to an instance of the `EditorController` and `KeyboardProcessor` classes. The `EditorController` inherits from the Smalltalk class `TextEditorController` and adds several functionalities such as a method `menu` which is used to launch the FOText methods that implement the different features. The `KeyboardProcessor` captures the events originated from the keyboard and is linked to an `ApplicationWindow` to embed the text area into the window.

4.2 FOText implementation

We implemented FOText in a standard object-oriented way in our favorite programming language and IDE: VisualWorks for Smalltalk and used the ChEOPS tool [5] to log our development operations as first-class change entities. At the beginning of the development of a new feature, we informed the IDE of its ID and type (*flexible* or *monolithic*). By doing that, our tool is capable of keeping track of what changes the features consist of. From the moment the changes are captured in first-class objects, they can be used to compose features and produce a family of program variations. Table 1 shows some statistics about the number of changes and dependencies that were captured. Note that the numbers of changes and dependencies are about the same size resulting in an average time complexity of $O(n^2)$.

Feature	# changes	# dependencies
Base	130	158
SaveAs	88	106
Save	65	74
Open	101	121
Copy_Cut_Paste	72	82
Find	86	98
SelectAll	89	102
Print	182	226
Help	137	154
Status_Title	159	193
Compress	151	147
Total	1260	1362

Table 1: Statistics of the size of FOText

The `Base` feature provides the main functionalities: a basic word processor that provides a window to type text and a menu with two features: `New` and `Quit`. To this base program, we incrementally add the implementation of the other features. Most of those features *add* a new method to the menu of the FOText application. Some of them, however, also introduce *modifications* (e.g. the `Open` feature modifies the menu method introduced by the `Base` feature) and *deletions* (e.g. the `Compress` feature deletes several *statements* and introduce new ones within existing methods).

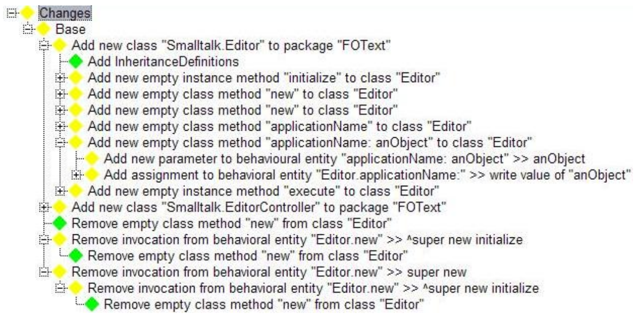
```
Changes
  Base
    Add new class "Smalltalk.Editor" to package "FOText"
      Add InheritanceDefinitions
      Add new empty instance method "initialize" to class "Editor"
      Add new empty class method "new" to class "Editor"
      Add new empty class method "new" to class "Editor"
      Add new empty class method "applicationName" to class "Editor"
      Add new empty class method "applicationName: anObject" to class "Editor"
        Add new parameter to behavioural entity "applicationName: anObject" >> anObject
        Add assignment to behavioral entity "Editor.applicationName:" >> write value of "anObject"
      Add new empty instance method "execute" to class "Editor"
    Add new class "Smalltalk.EditorController" to package "FOText"
    Remove empty class method "new" from class "Editor"
    Remove invocation from behavioral entity "Editor.new" >> ^super new initialize
      Remove empty class method "new" from class "Editor"
    Remove invocation from behavioral entity "Editor.new" >> super new
      Remove invocation from behavioral entity "Editor.new" >> ^super new initialize
        Remove empty class method "new" from class "Editor"
```

Figure 7: FOText: List of logged changes

Figure 7 presents a ChEOPS view which hierarchically presents the change objects that were captured implementing the Base feature. It contains *additions* and *deletions* of classes, methods, instance variable and statements. This shows that (a) our model is capable of expressing features that include deletions of program building blocks, (b) our model allows features to specify operations up to the level of statements and (c) that our approach allows to do FOP while programming in an ordinary object-oriented programming language.

4.3 Feature Composition

Our model allows the composition of a program variation by specifying the features that variation should include. Some compositions, however, are not possible due to unsatisfied dependencies. Thanks to the fine-grained level of feature specification, our tools can check wether a composition is valid. In case it is not, they can assist in resolving the conflict.

We implemented an extension to ChEOPS that includes the notions of monolithic and flexible features, the composition algorithm and a graphical engine that can produce diagrams such as those depicted in Figures 8 - 11. As a base for the graphical framework, Mondrian [12] was used. In this section, we present four compositions which demonstrate that our model and tools provide support to do FOP and that they increase the reusability of features.

4.3.1 A valid composition

In this first scenario, we want a variation of FOText that includes all features. Our tool informs that this composition is valid and depicts the change composition graph of Figure 8.

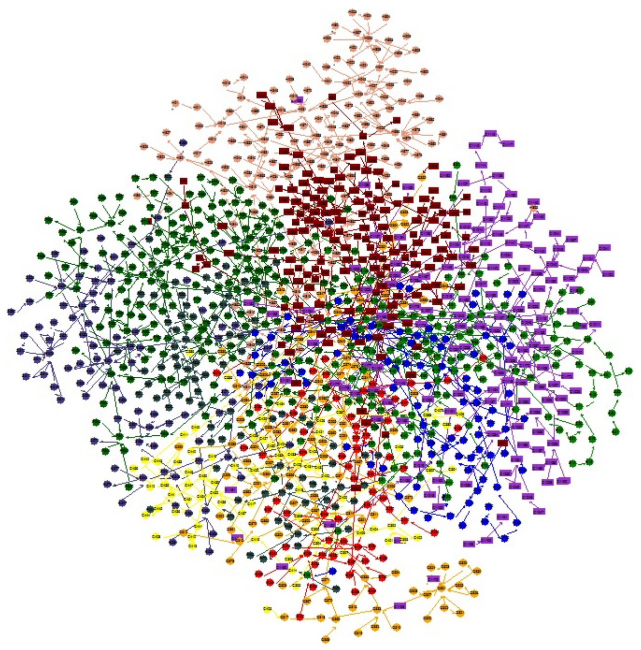

Figure 8: Composition of all features

This composition involves 11 features which are specified by 1260 changes. The time required to display the diagram in Figure 8 was 281873 milliseconds (approximately 5 minutes) with a computer with 2GB of RAM and a processor that clocks at 1.66GHZ. A closer inspection learned us that our composition algorithm validated the composition in only 183 milliseconds and that the remaining 281690 milliseconds were used by Mondrian for lay-outing all change objects.

4.3.2 An invalid composition

The second composition involves the `Base` and `Save` features. Figure 9 depicts the result of this composition: The changes belonging to the `Base` and `Save` features are respectively depicted as red and green circles. The black nodes represent the change objects from the `error` list: changes that belong to the features in the composition which have at least one unsatisfied dependency.

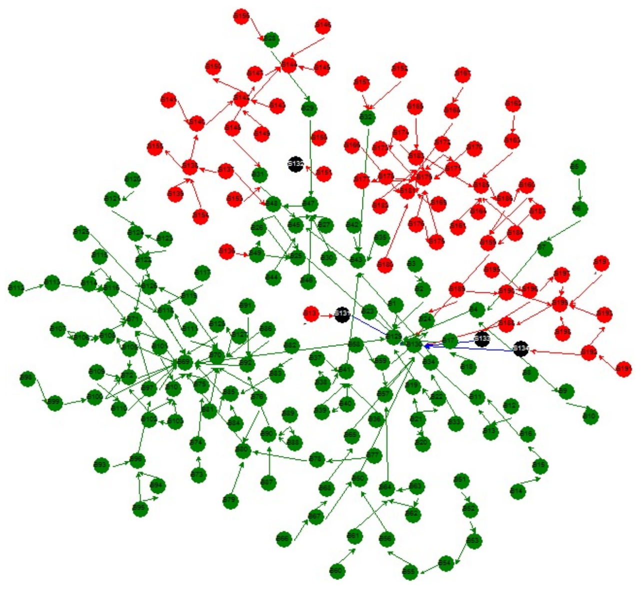

Figure 9: Composition of `Base` and `Save`

The graphical view of Figure 9 can be used to assist programmers to debug their compositions. By inspecting the black nodes of the diagram in Figure 9, one can find out that there are four changes from the `Save` feature that depend on changes of the `SavesAs` feature. Consequently, in this implementation of FOText, the `Save` feature cannot be included in a composition without including the `SavesAs` feature. In case this is not desired, the developer can use the fine-grained information of the inspected black first-class change objects to adapt the implementation of the concerned features.

4.3.3 Valid compositions by means of flexible features

Our approach provides *flexible* features which are deployed in a specific way depending on the composition they belong to. Consequently, a *flexible* feature provides a customized functionality depending on the features that are present in a composition. In the third and fourth scenario, we demonstrate this by composing the flexible `Compress` feature with different features.

We first compose `Compress` with the `Base` and `SaveAs` features. Figure 10 shows this composition: Changes belonging to `Base`, `SaveAs` and `Compress` are respectively depicted as green circles, blue circles and yellow boxes. Note that Figure 10 contains a gray node that belongs to the `Compress` feature, but that will not be applied due to its dependency to a change that does not reside in the composition (a change of the `Open` feature).

In the final composition, we add the `Compress` feature to a viewer version of FOText which is composed by the `Base` and `Open` features. The result of this composition is depicted by Figure 11. Changes of `Base`, `Open` and `Compress` are respectively depicted as green circles, blue circles and

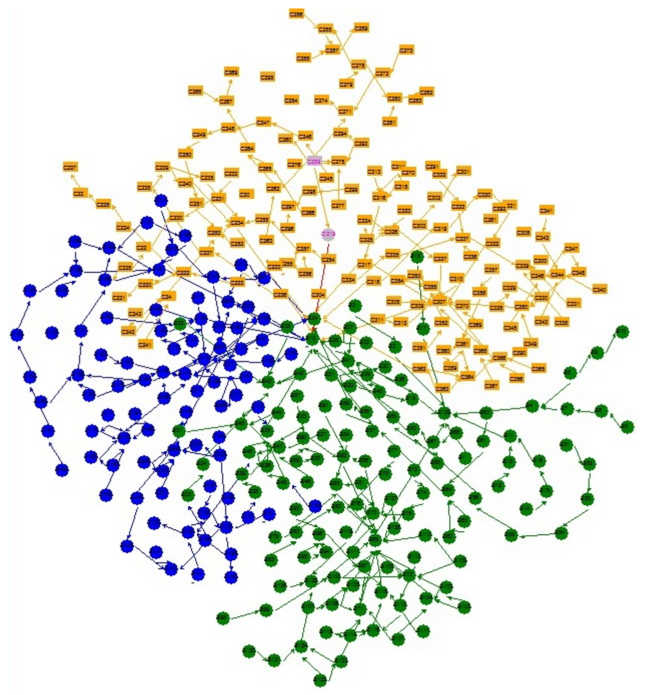
Figure 10: Composition of `Base`, `SaveAs` and `Compress`

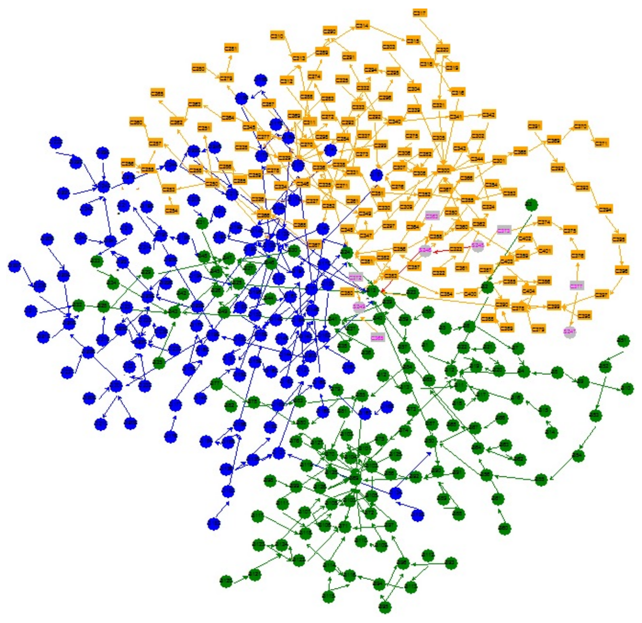
Figure 11: Composition of `Base`, `Open` and `Compress`

yellow boxes. In this composition, several changes of the flexible `Compress` feature are grayed out and omitted from the composition (because they depend on changes of the `SaveAs` feature).

A closer inspection of the gray entities of both figures learns us that different change objects of the `Compress` are concerned: C259 in Figure 10 and C261, C258, C229 in Figure 11. This shows how our approach and tools automatically customize *flexible* features to make compositions valid. It shows how this technique allows compositions that would not be permitted by other FOP approaches, but which do make sense. Consequently, this shows how our model allows a customized feature deployment, improving the reusability of features.

5. CONCLUSIONS AND FUTURE WORK

Procedural, functional and object-oriented programming languages do not provide enough means to cope with software variations. Adding a functionality to a family of software variations by means of such a language can only be accomplished by implementing the solution directly into the code of each variation. That solution suffers from a combinatorial explosion and hinders reusability [11].

A better approach is to use *Feature-Oriented Programming* (FOP), which allows the production of software variations by composing features. In FOP a feature is a modular building block that adds a functionality to a system. The state-of-the-art approaches to FOP lack expressiveness which is manifested by four problems we identified. All approaches specify features only by the *addition or modification* of software building blocks. Moreover, the *granularity* they provide rarely reaches the statement level. Thirdly, none of them allow a *bottom-up approach* to FOP that does not require a specific language support. Finally, most of them do not provide means to *manage features that implement crosscutting functionalities*. We believe that specifying features by means of change objects allow us to overcome these issues.

We propose a change-based approach to FOP based on the *Change-Oriented Software Engineering* model [6]. In our model, features are specified by a set of *changes* that have to be applied in order to implement the functionality that feature offers. *Changes* model the operations (*addition*, *modification* and *deletion*) of all kinds of software building blocks (classes, methods, instance variables and statements). *Changes* are instrumented with explicit dependencies which provide information about the validity of feature compositions.

We introduce *flexible* features as an appropriate concept for modeling crosscutting features. A *flexible* feature is specified by a set of changes that does not have to be applied as a whole in order to add the feature to a composition. We present a composition verification algorithm that is capable of automatically customizing *flexible* features in such a way that they never make a composition invalid. This improves the reusability of features as they can be added to any composition without having to adapt them.

We provide an implementation based on ChEOPS [6] that captures the changes as first-class entities and that allows the programmer to compose features. We also provide a graphical tool based on Mondrian [12] that might assist in debugging invalid compositions.

We evaluate our approach by implementing FOText (a simple word processor) in a standard object-oriented programming environment. We incrementally add eleven functionalities to FOText of which two are specified as flexible features. We use our tools to capture the development operations in first-class change objects and produce four compositions of which we evaluate the validity.

The evaluation shows that our model overcomes the four drawbacks we found in the state-of-the-art approaches () but also reveals two opportunities for future work. First, the evaluation shows that the usability of our graphical tools decreases when the number of changes grows. This insinuates that our model does not scale up. In order to tackle this issue, one track of feature work is to introduce filters that provide a customized view on the change objects, hiding away unwanted level of detail. Second, the order in which the features of a composition are specified influences the result of the composition validity. The second track of future work consists in overcoming this undesirable effect. The problem can be partially tackled by adapting the composition algorithm. In order to fully overcome it, however, a conceptual adaptation of our change-based model is required. In the current version, flexible features are specified by an extensive set of changes, rather than an intensive description of that set. We envision to include higher-order changes to represent intensions like: "Add an invocation to method `m` in *all* methods of class `C`". This will again increase feature reusability.

6. ACKNOWLEDGMENTS

We want to thank dr. Pascal Costanza and dr. Patrick Heymans for their vauable feedback with regards to this research. Also, we would like to acknowledge Leonel Merino, who performed most of the evaluation of our FOP approach. Finally, we want to thank the IWT and the Brussels Capital Region who financed this research.

7. ADDITIONAL AUTHORS

Additional authors: prof. dr. Theo D'Hondt (Vrije Universiteit Brussel, email: `tjdhondt@vub.ac.be`), prof. dr. ir. Yolande Berbers (Katholieke Universitei Leuven, email: `yolande@cs.kuleuven.be`).

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

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

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

[7] D. B. Ed Jung, Chetan Kapoor. Automatic code generation for actuator interfacing from a declarative specification. In *International Conference on Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ*, pages 2839 – 2844, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, June 1997.

[11] T. Männistö, T. Soininen, and R. Sulonen. Modeling configurable products and software product families. In *in Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-2001) - Workshop on Configuration*, 2001.

[12] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: an agile information visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144, New York, NY, USA, 2006. ACM.

[13] S. Nakkrasae and P. Sophatsathit. 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*, pages 773–780, New York, NY, USA, 2002. ACM.

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

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

[16] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.

[17] 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.