USING A CONCEPT-BASED APPROACH TO ASPECT-ORIENTED SOFTWARE DESIGN

Dennis Wagelaar & Lodewijk Bergmans

TRESE GROUP – UNIVERSITY OF TWENTE, P.O. BOX 217, 7500 AE, ENSCHEDE, THE NETHERLANDS

{wagelaar|bergmans}@cs.utwente.nl

http://trese.cs.utwente.nl

ABSTRACT – Aspect-oriented programming (AOP) has received considerable interest, in particular as an extension of object-oriented programming. However, current object-oriented software design techniques, such as UML, are not well suited to cope with aspect orientation. In this paper we discuss a design approach, called CoCompose, which supports aspect orientation and software evolution. The CoCompose design language adopts a generic *concept* construct for describing software systems. Design-level relationships between concepts can be expressed by applying the parameterised *feature* construct. Complete CoCompose models can be translated into executable programs using an automatic translation process.

Keywords

Aspect Orientation, Advanced Separation of Concerns, Composition Filters, Design Patterns, software design, software evolution.

1. INTRODUCTION & PROBLEM STATEMENT

One of the goals of software design models is to achieve/retain an overview of the software system under construction. It determines the high-level structure and behaviour of a software system that an implementation should follow. Because of this, software design techniques should leverage the programming techniques that can be used for describing behaviour and structure.

Aspect-oriented programming [1] has received considerable interest, in particular as an extension of object-oriented programming. However, current object-oriented software design techniques, such as the *Unified Modeling Language* (UML) [2], are not well suited to cope with aspect orientation and evolving software requirements. Some of the problems that may arise are discussed below:

- There are numerous design concepts (e.g. described by *Design Patterns* [3]), which cannot all be offered as (design) language constructs. Language extension mechanisms can be used to facilitate newly introduced design concepts. Currently used extension mechanisms do not offer a way to describe well-defined semantics for these design concepts, however.

For example, we can treat Design Patterns as design concepts. It is not feasible to present each design pattern as a separate design language construct. When using UML, we can use the stereotype extension mechanism to introduce the syntactic part of a design pattern. However, it is not possible to define the semantic part (i.e. how the application of the design pattern affects the design) for the new stereotype.

- In current design approaches, the implementation semantics (e.g. aspect, class, interface, method, etc.) of each design element need to be fixed when introducing the design element, even though it may be difficult to choose the exact semantics. When adding new design elements and/or requirements, existing design elements may require adopting different implementation semantics. If such a transformation is not possible, removal and reintroduction of the design elements is required.

For example, UML represents design elements using a fixed set of constructs (e.g. class, interface, method, etc.). It is not possible to introduce design elements in UML without determining their construct form. Consider a situation in which we want to be able to exchange an existing method within an object class with methods that follow a different strategy. We can apply the *Strategy design pattern* [3](315) to achieve this, but that requires changing the structure of the design.

- If the design does not map injectively to the implementation, automated translation from design to implementation is difficult and may be incomplete. Updates to the design require separate updates to the implementation. If, in addition, the structures of implementation and design differ, a design update may cause a disproportional amount of updates in the implementation. This is particularly well exemplified if the design represents crosscutting concerns that cannot be expressed in the implementation language.

For example, the Java [4] programming language does not natively support aspects. When modelling aspects in the design language, the aspects do not map proportionately to Java. The Java implementation does not preserve the structure of the design. If the mapping from the design to the implementation is not automated, a change to the aspect in the design may require many changes in the implementation.

This paper is structured as follows: in the next section we will present some further requirements. The approach to our design technique will be discussed next, followed by an explanation of the design language and the translation of this

design language. The design language and its translation will both be illustrated by an example.

2. REQUIREMENTS

In addition to addressing the problems stated in the previous section, we have defined a number of additional requirements for our design language:

- *Well-defined semantics of (design) language constructs*: in order to reason about the semantics of a design and to automate implementation, the semantics of the design language constructs must be well defined.
- *Scalability of design concept representation*: by expressing the semantics of a design concept in the design language itself, we can recursively define design concepts.
- *Comprehensibility and intuitiveness of the design*: the designs expressed in the design language are meant to be comprehensible and intuitively clear.
- *Design evolution*: one should be able to introduce the design elements in an abstract way. The design elements can gradually be refined to achieve an executable design.

3. APPROACH

This section describes the approach used to solve the problems and meet the requirements stated in the previous sections. Our design approach, called *CoCompose*, will be explained by discussing the language constructs (*features* and *concepts*), automated implementation and the visualisation of the language.

3.1 FEATURES AND CONCEPTS

CoCompose offers a mechanism, called *feature*, to represent (abstractions of) design concepts, which allows the designers to introduce new design concepts (e.g. relations or design patterns), to be used within one or more concrete designs. The definition of a feature can contain a description of the structure of the design concept, again expressed in terms of concepts and features. This means that features can be defined recursively.

To avoid forcing designers to prematurely select a particular implementation construct, our design language adopts the generic notion of *concepts* to represent elements in a design. Only when implementing the design, the concrete implementation constructs (e.g. aspects, object classes, methods, interfaces, etc.) of those concepts are determined. Changing concrete implementation constructs for concepts does not change the design.

3.2 EXECUTABLE DESIGN

CoCompose designs are executable, in the sense that a (complete) design can be translated into an executable language. This translation can be fully automated, thereby allowing each update of the design model to immediately result in an updated implementation, regardless of the impact upon the implementation. This is only possible because the elements of the design language (most notably *concept* and *feature*), have well-defined semantics. This differs from UML stereotypes, for example, where the meaning of a stereotype is defined outside the model.

The approach used for translation of CoCompose designs is based upon *Design Algebra* [6](150), which introduces techniques for determining and selecting concrete (programming) language constructs for implementing concepts (i.e. finding a *Design Space alternative*).

Design Algebra introduces several techniques to reduce the possible alternatives in the design space. Eventually that should result in a balanced alternative that best meets the qualities needed. CoCompose uses a method that is based upon these techniques. It will be referred to as the *Translation Process* and is described in section 6.

3.3 VISUALISING DESIGN

To address the comprehensibility and intuitiveness of designs, a visual representation of the language has been adopted. The ability to apply abstractions of design concepts (features) allows us to express a design at a high level of abstraction, hiding many details. This allows for the number of design elements to stay at an acceptable level for human understanding.

4. THE COCOMPOSE DESIGN LANGUAGE

The CoCompose design language allows the designer to model a software system using concepts. This design can then be translated into a target (programming) language. The design language elements will be discussed in the following paragraphs.

4.1 CONCEPTS

Concepts are the basic language constructs for CoCompose. They represent parts of a software system and can have relations to other concepts. CoCompose visualises concepts as shown in Figure 1.

Figure 1: A concept.

IMPLEMENTATIONS

Concepts can have implementations for several target languages and several concept forms. For example, a concept can have an implementation as a *Method* in Java. This imposes a constraint on the concept, which states that this concept can only be implemented in Java and only as a *Method*. By specifying several (alternative) implementations, the constraints on the concept will be less strict.

If no implementations are specified for a concept, it is assumed that features (described in the next paragraph) will define all of its implementation.

4.2 FEATURES

Features are reusable abstractions of design concepts, such as an *Association relation* or an *Observer design pattern*. They can be applied to concepts to describe them and the relations between them. Features have well-defined semantics, form

and constraints. The name, type, comments and specification of the feature describe its semantics. The form is described by the feature's roles (explained below). The constraints restrict the possible applications of features (e.g. only to concepts with certain properties).

A feature type represents a specific feature form and semantics. Several feature instances of a type can exist. Feature instances can be applied to concepts through roles (described below). CoCompose visualises features as shown in Figure 2.

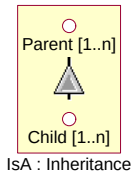

Figure 2: A feature.

In Figure 2, a feature named *IsA*, which is an instance of feature type *Inheritance*, is shown. This particular feature type defines two roles (as explained in detail below), named *Child* and *Parent*. For each of the roles its multiplicity ("[1..n]") is shown. The icon is particular to the feature type.

A feature can be used to model (a/o):

- Constraints on concepts,
- Relations between concepts,
- Design patterns applied to concepts,
- Software components.

ROLES

Feature roles describe the role of concepts in a feature relation. Concepts that are related through a feature fill a role of that feature.

For example, an *Inheritance* feature, as shown in Figure 2, has two roles: a *Parent* role and a *Child* role. A concept that fills a *Parent* role will be the ancestor in an inheritance relation.

Like features, roles can have assigned constraints, which restrict the possible ways to fill the role. In addition to these constraints, roles have specific multiplicity constraints. Multiplicity constraints specify how often a role can or must be filled. Possible multiplicity constraints are "0..1", "0..n", "1..1" and "1..n".

SOLUTION PATTERNS

Features have *solution patterns* to "solve" the feature to a structure of concepts and features that represent the feature semantics. This solution pattern is applied to the concepts that fill the feature's roles.

A solution pattern is expressed in the CoCompose language. By being able to describe solution patterns, one can recursively define CoCompose models. This will be shown in section 5.

A solution pattern consists of a *default part* and several *role parts*. The default part is a fixed structure of CoCompose language constructs that will be applied once for each feature type. Each role part is a parameterised structure of CoCompose language constructs that will be applied once for every time that role is filled by a concept. In a solution pattern all role parts are represented in a single model to show the relations between the role parts and the default part. Through the use of features with solution patterns, high-level design abstractions can be used in CoCompose designs, hiding the details through (possibly nested) features. Solution patterns are used in the first step of the Translation Process, which is described in section 6.

IMPLEMENTATION PATTERNS

In addition to solution patterns, features have *implementation patterns* to translate the feature to a specific (target) language (e.g. Java, C++). This implementation pattern is applied to the concepts that fill the feature's roles.

An implementation pattern is expressed in a specific language; it describes an implementation for the feature in a specific target language.

Like solution patterns, an implementation pattern consists of a *default part* and several *role parts*. The default part is a fixed structure of target language constructs that will be applied once for each feature type. Each role part is a parameterised structure of target language constructs that will be applied once for every time that role is filled by a concept.

Implementation patterns are used in the last step of the Translation Process, which is described in section 6.

5. EXAMPLE

Let us now consider an example CoCompose model of a window manager for a graphical user interface with several windowed controls.

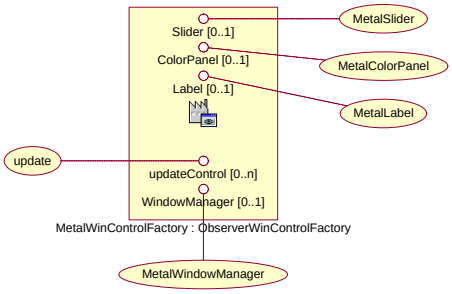

Figure 3: An example CoCompose model.

The model in Figure 3 shows a feature named *MetalWinControlFactory* of the high-level feature type *ObserverWinControlFactory*. This means that *MetalWinControlFactory* is an *application* of the *ObserverWinControlFactory* feature type to the concepts *update*, *MetalWindowManager*, *MetalLabel*, *MetalColorPanel* and *MetalSlider*. The application of the feature is done through roles; each concept fills a role in the application of the feature, as designated by connecting the concepts to the roles using solid lines.

For example, the *MetalWindowManager* concept fills the *WindowManager* role in the application of the *MetalWinControlFactory* feature, which means that the *MetalWinControlFactory* feature will impose *WindowManager* functionality on *MetalWindowManager* concept.

Now that the top-level model is given, we can show how to "solve" this feature by means of a solution pattern[1]. This solution pattern is depicted in Figure 4.

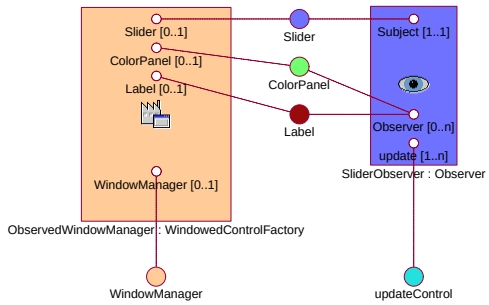

Figure 4: The solution pattern for the ObserverWinControlFactory feature type.

Note that the larger circles in this model represent the roles of the *ObserverWinControlFactory* feature type. These circles are called *role concepts* and will be filled by concepts when the feature is applied. The *Observer* design pattern has been used in this model to add *Observer* functionality to the *WindowedControlFactory* feature type. The *SliderObserver* feature has the same colour as the *Slider* role concept to show that this feature belongs to the *Slider* role part of the solution pattern[2].

The solution pattern for a *WindowedControlFactory* feature type is given in Figure 5.

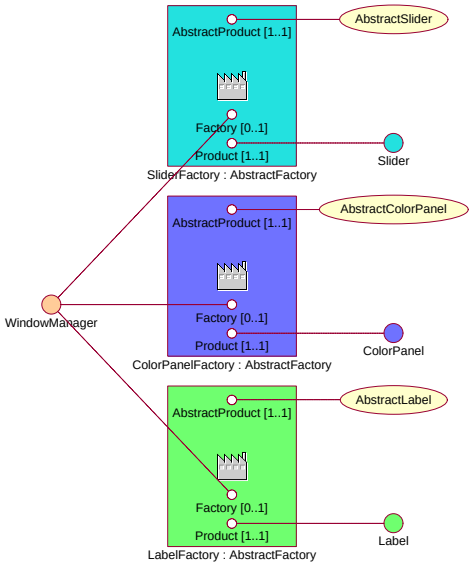

Figure 5: The solution pattern for the WindowedControlFactory feature type.

The model uses an *Abstract Factory* design pattern to enable the window manager to create several windowed controls. This is modelled by the application of several features of the type *AbstractFactory*. The *AbstractFactory* feature type has been applied for every product the factory needs to create. Separate feature instances are needed here, because each *AbstractProduct* relates to one *ConcreteProduct*.

[1] In an actual design process, either an existing feature type is reused, or a feature type is introduced to abstract from details that can be added at a later stage.

[2] For every time the *Slider* role is filled, a *SliderObserver* will be generated.

The solution pattern for the *AbstractFactory* feature type is depicted in Figure 6.

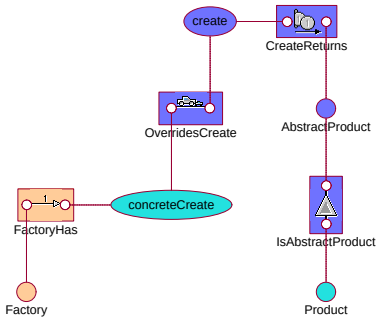

Figure 6: The solution pattern for AbstractFactory.

It is based upon the solution given in [3](87). Note that the *Abstract Factory* role has been left out, because the feature encapsulates the abstract behaviour.

The solution pattern for the *Observer* feature type is depicted in Figure 7.

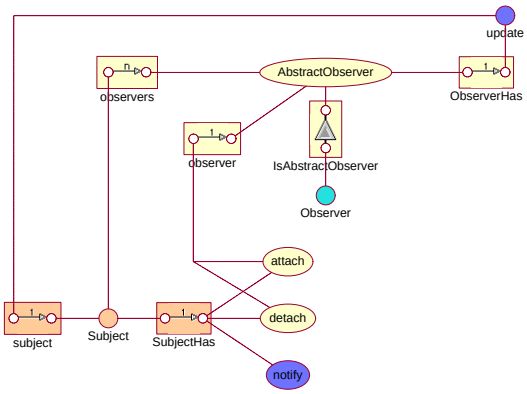

Figure 7: The solution pattern for Observer.

It is based upon the solution given in [3](293). Note that an *Abstract Subject* is not used in this solution pattern. The abstract subject functionality is *superimposed* to the Subject by directly applying the features to the Subject role concept.

6. THE TRANSLATION PROCESS

A process for translating a CoCompose model into a target (programming) language is needed in order to make the CoCompose language executable. The process used here consists of three steps:

I. FLATTENING THE MODEL – Solve all the features that have solution patterns. The result will be a lowest level model that contains only those features with implementation patterns expressed in the target language.

II. DETERMINE CONCEPT FORMS – For each concept, check all filled feature roles for possible target language construct forms. If implementations exist for this concept, also check the possible target language construct forms of those. Select the least complex form required from the joint set of possible concept forms.

III. GENERATE IMPLEMENTATION – Translate each concept to its concept form for the target language and apply the concept implementation, if any. For each feature, apply

the implementation pattern for the selected concept forms. The result is an implementation in the target language.

Each of these steps will be explained in the next paragraphs.

6.1 FLATTENING THE MODEL

Before one can translate a CoCompose model into a target language, all dependencies and links should be reachable in order to check them. To make this easier, one could flatten the model to a lowest abstraction level model, which no longer contains features without implementation patterns expressed in the target language.

For each feature it should be checked whether it has an implementation pattern expressed in the target language. If no such implementation pattern exists, the solution patterns will have to be checked for eventual translation into the target language[1].

A solution pattern traversal mechanism is used for this. This mechanism builds a tree of nodes represented by solution patterns. The algorithm for building this tree is described below in pseudo-code:

```
boolean CheckFeature(feature, tree)
// Checks if the feature will eventually translate to target.language
  if feature.hasImplemPatternIn(target.language) then
    return true;
  else if feature.hasSolutionPattern() then
    forall solution in feature.getSolutionPatterns()
      tree.insert(solution);
      if not CheckSolution(solution, tree) then
        tree.remove(solution);  // dead branch
      else
        return true;
      end if;
    end forall;
  end if;
end CheckFeature;

boolean CheckConcept(concept)
// Checks if the concept will translate into target.language
  if not concept.hasImplementations() then
    return true;
  else if concept.hasImplementationIn(target.language) then
    return true;
  end if;
  return false;
end CheckConcept;

boolean CheckSolution(solution, tree)
// Checks if applying the solution pattern will result in a model
// that eventually translates to target.language
  forall concept in solution.concepts
    if not CheckConcept(concept) then
      return false;
    end if;
  end forall;
  forall feature in solution.features
    if not CheckFeature(feature, tree) then
      return false;
    end if;
  end forall;
  return true;
end CheckSolution;
```

The first step is to call CheckSolution using the model and an empty tree as arguments. When CheckSolution returns true, there is a valid tree for flattening the model in a way that it can be translated to the target language. The target in this algorithm refers to the target environment, e.g. a Java source path.

If the tree has been determined, the solution patterns in this tree can be applied[2]. This will result in a model containing only features having implementation patterns expressed in the target language and concepts having either no implementation or an implementation expressed in the target language.

When a solution pattern is applied, first the default part is applied once for every feature type. Note that this is done globally and not per feature. Then, for each time a concept fills a feature role, the role part is applied for that concept. Below is the pseudo-code for applying a solution pattern:

```
ApplySolution(solution, target)
// Applies the solution pattern to the target environment
  ApplyDefaultPart(solution, target);
  forall role in solution.feature.roles
    forall concept in role.getLinks()
      ApplyRolePart(solution, target, role, concept);
    end forall;
  end forall;
end ApplySolution;

ApplyDefaultPart(solution, target)
// Applies the default part of the solution pattern
  if not target.featureTypeApplied(solution.feature.type) then
    target.insert(solution.getDefaultpart());
  end if;
end ApplyDefaultPart;

ApplyRolePart(solution, target, role, concept)
// Applies the role part of the solution pattern for the given concept
  target.insert(solution.getRolepart(role, concept));
end ApplyDefaultPart;
```

6.2 DETERMINE CONCEPT FORMS

When translating a CoCompose model into a target (programming) language, the concepts will need to be translated into constructs of that target language. The construct into which a concept will be translated is called the *concept form* for that target language (this is called *property* in [6]). It is based upon the available implementation patterns of features, for which the concept fills a role, and concept implementations, if any.

Each feature that has an implementation pattern in the target language can have several role-part implementations for the same role. Each role-part implementation for that role provides an implementation for a different concept form.

For example, the *Parent* role of an *Inheritance* feature may have an implementation in Java for a concept form of *Class* or *Interface*. This is described as follows:

$$F_{Parent} = \{Class, Interface\}$$

[1] This means checking whether the resulting model, after applying the solution patterns, contains features with implementation patterns expressed in the target language and concepts that have either no implementation or an implementation expressed in the target language

[2] Note that applying a path of solution patterns from the tree eliminates flattening alternatives that might have been useful or even better, depending on the result of the second step (*Determine Concept Forms*). This problem will be addressed in the future.

A concept filling this role can now be implemented as one of these concept forms. In a CoCompose model a concept will often fill more than one role. Consider a *SingleRelation* feature, which models a one-way, one-to-one relation between two concepts. It has an *Owner* and *Property* role. The *Property* role has a solution in Java for a concept form of *Class*, *Interface* or *Method*[1]. This is described as follows:

$$F_{Property} = \{Class, Interface, Method\}$$

If a concept fills this role as well, the intersection of the concept form sets for each role must be taken:

$$F_{Concept} = F_{Role\_1} \cap \ldots \cap F_{Role\_n}$$

Which in this case results in:

$$F_{Concept} = F_{Parent} \cap F_{Property} = \{Class, Interface\}$$

If a concept has pre-defined implementations, the concept forms of these implementations will have to be taken into account as well. Otherwise, the concept form will be completely determined by the available solution patterns of the feature types (*Inheritance* and *SingleRelation* in this case).

If there are multiple valid concept forms, one of them must be selected. This selection process could be automated through the use of heuristics. One straightforward heuristic is a user-defined *translation priority* for all concept forms available for a target language. This translation priority determines what concept form will be chosen from a set of possible concept forms. The CoCompose translation priority heuristic chooses the concept form with the highest defined priority.

For example, in the Java language the priority of language constructs can be described as follows:

$$C_{Class} < C_{Interface} < C_{Method} < C_{BasicType}$$

This results in the described example concept being translated to an *Interface*, because it is considered the highest priority concept form possible. Of course, any other definition of translation priority is possible.

6.3 GENERATE IMPLEMENTATION

To generate an implementation of the model in the target language, the concepts should first be translated into the concept forms that were determined in the previous step. Then, the concept implementations for the selected concept form should be applied to all concepts that have an implementation. Finally, the implementation patterns of the features for the selected concept forms should be applied. Below is the pseudo-code that describes this:

```
forall concept in model.concepts
   target.insert(concept, concept.form);
   if concept.hasImplemAs(concept.form) then
      target.insert(concept, concept.getImplemAs(concept.form));
   end if;
end forall;

forall feature in model.features
   feature.applyImplemPattern(target);
end forall;
```

[1] Note that the possible concept forms of a feature role may depend on the concept form chosen to fill the other role. For example, a *Method* can fill the *Owner* role of the *SingleRelation* as long as a *Method* does not fill the *Property* role.

The algorithm for feature.applyImplemPattern is variable and it is defined externally.

7. APPLYING THE TRANSLATION PROCESS TO THE EXAMPLE

Let us now see how the translation process works for the example model shown in Figure 3. The example model will be translated into a ConcernJ [5] implementation in order to show the mapping to an Aspect Oriented programming language. The translation process application will be explained for each step. Note that the result of each step can be found in the appendices.

7.1 FLATTENING THE MODEL

The example model has one feature, of which the solution pattern is described in CoCompose. This feature will be solved in the first flattening iteration. The flattening step will have to be repeated another two times to eliminate all CoCompose solutions. The flattening of the example model is described in Appendix A.

7.2 DETERMINE CONCEPT FORMS

Now that all concepts are visible in the flattened example model, the target language construct forms, to which the concepts will translate, can be determined. The target language in this case is ConcernJ, a programming language that facilitates Composition Filters and Aspect Orientation in Java. The concept form translation priority heuristic can be described as follows:

$$C_{Concern} < C_{Method} < C_{BasicType}$$

Note that only the concept forms used for the example are described here. We can describe the deduction of a concept form using the following template:

```
Feature1.Role                                  → {Concern, Method, ...}
...
Featuren.Role                                  → {Concern, ...}
Concept.Implementation                         → {Concern, ...}
Concept.Forms = {Concern, Method, ...} ∩ {Concern, ...} = {Concern, ...}
Concept.Form = Priority({Concern, ...})
```

In the template, the possible concept forms of the feature roles are determined first. Then the possible concept forms for the concept implementations are determined. The intersection of these sets of possible concept forms is taken and the concept form with the highest priority heuristic value is chosen. For each concept the form and its deduction is given in Appendix B.

Note that the concept form of the concept filling the *Property* role determines the possible concept forms for the *Owner* role of a *SingleRelation* feature. For example, if a *Method* fills the *Property* role, a *Class* can only fill the *Owner* role. If a *Class* fills the *Property* role, then *Owner* can be a *Class* or a *Method*.

7.3 GENERATE IMPLEMENTATION

The ConcernJ concept implementations and feature implementation patterns will not be given here. Instead, the resulting ConcernJ implementation is given in Appendix C.

We will illustrate this step by focusing on the implementation of the *MetalSliderObserver* feature introduced in Appendix A, Figure 8. This feature translates into a ConcernJ *concern* construct, preserving part of the structure used in the CoCompose model. This shows ConcernJ can describe the application of a feature. A *filtermodule* is used to describe the *Subject* role and the *Observer* role. The *update* role is encapsulated in the *AbstractObserver* concern.

8. RELATED WORK

In this section we briefly point to a selection of related work.

K. Czarnecki and U. Eisenecker have also used the notion of concept and feature in [7] in the area of *Domain Engineering*. Concepts in *Conceptual Modeling* differ from CoCompose concepts in that they are not necessarily disjoint. Features in *Feature Modeling* are meant to describe the *possible* properties of a single concept, unlike CoCompose features that describes the *actual* properties of several concepts.

The idea of abstracting from programming language construct forms is also used in the area of template/generic programming. In [8], the word "concept" is also used to describe an abstraction from a programming language construct. The difference with CoCompose concepts is that these concepts fully specify their own interface or common form. CoCompose concepts can have their form imposed by features, thus separating each aspect of the concept form in a dedicated feature.

The *Composition Patterns* design approach introduced by S. Clarke and R. Walker in [9] also uses parameterisation in composing several UML models. The most important difference with CoCompose is that the application of a composition pattern is not described explicitly in the UML model that uses it, but it is described separately by composition operators.

N. Noda and T. Kishi have researched the application of design patterns in [10]. They make a separation between the *Design Pattern* and *Application Core*. A similar approach is used by CoCompose, with the difference that the *Design Pattern Application* is, in turn, separated from the rest of the design, or *Application Core*.

9. DISCUSSION & CONCLUSION

The concept-based approach that has been explained in this paper improves stability of designs by postponing the choice for specific language constructs. The recursive modelling of concepts supports a focus on design semantics.

The feature mechanism can be used to model many design concepts, ranging from aspects to design patterns. Because feature solution patterns can be CoCompose models themselves (parameterised by the concepts that fill the roles of the feature), features can be defined recursively. This allows for composition of separate concerns.

By imposing semantics upon concepts through the use of features, the concept semantics can be separated per concern. The feature solution patterns allow for several solutions to be defined for one feature. Choosing between several solution patterns for application within the same context (same concept forms, same target language) is not yet researched.

As this paper has shown, CoCompose maps well to an aspect-oriented programming language, such as ConcernJ. By using feature implementation patterns expressed in other target languages, a mapping to other programming languages, such as AspectJ [11], Hyper/J [12], Java or C++, can be made as well. Of course, implementation pattern definition is more straightforward for a programming language that already supports several advanced composition mechanisms (such as *aspects*, *hyperslices*, *composition filters*, etc.), since the structure of the design can then be (partially) preserved.

10. ACKNOWLEDGEMENTS

We would like to thank Bedir Tekinerdogan and Joost Noppen for reviewing this paper and their comments on this subject.

REFERENCES

[1] Aspect Oriented Software Development, http://www.aosd.net

[2] UML specification, OMG website, http://www.omg.org/uml.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman, Inc., Reading, Massachusetts, 1995, ISBN 0-201-63361-2.

[4] Java, http://java.sun.com.

[5] L. Bergmans, M. Akşit, On the Role of Encapsulation in Aspect-Orientation, University of Twente, Enschede, The Netherlands, 2001, to appear on http://trese.cs.utwente.nl.

[6] B. Tekinerdogan, Synthesis-Based Software Architecture Design, PhD Thesis University of Twente, Enschede, The Netherlands, 2000, ISBN 90-365-1430-4.

[7] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison Wesley, 2000, ISBN 0-201-30977-7.

[8] J. Willcock, J. Siek, A. Lumsdaine, Caramel: A Concept Representation System for Generic Programming, Indiana University, Bloomington, USA, in OOPSLA2001 C++ Template Workshop.

[9] S. Clarke, R. Walker. Composition Patterns: An Approach to Designing Reusable Aspects, in proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001.

[10] N. Noda, T. Kishi, Implementing Design Patterns Using Advanced Separation of Concerns, NEC Corporation, Japan, in OOPSLA 2001 workshop on Advanced Separation of Concerns in Object-Oriented Systems.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An Overview of AspectJ, University of British Columbia, Vancouver, Canada, Xerox Palo Alto Research Center, Palo Alto, USA, University of California, San Diego, USA, in Proceedings of ECOOP 2001, pp. 327-353, 2001.

[12] H. Ossher, P. Tarr, Multi-Dimensional Separation of Concerns and The Hyperspace Approach, IBM T.J. Watson Research Center, Yorktown Heights, USA, in Software Architectures and Component Technology, Kluwer, 2000, ISBN 0-7923-7576-9.

APPENDIX A: THE FLATTENED MODEL FOR THE EXAMPLE

This appendix shows the application of the first step of the Translation Process, the flattening, applied to the example. Three iterations are needed to achieve a fully flattened model.

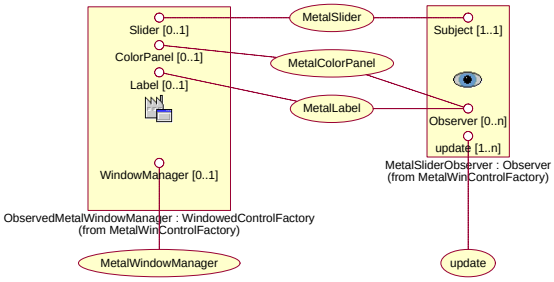

Figure 8: The example model after the first flattening step.

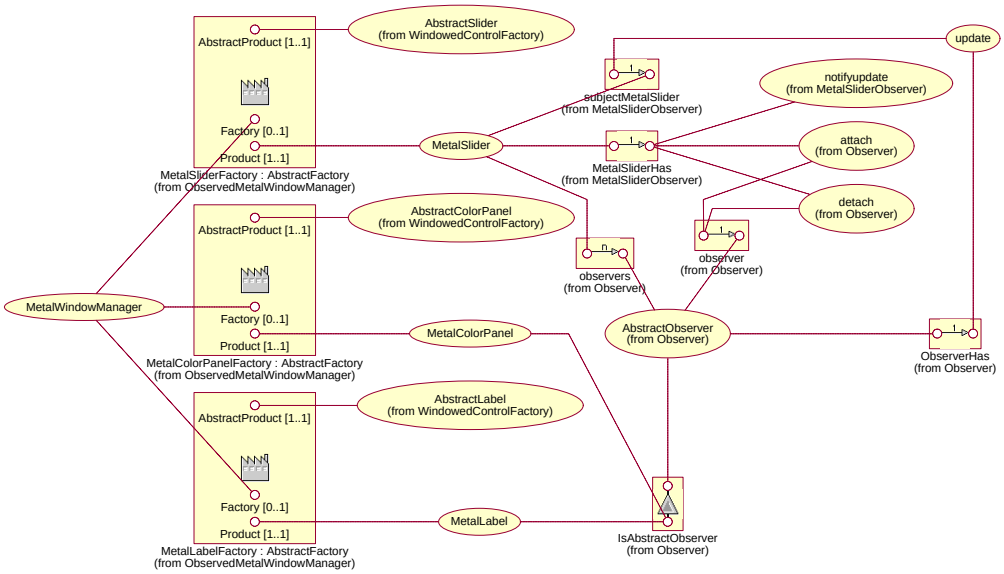

Figure 9: The example model after the second flattening step.

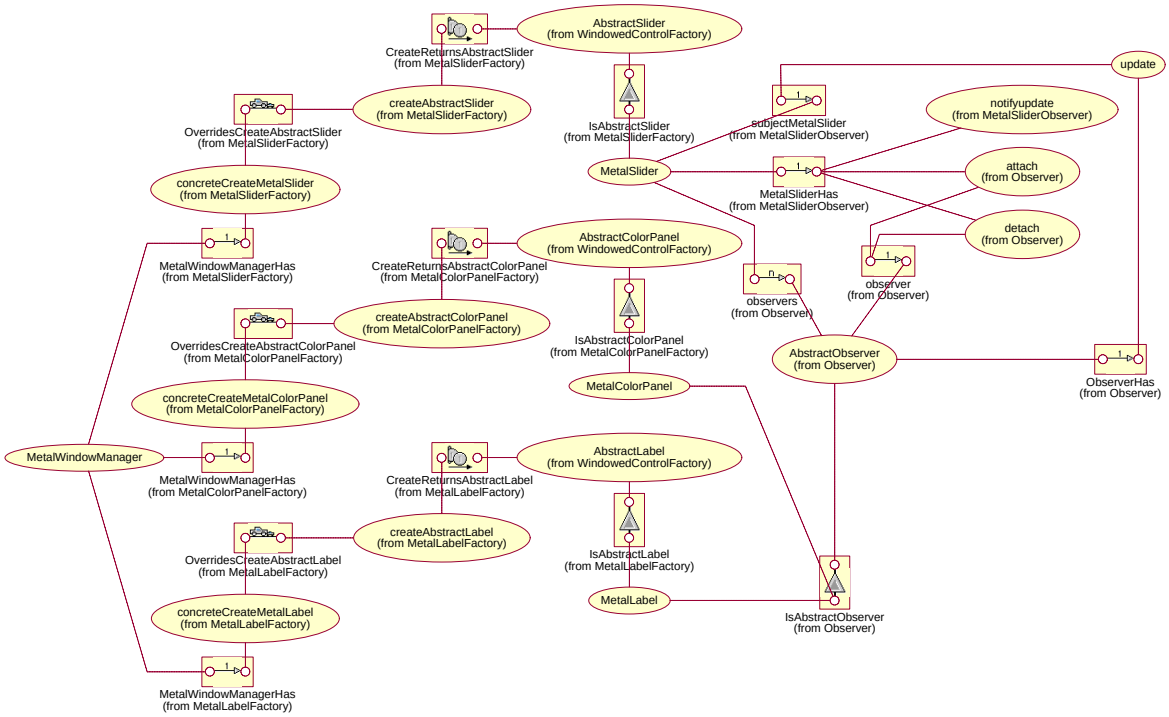

Figure 10: The example model after the third and last flattening step.

APPENDIX B: CONCEPT FORMS FOR THE EXAMPLE

This appendix shows the second step of the Translation Process, concept form determining, applied to the example. Note that the concept forms for the concepts *concreteCreateMetalColorPanel*, *concreteCreateMetalLabel*, *createAbstractColorPanel*, *createAbstractLabel*, *AbstractColorPanel*, *AbstractLabel* and *MetalLabel* are omitted, because it does not contribute in additional explanation.

METALWINDOWMANAGER
MetalWindowManagerHas.Owner → {Concern} (3x)
MetalWindowManager.Forms = {Concern}
MetalWindowManager.Form = Priority({Concern}) = **Concern**

CONCRETECREATEMETALSLIDER
MetalWindowManagerHas.Property → {Concern, Method}
OverridesCreateAbstractSlider.NewMethod → {Method}
concreteCreateMetalSlider.Implementation → {Method}
concreteCreateMetalSlider.Forms = {Concern, Method} ∩ {Method} ∩ {Method} = {Method}
concreteCreateMetalSlider.Form = Priority({Method}) = **Method**

CREATEABSTRACTSLIDER
OverridesCreateAbstractSlider.OldMethod → {Method}
CreateReturnsAbstractSlider.Originator → {Method}
createAbstractSlider.Forms = {Method} ∩ {Method} = {Method}
createAbstractSlider.Form = Priority({Method}) = **Method**

ABSTRACTSLIDER
CreateReturnsAbstractSlider.Result → {Concern, BasicType}
IsAbstractSlider.Parent → {Concern}
AbstractSlider.Forms = {Concern, BasicType} ∩ {Concern} = {Concern}
AbstractSlider.Form = Priority({Concern}) = **Concern**

METALSLIDER
subjectMetalSlider.Property → {Concern, Method}
MetalSliderHas.Owner → {Concern}
observers.Owner → {Concern }
IsAbstractSlider.Child → {Concern}
MetalSlider.Forms = {Concern, Method} ∩ {Concern} ∩ {Concern} ∩ {Concern} = {Concern}
MetalSlider.Form = Priority({Concern}) = **Concern**

METALCOLORPANEL
IsAbstractObserver.Child → {Concern}
IsAbstractColorPanel.Parent → {Concern}
MetalColorPanel.Forms = {Concern} ∩ {Concern} = {Concern}
MetalColorPanel.Form = Priority({Concern}) = **Concern**

UPDATE
ObserverHas.Property → {Concern, Method}
subjectMetalSlider.Owner → {Concern, Method}
update.Forms = {Concern, Method} ∩ {Concern, Method} = {Concern, Method}
update.Form = Priority({Concern, Method}) = **Method**

NOTIFYUPDATE
MetalSliderHas.Property → {Concern, Method}
notifyupdate.Implementation → {Method}
notifyupdate.Forms = {Concern, Method} ∩ {Method} = {Method}
notifyupdate.Form = Priority({Method}) = **Method**

ATTACH
MetalSliderHas.Property → {Concern, Method}
observer.Owner → {Concern, Method}
attach.Implementation → {Method}
attach.Forms = {Concern, Method} ∩ {Concern, Method} ∩ {Method} = {Method}
attach.Form = Priority({Method}) = **Method**

DETACH
MetalSliderHas.Property → {Concern, Method}
observer.Owner → {Concern, Method}
detach.Implementation → {Method}
detach.Forms = {Concern, Method} ∩ {Concern, Method} ∩ {Method} = {Method}
detach.Form = Priority({Method}) = **Method**

ABSTRACTOBSERVER
observers.Property → {Concern}
observer.Property → {Concern, Method}
ObserverHas.Owner → {Concern}
IsAbstractObserver.Parent → {Concern}
AbstractObserver.Forms = {Concern} ∩ {Concern, Method} ∩ {Concern} ∩ {Concern} = {Concern}
AbstractObserver.Form = Priority({Concern}) = **Concern**

APPENDIX C: SOURCE CODE OF THE TRANSLATED EXAMPLE IN CONCERNJ

This appendix shows the third step of the Translation Process, code generation, applied to the example. Note that the implementation of *AbstractLabel*, *MetalLabel*, *MetalColorPanelFactory* and *MetalLabelFactory* is omitted, because it does not contribute in additional explanation.

```
concern MetalWindowManager begin
   implementation in java file "MetalWindowManager.jwf";
end concern MetalWindowManager;

public class MetalWindowManager {
   public MetalWindowManager() {
   }
   // Further implementation…
}
```

```
concern AbstractSlider begin
   implementation in java file "AbstractSlider.jwf";
end concern AbstractSlider;

public class AbstractSlider {
   public AbstractSlider() {
   }
   // Further implementation…
}
```

```
concern AbstractColorPanel begin
   implementation in java file "AbstractColorPanel";
end concern AbstractColorPanel;

public class AbstractColorPanel {
   public AbstractColorPanel() {
   }
   // Further implementation…
}
```

```
concern MetalSlider begin
   implementation in java file "MetalSlider.jwf";
end concern MetalSlider;

public class MetalSlider {
   public MetalSlider() {
   }
   // Further implementation…
}
```

```
concern MetalColorPanel begin
   implementation in java file "MetalColorPanel.jwf";
end concern MetalColorPanel;

public class MetalColorPanel {
   public MetalColorPanel() {
   }
```

```
    // Further implementation…
}
```

```
concern AbstractObserver begin
   implementation in java file "AbstractObserver.jwf";
end concern AbstractObserver;

public class AbstractObserver {
   public AbstractObserver() {
   }
   // Further implementation…
}
```

```
concern MetalSliderFactory begin
   filtermodule factoryModule begin
      methods
         createAbstractSlider() : AbstractSlider;
   end filtermodule factoryModule;
   filtermodule productModule begin
      internals
         abstractSlider : AbstractSlider;
      inputfilters
         disp : Dispatch = { true => abstractSlider.* };
   end filtermodule productModule;
   superimposition begin
      selectors
         factory := Set(MetalWindowManager);
         product := Set(MetalSlider);
      methods
         factory <- { createAbstractSlider };
      filtermodules
         product <- { productModule };
   end superimposition;
   implementation in java file "MetalSliderFactory.jwf";
end concern MetalSliderFactory;

public class MetalSliderFactory {
   public AbstractSlider createAbstractSlider() {
      return new MetalSlider();
   }
}
```

```
concern Observer begin
   filtermodule subjectModule begin
      internals
         observers : Vector;
      methods
         getObservers() : Enumeration;
         attach(observer : AbstractObserver);
         detach(observer : AbstractObserver);
   end filtermodule subjectModule;
   filtermodule observerModule begin
      internals
         abstractObserver : AbstractObserver;
      inputfilters
         disp : Dispatch = { true => abstractObserver.* }
   end filtermodule observerModule;
   superimposition begin
      selectors
         subject = Set(MetalSlider);
         observer = Set(MetalColorPanel, MetalLabel);
      methods
         subject <- { attach, detach };
      filtermodules
         subject <- { subjectModule };
         observer <- { observerModule };
   end superimposition;
   implementation in java file "MetalSliderObserver.jwf";
end concern MetalSliderObserver;

public class MetalSliderObserver {
   public Enumeration getObservers() {
      return observers.elements();
   }
   public void attach(AbstractObserver observer) {
      observers.add(observer);
   }
   public void detach(AbstractObserver observer) {
      observers.add(observer);
   }
}
```

```
concern MetalSliderObserver begin
   filtermodule abstractObserverModule begin
      methods
         update(subjectMetalSlider : MetalSlider);
   end filtermodule abstractObserverModule;
   filtermodule subjectModule begin
      methods
         notifyupdate();
   end filtermodule subjectModule;
   superimposition begin
      selectors
         abstractObserver = Set(AbstractObserver);
         subject = Set(MetalSlider);
      methods
         abstractObserver <- { update };
         subject <- { notifyupdate };
      filtermodules
         abstractObserver <- { abstractObserverModule };
         subject <- { subjectModule };
   end superimposition;
   implementation in java file "MetalSliderObserver.jwf";
end concern MetalSliderObserver;

public class MetalSliderObserver {
   public void update(MetalSlider subjectMetalSlider) {
   }
   public void notifyupdate() {
      AbstractObserver observer;
      Enumeration obs = getObservers();
      while (obs.hasMoreElements())
      {
         observer = (AbstractObserver) obs.nextElement();
         observer.update(this);
      }
   }
}
```