A UML Profile for Delta-Oriented Programming to Support Software Product Line Engineering

Maya R. A. Setyautami
Fakultas Ilmu Komputer
Universitas Indonesia
mayaretno@cs.ui.ac.id

Reiner Hähnle
Dept. of Computer Science
Technische Univ. Darmstadt
haehnle@cs.tu-darmstadt.de

Radu Muschevici
Dept. of Computer Science
Technische Univ. Darmstadt
radu@cs.tu-darmstadt.de

Ade Azurat
Fakultas Ilmu Komputer
Universitas Indonesia
ade@cs.ui.ac.id

ABSTRACT

Feature-based approaches to software design, like delta-oriented programming, are well-suited to support multi-product software development paradigms, such as Software Product Lines. Currently, the popular UML notation does not support delta-oriented software design, so that several ad-hoc notations tend to be used. This paper presents a systematic approach to import concepts from delta-oriented programming into the mainstream notation UML. This is done with minimal overhead by specifying a new, slim, delta-oriented UML profile. It is compatible with languages that support delta-oriented programming such as DeltaJ and ABS. The usefulness of the profile is evaluated with a case study.

CCS Concepts

•**Software and its engineering** → *Software design engineering;*

Keywords

Unified modelling language; Delta-oriented programming

1. INTRODUCTION

The modeling of product *variability*—identifying and managing the commonalities and differences among software products—is a key issue of any software product line (SPL) development process. Variability modeling spans all phases of SPL development, from analyzing the problem by gathering requirements, to design and implementation of a solution in the form of software. Variability modeling is thus a concern in both *problem space* and *solution space*.

In the problem space, variability is typically modeled by abstracting requirements to *features* and describing dependencies between features in a *feature model*. In the solution space, features are mapped to implementation artifacts. This is often done in an ad-hoc manner and implicitly by using `#ifdef`s and similar conditional compilation concepts. More systematic support on the language level is provided by feature-oriented programming (FOP) that implements features using *feature modules* [1]. A more recent approach is *delta-oriented programming* (DOP), an object-oriented language concept that allows a flexible "n-to-m" mapping of features to *delta modules* (or *deltas* for short): a feature can be implemented using multiple deltas, while a delta can supply the (partial) implementation for multiple features.

A standard approach for specifying and visualizing the design of software systems is the widely used Unified Modeling Language (UML). At present UML does not include modeling elements for specifying the structural variability of a software design. While extensions to cover feature-oriented design have been put forward [8, 11], such extensions only enable variability modeling at a very abstract level in the problem space. Critically, they provide no connection to the model elements used in designing the solution, such as UML class diagrams. This makes it hard to realize a key ingredient of any modern SPL development processes: *traceability* between problem space and solution space. To remedy this situation, in this paper we propose a systematic connection between delta-oriented programming and UML structure diagrams. This will take the form of a UML profile that comes with minimal overhead.

2. DELTA-ORIENTED PROGRAMMING

Delta-oriented programming is a feature-oriented programming paradigm suitable for the development of Software Product Lines [9]. It constitutes a two-tier approach for addressing product variability: First, so-called code deltas are associated with a feature they are supposed to implement. A delta is a set of syntactic code changes that specify in a structured manner how existing code has to be modified to implement a given feature. Second, to obtain from a given program P a new version P_f that supports feature f, one applies to P the code deltas associated to f. The resulting "flat" standard program P_f realizes feature f. Delta application is performed by a dedicated compilation step, which makes sure that the resulting *product* P_f is well-typed and the delta application sequence satisfies possible ordering constraints. The ability to apply several deltas consecutively makes the

SPLC '16, September 16-23, 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2934466.2934479

approach compositional. For details, see Schaefer et al. [9].

In contrast to deltas, conventional OO languages often encode feature-driven variability with the help of class inheritance. But combining both, variability and functional aspects in one and the same code base, tends to result in code that is hard to understand and maintain. The ability to clearly separate feature design and functional design aspects is an advantage for the development of feature-rich software [4]. Another benefit is that the product obtained after delta application contains exactly the code required to implement the requested features and none else.

2.1 Delta Oriented Programming with ABS

The ABS language [6] is an object-oriented, concurrent, executable modeling language. The full specification of its syntax and semantics is available[1]. ABS provides language constructs and tools for modeling SPLs using the DOP approach. Specifically, it supports feature modeling at the design stage, while it provides DOP at the level of code.

An SPL is implemented in ABS as a set of core modules (the *core*, cf. Section 3.1) and a set of delta modules that modify the core. *Delta modules* represent the variability of an SPL at the implementation level. ABS delta modules modify a core module by adding, modifying or removing program elements, including attributes, classes, methods and interfaces (cf. Section 3.3 for an example).

Features and deltas in ABS are connected in such a way that features can be easily traced to the delta modules that provide their implementation, and vice versa, via a *product line configuration* (cf. Section 3.4). Each feature can be implemented using one or more deltas, while each delta may contribute to the implementation of one or more features. A product line configuration provides the information to determine which deltas will be applied in which order to the core, whenever a set of desired features is selected.

A set of selected features that satisfy the constraints mandated by the feature model is called a *product*. The result of a sequence of delta applications to a core mandated by a given product is called a *software product*.

2.2 Delta-Oriented Programming with DeltaJ

DeltaJ [7, 9] is a Java-based programming language that supports DOP. It is similar to ABS in scope and syntax, with core modules, delta modules and product generation.

The latest version of DeltaJ [7] does not require a core; product generation relies only on the composition of delta modules. Deltas may add, remove, or modify classes by modifying their methods and attributes. The feature declaration and a set of valid feature configurations are contained in a *delta-oriented product line* module. For each delta one must specify the features that require its presence.

Product generation is similar as in ABS by composing all delta modules requested by a feature selection. The first delta is applied to the empty program, because there is no core module, then the second delta is applied to the outcome of the first delta application, etc.

3. THE UML-DOP PROFILE

As a modeling language that is widely used in software development, UML has a standard syntax and semantics. However, sometimes UML syntax and semantics are not sufficient to express a specific system concept in a particular domain (e.g., real-time, business process modeling, finance). One approach provided by the OMG to overcome this problem is UML *profiles* [5]. These can be used to customize UML syntax for a specific domain or programming language.

A UML profile is defined by a set of extension mechanisms that permit customization: *stereotypes*, *tagged values*, and *constraints* (see http://www.omg.org). A stereotype is a class type that extends another UML class with a specific mechanism and that must be used together with its extension class. A stereotype class has properties or attributes, called tagged values.

We define a UML profile, called UML-DOP, to capture delta-oriented concepts in UML notation. Each syntax element of the static design view of DOP extends a UML meta class and is mapped to stereotypes, tagged values, and constraints. Although the profile is defined based on ABS and DeltaJ, for simplicity we use ABS for examples of the profile application. There is an important advantage of basing the UML-DOP profile on the DOP approach: as there is a one-to-one mapping from DOP elements to UML model elements, there is a deterministic translation from ABS/DeltaJ code stubs to UML model elements. Vice versa, UML (with the UML-DOP profile) can encode product variability in exactly the same efficient manner as it is possible in DOP.

3.1 Core Modules

ABS specifies *core* behavioral modules based on object-oriented modeling. Each object of a system is an instance of a class and each class must implement one or more interfaces. Core modules declare a list of model elements for export and import to regulate access from and to other modules [6]. The following code snippet is an example of an ABS core module in the AISCO case study (see Section 4).

```
module Program;
interface Program {
  Program getProgram();
  Unit setProgram(String name, Int amount);
}
class ProgramImpl(Int id, String name, Int amount)
implements Program {
  Program getProgram() { return this; }
  Unit setProgram(String name, Int amount) { ... }
}
```

Mapping to UML

We extend the UML class definition in the UML-DOP profile to cover properties specific to DOP. A class in DOP has mostly the same semantics as a UML class. Hence, an ABS class maps to a UML class. However, we must account for the differences between ABS and UML classes. For example, ABS has class parameters that initialize the class attributes with given parameter values. As there exist no class parameters in UML, we map ABS class parameters to a suitable constructor method in UML: for each ABS class declaration with a non-empty parameter list $\overline{p}$ we create a public constructor method in UML whose name is the same as the ABS class name and with parameter list $\overline{p}$. In addition, private attribute declarations for the elements of $\overline{p}$ are added to the UML class.

Classes in ABS implement one or more interfaces and are declared inside a module. ABS modules behave similarly to UML *packages* that have export and import lists. Hence

[1] http://tools.hats-project.eu/download/absrefmanual.pdf

we map an ABS module to a UML *package* with stereotype «module». The **export** and **import** directives of ABS modules are mapped to UML dependencies stereotyped «export» and «import», respectively. Figure 1 shows the UML Diagram resulting from mapping the core ABS module Program based on the stereotypes in the UML-DOP profile.

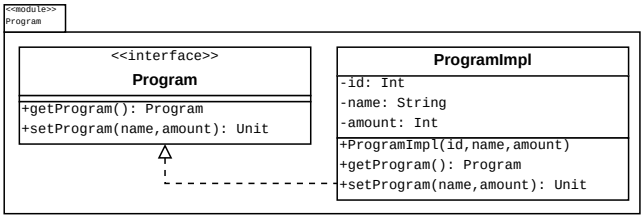

Figure 1: UML representation of ABS core module

3.2 Feature Modeling

An ABS feature model is defined using a textual variability language that organizes features in a tree structure similar to feature diagrams. A feature model specifies properties for each feature. It also specifies dependencies among features through grouping, as well as "require"/"exclude" constraints. The following code snippet is part of the ABS feature model for the AISCO case study (cf. Section 4).

```
root AISCO {
  group allof {
    ProgramData {
      group [1..*] {
        opt Periodic, opt Continuous, Eventual } },
    opt DonationData {
      group [0..*] { Money, Item, Confirmation } }, } }
```

In this example a mandatory feature ProgramData has two optional child features, Periodic and Continuous, and one mandatory feature Eventual. There is also an optional feature DonationData that can have one or more child features Money, Item and Confirmation. The group constraint for the root feature AISCO is **allof**, meaning all elements of the group must appear. The group constraint for feature ProgramData has cardinality [1..*] implying that one can choose one or more child features, and the constraint for DonationData has cardinality [0..*], which is equivalent to making all child features optional.

Mapping to UML

We map a feature to a UML *component* with stereotype «feature», because a feature is more abstract than a class and has various dependencies. The implementation of a feature is given by delta modules, as explained in Section 3.3.

The group constraint of a feature is given as a *cardinality* [3] (e.g. **allof**, **opt**, [1..*]) that specifies the number of child features that can be selected within the group. We map a feature group to a UML *port* with a cardinality *property* of type string. The relation between parent features and child features is mapped to a UML *dependency* with stereotype «optional» or «mandatory». Additional relations (constraints) among feature in ABS (for example **require** and **exclude**) are mapped to a UML *dependency* with corresponding stereotype (for example «require» and «exclude»).

Figure 2 shows the UML diagram that results from mapping the ABS feature model in Section 3.2 to UML based on the stereotypes defined above.

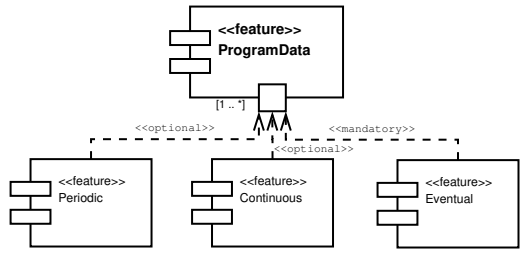

Figure 2: UML representation of ABS feature model

3.3 Delta Modeling

ABS delta modules modify a set of ABS core modules. The following code is an example of a delta module named DEventual. It modifies class ProgramImpl to add a new attribute end_date and a method setEndDate.

```
delta DEventual;
modifies class Program.ProgramImpl {
  adds String end_date = "";
  adds Unit setEndDate (String e) { end_date = e; }
}
```

Mapping to UML

We extend UML to represent the central concept of DOP, delta modules. A delta describes a set of changes to a given ABS core modules in order to (partially) implement one or more features. A delta module can modify multiple classes and interfaces. Hence we map a delta module to a UML *package* with stereotype «delta». It consists of one or more modified classes or modified interfaces that has an association with the original class. The modified class is mapped to a UML class with stereotype «modifiedClass» and the modified interface is mapped to a UML interface with stereotype «modifiedInterface». The association between the original class/interface and the modified class/interface is stereotyped «adds», «removes» or «modifies», depending on the type of modification.

The *modifier* that describes how a certain element of the class/interface is modified is mapped to a UML *property* (for attributes) or UML *operation* (for methods) and has one of the stereotypes «adds», «removes» or «modifies». For example, if a new attribute is added to the modified class, there will be a UML *property* with stereotype «adds» representing that attribute.

Figure 3 shows the UML diagram that represents the ABS delta module DEventual. The delta module is represented as a UML *package* DEventual with stereotype «delta». The class ProgramImpl modified by the delta adds a new attribute and a method. This is represented by the modified class ProgramImpl with the end_date attribute stereotyped with «adds». The new method is represented by the operation *setEndDate* with stereotyped «adds».

3.4 Product Line Configuration

A *product line configuration* specifies the (*many-to-many*) relation between *features* in feature modeling and *deltas* in delta modeling. Based on the set of features in a specific product, the configuration defines which delta modules

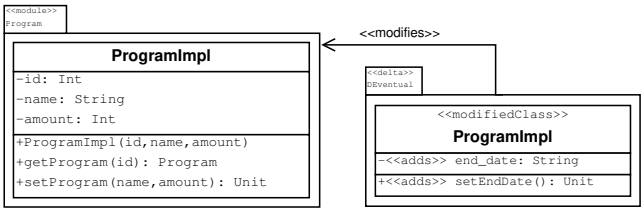

Figure 3: UML representation of an ABS delta

should be applied. In ABS the declaration starts with the name of a product line, followed by the list of features and a configuration for each delta module [2].

The code snippet below is part of a product line configuration for the AISCO product line (cf. Section 4). The **when** clause specifies an *application condition* [9]. For example, the delta `DEventual` is applied whenever the `Eventual` feature is selected in a product. Our example has only one-to-one relations between deltas and features. In general, a feature may trigger application of more than one delta and some deltas might be applied only for a combination of features. It is also possible to specify a partial order on the application of deltas using an **after** clause (not shown here).

```
productline AISCO;
delta DPeriodic when Periodic;
delta DEventual when Eventual;
delta DContinuous when Continuous;
```

Mapping to UML

In Sections 3.2 and 3.3 we mapped features to UML *components* and ABS delta modules to UML *packages*. Product line configurations define a link between features and deltas. Hence we represent a product line configuration as a UML *dependency* between a UML *component* with stereotype `«feature»` that represents the feature, and a UML *package* with stereotype `«delta»` that represents the delta module. This UML *dependency* has stereotype `«when»` to indicate the application condition. If more than one delta is applied for a feature, we can specify the order using a UML *dependency* between two deltas with stereotype `«after»`. Figure 4 shows a UML diagram that models part of the product line configuration above.

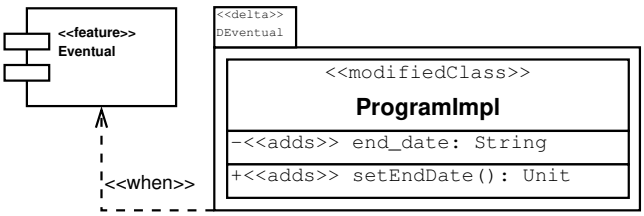

Figure 4: UML representation of ABS product line configuration

3.5 Product Selection

An ABS product selection defines product variants based on the features that they include. The AISCO case study (cf. Section 4) defines several product variants; the `MUI` product shown below represents one such variant.

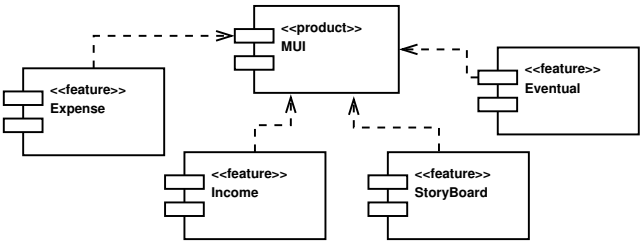

Figure 5: UML representation of AISCO product

```
product MUI (Eventual, StoryBoard, Income, Expense);
```

The set of requested features is declared after the product name in parentheses. For example, the `MUI` product implements the features `Expense`, `Income`, `StoryBoard` and `Eventual`.

Mapping to UML

In the UML-DOP profile we have to represent the relation between products and features specified by a product selection. A product has one or more features and a feature can be selected for multiple products. In Section 3.2 we mapped features to UML *components*. Because a product is essentially a set of features, we represent a product as a UML *component* with stereotype `«product»`. The features implemented within that product are represented by a UML *dependency* to that component. Figure 5 shows the UML representation of the `MUI` product declared above.

3.6 UML-DOP Profile Summary

A summary of the mapping between some DOP elements and stereotyped UML elements of the UML-DOP profile is shown in Table 1. The UML extensions are characterized by their stereotype names. The DOP elements Export Module, Import Module, Optional, Mandatory, Require, Exclude, When, After are mapped to UML Base Class Dependency with obvious stereotype names `«export»`, etc; elements Feature and Product are mapped to UML Base class Component.

DOP Element Name	UML Base Class	Stereotype Name
Module	Package	`«module»`
Delta Module	Package	`«delta»`
Delta Parameter	Property	`«deltaParam»`

Module Modifier	Association
	
	Class
	Interface
		`«adds»`, `«removes»`
		`«modifies»`
		`«modifiedClass»`
		`«modifiedInterface»`

Modifier	Operation;
	Property
		`«adds»`
		`«removes»`
		`«modifies»`

Table 1: Map of DOP elements to UML stereotypes

4. EVALUATION

We evaluate the completeness of our UML-DOP profile by applying the profile to an delta-oriented SPL modeled in ABS. We chose the "Adaptive Information System for Charity Organizations" (AISCO), a software system that helps

charity organizations to publish their activities and to generate financial reports. By applying the UML-DOP profile, a design model of the entire AISCO SPL is obtained, of which a fragment is displayed in Figure 6. It shows the core `Program` module, containing the `Program` interface and an implementing class. Product selection is illustrated with the product `MUI` that has four features. Among these, the `Eventual` feature is implemented by applying the delta `DEventual`, which adds a new attribute and method to the `ProgramImpl` class. A more detailed description of the case study is part of the technical report accompanying this paper [10].

The UML-DOP profile makes it possible to represent the delta-oriented design of the entire AISCO product line. Feature variability, feature implementation, the modifications implemented by deltas, as well as product selection are all clearly visible in a single UML diagram.

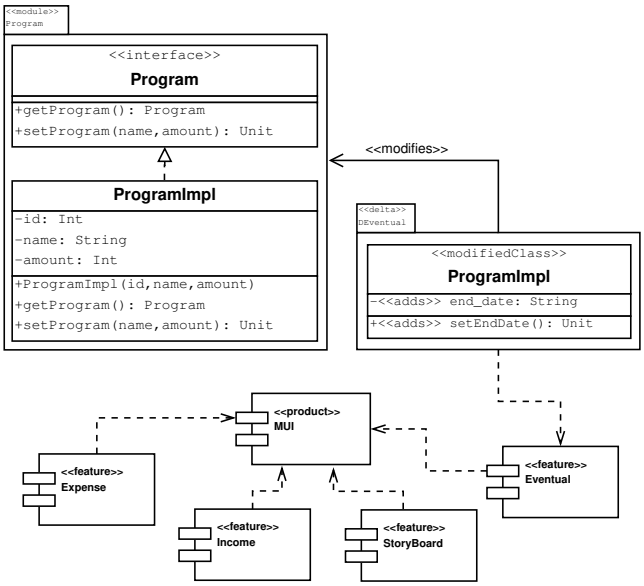

Figure 6: AISCO UML-DOP diagram

5. CONCLUSION AND FUTURE WORK

We defined a UML-DOP profile that permits to represent software product line variability using the popular UML notation. By reusing stereotyped UML elements all delta-oriented programming elements can be represented in UML. In the profile these stereotypes extend the UML metaclasses *Class*, *Interface*, *Component*, *Package*, *Dependency*, *Association*, *Property*, and *Operation*.

The proposed profile is compatible with current implementations of DOP, the ABS modeling language and the DeltaJ extension of Java. The grounding of our suggested profile in actual programming languages has the advantage that one can connect the diagrams with executable code. Hence, our UML-DOP profile is more than a mere visual design notation, because it reflects precisely the structure of the underlying implementation. This is a suitable basis for end-to-end (feature model to executable code) modeling in SPL development and for round-trip engineering of SPLs. These are topics we would like to explore in the future.

Our UML-DOP profile can be used as a basis for transformation rules from standard UML designs to feature-oriented designs expressed with DOP elements and back. We intend to support automatic translation between standard UML and UML-DOP using Text-to-Model transformation. Of specific interest are refactoring rules that can be applied to transform a legacy system into a feature-based DOP design.

Acknowledgments

Research partly funded by the EU project FP7-ENVISAGE No. 610582, see http://www.envisage-project.eu. Thanks to Crystal Chang Din for her valuable feedback on earlier drafts. Thanks to members of RSE Lab Fasilkom UI for their support in identifying the case study.

6. REFERENCES

[1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE TSE*, 30:355–371, 2004.

[2] D. Clarke, N. Diakov, R. Hähnle, E. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.

[3] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Software Product Lines*, pages 162–164. Springer, 2004.

[4] G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. de Almeida Maia. On the use of feature-oriented programming for evolving software product lines. *Sci. Comput. Program.*, 93:65–85, 2014.

[5] L. Fuentes-Fernandez and A. Vallecillo-Moreno. An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, V:6–13, 2004.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects FMCO)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.

[7] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. DeltaJ 1.5: Delta-oriented programming for Java 1.5. In *Principles and Practice of Programming in Java*, PPPJ '14, pages 63–74, New York, NY, USA, 2014. ACM Press.

[8] T. Possompès, C. Dony, M. Huchard, and C. Tibermacine. Design of a UML profile for feature diagrams and its tooling implementation. In *International Conference on Software Engineering & Knowledge Engineering*, SEKE'2011, pages 693–698, Miami Beach, USA, 2011. Knowledge Sys. Inst.

[9] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Intl. Software Product Line Conf.*, SPLC '10, pages 77–91, Berlin, Heidelberg, 2010. Springer.

[10] M. R. A. Setyautami, R. Hähnle, R. Muschevici, and A. Azurat. A UML profile for delta-oriented programming to support software product line engineering. Technical Report TUD-CS-2016-0100, Technische Universität Darmstadt, 2016.

[11] V. Vranic and J. Snirc. Integrating feature modeling into UML. In *Net Object Days/GSEM*, LNI, pages 3–15, Erfurt, Germany, 2006. GI.