ABSTRACT
Software system family development aims at saving development cost by structured reuse of base assets for a group of similar products. Components (as part of the base assets) that are used for implementing a product line have to be flexible and highly customizable. In this paper CaesarJ, a new aspect-oriented programming language, is proposed that supports the development of such flexible components. The language facilitates the encapsulation of crosscutting concerns and provides a means to develop components as collaborations of classes (Aspectual Collaborations) independent of the context in which they will be used. This paper shows how features of a product line can be designed as Aspectual Collaborations and can be composed afterwards to form a specific product. With the help of a generator, features implemented in CaesarJ can be automatically glued together. To support this, a design notation for modeling Aspectual Collaborations is used as Domain Specific Language (DSL). This DSL allows to (graphically) specify how components (implementing features in the context of a software system family) are linked together. The glue code is then generated automatically. The tools support product family instantiation by combining both the power of Aspectual Collaborations and Model-Driven Software Development (MDSD). Both foster a high level of abstraction in order to increase efficiency, reuse, and software quality.

1. INTRODUCTION
The idea of software product families [1] is not new. Parnas already wrote in the 1970s “We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties and then determining the special properties of the individual family members.” The basic idea is to save cost and speed up development for a group of similar products by structured reuse of base assets, such as architecture, platforms and configurable components. The members of a software product family differ by the features they include. Software product family features thus have to be implemented in such a way that they support variation, which means that they can be separately added to a system or are configurable to accommodate the necessary variability. Sometimes features can be implemented in one coherent component, but often the functionality belonging to a feature spans several parts (components or classes) of an application. In the latter case adding a feature or configuring it consistently is a complex task: The addition or configuration has to be done consistently for the whole feature. Aspect-oriented languages like CaesarJ [3] and ObjectTeams [4] support the modular implementation of collaborating types, so called Aspectual Collaborations. This concept allows implementing crosscutting features independent from each other and from the underlying system they are added to. To instantiate an instance of the family, those collaborations have to be bound to each other to form a complete system. Configuring a system should ideally not require much programming effort. Thus we prototypically implemented a small but sufficient domain specific language (DSL) and generator tool that generates the bindings for CaesarJ collaborations. The DSL uses the concepts of a graphical design notation for Aspectual Collaborations called UML for Aspects (UFA) [6], but unlike UFA it extends UML using stereotypes. This paper shows how aspect-oriented software development and MDSD can be combined to support product family implementation and instantiation.

The remainder of the paper is organized as follows: Section 2 provides an overview of the Aspectual Collaborations model by means of language features of CaesarJ. Section 3 introduces the concepts of the design notation UFA and the developed DSL that extends UML using stereotypes. Section 4 describes the basic idea of MDSD and presents the developed tools and their code generation facilities. We conclude with a note on related work and a summary in section 5 and 6.

2. ASPECTUAL COLLABORATIONS
Aspectual Collaborations are a new programming model that supports the encapsulation of crosscutting concerns. Crosscutting concerns are often implemented by multiple collaborating objects and for this purpose Aspectual Collaborations offer modules that can encapsulate multiple interdependent types [2]. Languages that support this new programming model include CaesarJ [3] and ObjectTeams [4]. In this section we use the CaesarJ language features to describe the Aspectual Collaborations model.

CaesarJ components are collaborations of classes which facilitate the modularization of crosscutting concerns. A collaboration interface specifies the abstractions (types) that together form the component independent of the context in which it will be used. Figure 1 shows an example of such an interface for a generic Observer pattern [5]. This so-called collaboration interface serves as the interface between its implementation and its mapping to a base application. The interface defines two nested classes (roles) Subject and Observer and their respective methods. The interface captures the generic Observer pattern functionality encoded in its own model and terminology completely independent of the context in which it will be used.

1 Classes within an Aspectual Collaboration are called roles.
abstract public cclass ObserverProtocol {  
    abstract public cclass Subject {  
        abstract public void add (Observer o);  
        abstract public void remove (Observer o);  
        abstract public void notify();  
    }  
    abstract public cclass Observer {  
        abstract public void update(Subject s);  
        abstract public void start (Subject s);  
        abstract public void stop (Subject s);  
    }  
}  

Figure 1: Observer Pattern Interface

Figure 2 shows a possible implementation of the pattern. It overrides the virtual classes (nested classes) Subject and Observer inherited from the interface. The method update() of class Observer cannot be implemented within this generic component implementation as it depends on the context in which the pattern will be used in. Therefore the method remains abstract and is not made concrete until deployment time.

public cclass ObserverProtocolImpl extends ObserverProtocol{  
    abstract public cclass Subject {  
        //implementation of methods add, remove, and notify  
    }  
    abstract public cclass Observer {  
        //implementation of methods start and stop  
    }  
}  

Figure 2: Generic Observer Pattern Implementation

The use of an Aspectual Collaboration within a certain context is specified in a so-called binding. Figure 8 shows how the generic Observer pattern can be applied to some existing application - a library management system in this case. Pointcuts and advices allow the extension of the behavior of the application with functionality of the pattern. They specify when the execution of the pattern behavior should be triggered. The wraps construct helps to specify which class of the application plays which role of the Aspectual Collaboration. It declares one class as a wrapper of another class and one can access the wrappee by using the wrappee keyword. The abstract method update() of class Observer is made concrete by delegating to method updateStatus() of BookManager class. Binding and implementation can later be composed using mixin composition to form the final application (see Figure 3).

public cclass LO extends ObserverProtocolImpl & ObserverLibrary {}  

Figure 3: Mixin Composition

As already shown in [2], Aspectual Collaborations provide language support for the implementation of features in the context of software system family development as they provide support for very flexible component implementations. Features that are implemented as Aspectual Collaborations can freely be composed afterwards to form a specific product (as a member of the software system family). For a detailed description of all CaesarJ language features and more examples showing its suitability for software system family development please see [3] and [2]. In this paper we only focus on the language features used within our work on graphical notations and tools for pluggable collaborations to support product family instantiation.

The next section gives an overview of graphical design notations that support the concept of Aspectual Collaborations at design time and serves as a basis for the tools that provide automatic generation of component bindings.

3. UML FOR ASPECTS

UFA (UML for Aspects) [6] is a modeling language for aspect-oriented design that strongly supports the concept of Aspectual Collaborations. This concept has, as described in the previous section, the following properties:

- Components can be implemented completely independent of the application they will be used in.
- Component and application can be composed without having to pollute any of them with integration details. All the integration takes place in a connector-like module called binding. Neither the component nor the application has to be modified. This leads to highly reusable components.
- The behavior implemented in the independent components can be abstract, i.e. they can contain abstract methods.
- Aspectual Collaborations support non-invasive and independent component refinement.

With UFA all these concepts can be expressed at design level supporting a smooth transition from design to implementation [6]. This avoids inconsistencies between design and implementation as Aspectual Collaborations are seamlessly supported during the early stages of the development lifecycle. In this paper we focus on design models and their automatic transition to a concrete implementation, but work has also been done on full lifecycle support [7].

We introduce the design notation UFA and its concepts using the example illustrated in [6]. It shows an application for managing books in a library. In the example, BookCopys are books (the real physical entities) that can be borrowed and returned by clients of the library. A BookManager acts as a catalogue of books and registers or unregisters BookCopys as they are bought or returned to the library. The status of a BookCopy (whether it is available or not) should be kept up to date, i.e. the status should be updated whenever a BookCopy is borrowed or returned by a client. To fulfill this requirement, the Observer pattern [5] should be applied to the library application with each BookCopy in the subject role and BookManager in the observer role. As the subject-observer protocol functionality crosscuts the library management functionality, it is implemented as an Aspectual
Collaboration. This allows the definition and encapsulation of the different concerns (subject-observer protocol and library management functionality) in separate modules. Figure 4 illustrates how the design of the final application looks like using UFA.

**Figure 4: UFA diagram**

The notation extends UML packages to encapsulate complex behavior. UFA packages have more semantics than standard UML packages as they can, in addition to classes and interfaces, contain attributes and methods. In UFA, packages are first class entities. The basic library management functionality is designed within the Library package; it contains the classes BookCopy, and BookManager. The library management application is completely unaware of the crosscutting subject-observer protocol functionality being applied to it. The Observer pattern is designed within the Observer package; it contains the classes (roles) Subject and Observer. This package is marked as (abstract) as is contains abstract methods that can only be bound to concrete methods when using the pattern in the context of a certain application. In this case the update() operation of class Observer is the incomplete part. Its behavior cannot be defined at this generic level, i.e. what exactly has to be done whenever a notification occurs depends on the application the pattern is applied to. This leads to a highly reusable design of the subject-observer protocol functionality as all the application dependent parts remain abstract until binding time. The connector package ObserveLibrary specifies how the generic Observer pattern is applied to the library management application. The connector adapts the core (package Library), indicating that it may use and influence it, and refines the abstract package (package Observer), indicating that abstract parts are made concrete. Both, abstract package and core package are unaware of the adaptation and refinement respectively, only the connector holds the information to connect both worlds.

The classes inside the connector package are enhanced UML classes and support the adaptation of the abstract collaboration to a certain application. The class identifier field of these special classes is used to express which role of the abstract collaboration is played by which class of the core application, e.g. see Figure 4 where role Subject is played by class BookCopy (indicated by Role=BaseClass). Two types of method bindings are available in UFA, namely advice weaving where the behavior of the core application is modified and delegation binding where behavior of the core application is used to fill the abstract parts of the collaboration. Advice weaving is called callin in UFA as the core application calls into a role method (indicated by ←). In the example the method notify() of role Subject is called after the methods borrowCopy() or returnCopy() of class BookCopy (indicated by notify ← after {borrowCopy, returnCopy}). Delegation binding is called callout in UFA as role classes use external behavior in order to fill their incomplete parts (indicated by →). In the example the method update() of role Observer delegates to the method updateStatus() of class BookManager (indicated by update → updateStatus). For a more detailed description of the syntax and semantics of UFA see [6].

UFA supports all properties of the Aspectual Collaborations model as described above but is still lacking tool support. The notation is missing standard UML conformity which prevents developers from using modeling tools that are based on UML. In order to make modeling with Aspectual Collaborations and UFA more convenient we developed sUFA (standard UML for Aspects), a notation that is based on UFA but standard UML compliant. Developers can now gain all the benefits they are used to in OO design; they can do their design using familiar tools and environments, interchange tools and benefit from code generation facilities. Figure 5 shows how the example presented above is designed using sUFA.

**Figure 5: sUFA diagram**

As can be seen in Figure 5, we used UML stereotypes, one of UML's standard extension mechanisms. The relationships between the packages are dependency relationships stereotyped with extend and adapt respectively. The abstract collaboration package is stereotyped with abstract and the connector
Delegation binding is supported by stereotyping a method with "callout". In class Observer=BookManager the first method indicates that the method update() of class Observer delegates to the method updateStatus() of class BookManager. Again, the arrow used in UFA is substituted by the "|" symbol.

The next section gives an overview of the code generation facility that generates code for the component (feature) integration from the graphical models.

4. CODE GENERATION

Model-Driven Software Development (MDSD) is about developing executable software from higher-level models [8]. Models (either graphical or textual) are specifications that cannot be directly executed and therefore have to be transformed to more concrete models and/or finally code. The entities within the model are independent from any implementation technology or target platform and are usually specified together with domain experts. Such models form Domain Specific Languages (DSLs) and provide a higher level of abstraction than general purpose languages and offer a domain-specific syntax and also domain-specific error messages. MDSD helps automating the development process which increases efficiency, reuse, and software quality.

We provide an MDSD approach for pluggable collaborations in a way that developers only have to (graphically) specify how components are linked together and all the "glue" code is created automatically. For pluggable collaborations this means that developers use sUFA as a modeling language to link components and all the bindings are generated automatically. Here we act on the assumption that all the components exist (i.e. which is typically the case for base assets in software system family development) and only need to be linked together. If needed it would be straightforward to generate skeleton implementations for the components as well. For our purposes we use Enterprise Architect [9] as modeling tool and openArchitectureWare as generator framework [10].

The openArchitectureWare tool platform is an open source MDD generator framework that is implemented in Java and supports arbitrary input formats, metamodels, and output formats. The framework is well integrated into eclipse and offers a variety of features and support for MDSD. For a detailed description of all tool components please see [10]. Figure 6 provides an overview of the core functionality of openArchitectureWare. First, a metamodel of the domain has to be defined and implemented as Java classes (metaclasses). Second, a model has to be created (an instance of the metamodel) which can be done using for example UML tools, textual languages or XML. The generator parses the respective model and builds a syntax tree in memory by instantiating the metaclasses. In order to create the output (e.g. code) templates have to be written. The generator then navigates through the syntax tree and performs the actions described in the templates.

Figure 7 provides an overview on how the modeling and code generation process works. The first step is the development of an sUFA model using Enterprise Architect and its later export to the XMI format [11]. Even though one of the goals of XMI is to provide interoperability and standardized data exchange between various UML modeling tools, the XMI export output differs between design tools. To overcome this problem we transform the XMI model into an XML format we specifically designed for our purpose. This means that when the modeling tool should be changed (e.g. from Enterprise Architect to Rational Rose [12]), only this transformation has to be adapted. Everything further down the code generation chain remains the same. The generation process can even be started using the XML model only, i.e. without a graphical representation of the sUFA model and by writing the XML file by hand. After the transformation is done, the model is validated for syntactic and semantic correctness. Syntactic correctness is checked using the Document Type Definition (DTD) for our self defined XML format. If this step is passed successfully, the following semantic checks are done:

- Do all the roles and bound classes exist within the collaboration and the core respectively?
- Do all the methods used in callins exist?
- Do all the methods used in callouts exist?
- Are all abstract methods of the collaboration bound to concrete methods of the core, i.e. do they delegate to concrete methods?
The generation process (transformation, validation, generation) is controlled by Ant scripts [13] and code generation only starts when the validation part finishes successfully. To provide code generation, we implemented the metamodel of sUFA as Java classes and wrote templates for the bindings. Code generation is available for both CaesarJ and ObjectTeams by providing different templates for both languages. Figure 8 shows the binding code that is generated for the CaesarJ version of the library example.

```java
abstract public cclass ObserveLibrary extends ObserverProtocol {
    public cclass Subject wraps BookCopy {
        public BookCopy getWrappee() {
            return wrappee;
        }
    }
    public cclass Observer wraps BookManager {
        public void update(SubjectRole s) {
            wrappee.updateStatus(s.getWrappee());
        }
    }
    after(BookCopy bc): (call(public void borrowCopy()) || call(public void returnCopy()))
        && target(bc) {
            Subject(bc).notify();
        }
    after(BookManager bm, BookCopy bc):
        (call(public void buy(BookCopy))
            && target(bm) && args(bc) {
            Observer(bm).start(Subject(bc));
        }
    after(BookManager bm, BookCopy bc):
        (call(public void drop(BookCopy))
            && target(bm) && args(bc) {
            Observer(bm).stop(Subject(bc));
        }
}
```

Figure 8: Generated Binding

Our approach shows how MDSD can be used together with pluggable collaborations to support the product instantiation process for product families. Existing components (base assets) can easily be connected to form a specific product using the graphical notation sUFA and an arbitrary modeling tool. Users can even specify how components and application are linked together only by using XML, they just have to follow a defined DTD. Abstract collaborations can easily be used in a different context, it is just the connector that has to be changed and all the binding code is generated automatically. This is especially useful within software product line development. Features could be implemented as collaborations and linked together using our approach in order to create a concrete product.

5. RELATED WORK

Many different aspect-oriented design modeling approaches have been presented in the recent years [16]. Some of them are based on class level decomposition which is not suitable for the use within software system family development. Features typically implement complex functionality that spans several parts of an application and therefore package level decomposition is more appropriate. Composition Patterns [14] [15], the most advanced approach in the area of aspect-oriented modeling, focus on package level decomposition and provide mappings for AspectJ and Hyper/J. Evaluating the suitability of Composition Patterns for feature-oriented decomposition and using AspectJ and Hyper/J as implementation technologies is part of our future work. For a detailed study and comparison of aspect-oriented analysis and design approaches we refer to [18].

6. SUMMARY UND FUTURE WORK

Software system family development aims at saving development cost by structured reuse of base assets such as configurable components. These components are very likely to implement functionality that spans several parts of an application. Aspectual Collaborations allow handling these so-called crosscutting concerns by offering modules that support their encapsulation. The system can then be configured by linking the collaborations to form a specific product.

In this paper we have shown how Aspectual Collaborations can be used to design and implement features of a software system family and how MDSD can support product instantiation. We developed a means to graphically specify how components that implement features can be linked together to form a complete system. This small DSL is combined with a generator tool that automatically generates all the glue code necessary to combine the different features.

We are currently working on the development of a Graphical Editing Framework (GEF) [17] editor that supports the UFA design notation presented in [6]. This will avoid having to switch between tools and constraints can already be checked at modeling time. The editor will provide the same code generation facilities as
presented in this paper including a model export and import facility. Future work also includes the ability to specify constraints that ensure that only valid feature combinations are created. This means that for example features conflicting with each other can not be combined. These constraints can be checked at modeling time and domain specific error messages can be provided.

7. ACKNOWLEDGMENTS
We thank Stephan Herrmann for his very valuable comments and feedback about sUFA.

8. REFERENCES