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

Maya R. A. Setyautami[1], Reiner Hähnle[2], Radu Muschevici[2], and Ade Azurat[1]

[1] Fakultas Ilmu Komputer
Universitas Indonesia
Depok, Indonesia
{mayaretno|ade}@cs.ui.ac.id

[2] Department of Computer Science
Technische Universität Darmstadt,
Darmstadt, Germany
{haehnle|radu}@cs.tu-darmstadt.de

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.

1 Introduction

The modeling of product *variability*—to identify and to manage the commonalities and differences among software products—is a key issue of any software product line (SPL) development process [14]. 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 the *problem space* and the *solution space.*

In the problem space, variability is typically modeled by abstracting requirements to *features* and describing dependencies between features in a *feature model* [10]. 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* [3, 16]. A more recent approach is *delta-oriented programming* (DOP), an object-oriented language concept that allows a flexible

[⋆] Technical Report TUD-CS-2016-0100, `https://www.se.tu-darmstadt.de/publications/details/?tx_bibtex_pi1[pub_id]=TUD-CS-2016-0100` "n-to-m" mapping of features to *delta modules* (or *deltas* for short): a feature can be implemented using multiple delta modules, while a delta module 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 [15, 20], 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. Moreover, in the solution space the distinction between architectural design and variability modeling is blurred, because UML modeling elements such as inheritance relations or associations are used for either purpose. 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 is very lightweight and comes with minimal overhead.

The remaining paper is organized as follows. In the following section we motivate our design choices and highlight the advantages of our approach in the context of SPLE. To make the paper self-contained, we briefly describe DOP as well as two of its realizations (ABS and DeltaJ) in Section 3. Our delta-oriented UML profile, described in Section 4, is called UML-DOP and can be seen as a static design based on the delta-oriented languages ABS and DeltaJ, in a similar manner as UML is a static design view on OO languages like Java or C++. In Section 5 we demonstrate the viability of our approach by applying it to a case study modeling an Adaptive Information System. We review related work in Section 6, and conclude, as well as outline future work, in Section 7.

2 Motivation

Among several available feature-oriented programming paradigms [4, 16, 17] we chose delta-oriented programming [17] as the basis of our work for the following reasons: first, DOP is an object-oriented paradigm compatible with mainstream programming languages and development processes; because of this, it was possible to design a natural UML profile with low overhead for the DOP extension of UML; second, even though DOP is a relatively recent paradigm, there exist two implementations: ABS [8, 9], for *Abstract Behavioral Specification*, is a mature, executable modeling language that supports DOP; DeltaJ [11] is an extension of Java that fully integrates delta modules into Java's package system.

A key issue we want to address in this work is *traceability* between problem and solution space in an SPL development process. For delta-oriented SPLs this means that the variability model includes information about which delta modules implement a certain feature, and what features are implemented by a certain delta module. An advantage of traceability between feature and delta model is automation: upon selecting a set of features, the corresponding software product can be generated automatically by tracing the corresponding delta modules and composing them.

Designing the variability model of an SPL using DOP is a creative process, comparable to, e.g., object-oriented design. The task essentially amounts to providing a set of reusable delta modules that together implement the features of the SPL, and can be composed in various ways to obtain all products of the SPL. A visual design language to aid this process is highly useful but has been missing so far from the delta-oriented SPL development process.

3 Delta-Oriented Programming

Delta-oriented programming is a feature-oriented programming paradigm suitable for the development of Software Product Lines [17]. 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. In an OO framework the typical granularity, which is also supported by the ABS and DeltaJ languages, are deltas for attributes, operations, and classes. 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 approach compositional. For details, see Schaefer et al. [17].

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 [6]. Another benefit is that the product obtained after delta application contains exactly the code required to implement the requested features and none else.

It is important to point out that even software not architected as a SPL benefits from DOP: any non-trivial software project involves differently instrumented code variants for debugging and testing purposes. It is highly advantageous to be able to maintain these separately.

3.1 Delta Oriented Programming with ABS

The ABS language [9] is an object-oriented, concurrent, executable modeling language. It supports modeling at the level of data types, functions, imperative statements and objects. ABS has interfaces, interface inheritance, and classes, but no code inheritance, hence no abstract classes; instead, delta composition as outlined above is available as a mechanism of code reuse.

ABS comes with a compiler and tool set that includes code generation backends (Java, Erlang and Haskell), test case generators, as well as various static analysis tools [21]. It has been successfully used in industrial projects [2]. The full specification of ABS's syntax and semantics is available in the ABS Language Specification [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*) and a set of delta modules that modify the core. The core typically implements a basic functionality common to all (or most) products of the SPL (cf. Figure 1 for an example). *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. Figure 5 for an example).

Determining what goes into the core and what functionality is implemented using deltas is part of the design process of an SPL and therefore up to the ABS modeler. A core can be even empty and all functionality may be contained in deltas.

In addition to core modules and deltas for code modularization, ABS supports feature-oriented SPL design. Features have a corresponding implementation which in ABS is described by deltas. Feature models are denoted in ABS using the textual variability language μTVL, a variant of feature diagrams [19] (cf. Figure 3).

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. Figure 7). 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*. Figure 10 shows two such products declared in the ABS language. The result of a sequence of delta applications to a core mandated by a given product is called a *software product*.

3.2 Delta-Oriented Programming with DeltaJ

DeltaJ [11, 17] 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 [11] 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. If the delta modules are not applicable in the requested order, product generation fails.

4 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, etc.). The Object Modeling Group (OMG) provides several approaches to overcome this problem. One of them are UML *profiles* [7] that can be used to customize UML syntax for a specific domain or programming language. Several programming languages and frameworks, including CORBA, CCM, CCCMP, CCA, EJB, Java [12], already have a UML profile.

A UML profile is defined by a set of extension mechanisms that permit customization: *stereotypes*, *tagged values*, and *constraints* [13]. 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. Optional constraints may impose limitations on their usage.

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, to simplify the explanation, we explain the profile definition with ABS.

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 and easy-to-maintain manner as it is possible in DOP.

4.1 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.

Each class declares methods and attributes. A core module in ABS syntax is similar to a package in Java syntax, but disallows inheritance and overloading.

Multiple inheritance is allowed at the interface level [9]. There is an optional main method which is declared between two braces. Main methods are executed by default when running an ABS model. The following code snippet (Figure 1) is an example of an ABS core module called `MProgram` that models a program[3] in the AISCO case study (see Section 5).

```
module MProgram;
export *;

interface Program {
  Int getId();
  String getName();
  Int getAmount();
  Program getProgram();
  Unit setProgram(String n, Int a, String d, String desc, String c);
}

class ProgramImpl(Int id, String name, Int amount) implements Program {
  String date = "";
  String description = "";
  String contact = "";
  Int getId() { return id; }
  String getName() { return name; }
  Int getAmount() { return amount; }
  Program getProgram() { return this; }
  Unit setProgram(String n, Int a, String d, String desc, String c) {
    name = n; amount = a;
    date = d; description = desc; contact = c;
  }
}
```

Fig. 1. The ABS core module `MProgram`

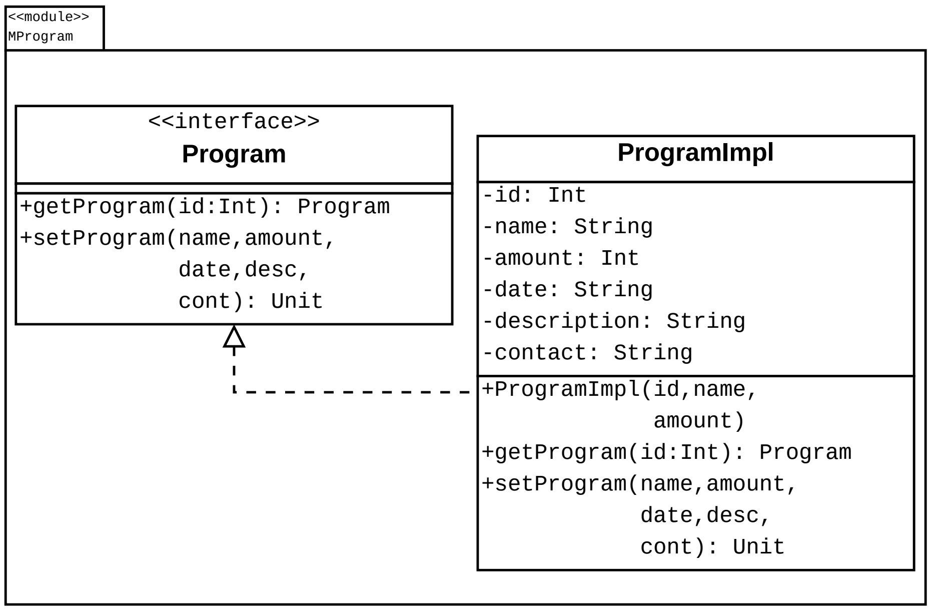
Mapping to UML We need to extend the UML class definition in the UML-DOP profile to cover properties specific to DOP. A class in DOP has mostly the same semantics as a UML class. Hence, a class in ABS maps to a UML class. However, we must account for the differences between ABS and UML classes, for example, parameters in the class declaration. A class parameter in ABS has the same semantics as a constructor method in UML that initializes the class attributes with the given parameter values. As there exist no class parameters in UML, we map ABS class parameters to a suitable constructor method in UML:

[3] Please note that the identifier `Program` here refers to the usual English sense, “plan” or “outline”, not to the technical meaning of “code” in Computer Science.

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 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 2 shows the UML Diagram resulting from mapping the core ABS module `MProgram` based on the stereotypes in the UML-DOP profile (not all methods are displayed). The UML class `ProgramImpl` represents the class with the same name in the ABS model. It implements interface `Program` and belongs to a package `MProgram` with stereotype «`module`» that represents the ABS module.



<<module>>
MProgram
<<interface>>
Program
+getProgram(id:Int): Program
+setProgram(name,amount, date,desc, cont): Unit
ProgramImpl
-id: Int
-name: String
-amount: Int
-date: String
-description: String
-contact: String
+ProgramImpl(id,name, amount)
+getProgram(id:Int): Program
+setProgram(name,amount, date,desc, cont): Unit

Fig. 2. UML representation of ABS core module `MProgram`

4.2 Feature Modeling

As mentioned in Section 3.1, 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 code snippet in Figure 3 is part of the ABS feature model for the AISCO case study (cf. Section 5). There is a mandatory feature `ProgramData` that 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 `Donation-Data` has cardinality `[0..*]`, which is equivalent to making all child features optional.

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

Fig. 3. An ABS feature model

Mapping to UML The UML standard includes no dedicated diagram type for feature modeling along the lines of feature diagrams. Hence we need to adapt a suitable UML diagram type for the UML-DOP profile. We decided to 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 4.3. The stereotype «`feature`» has a tagged value *isRoot* of type *Boolean* to indicate whether it is at the root of the feature diagram.

The group constraint of a feature is given as a *cardinality* [5] (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 4 shows the UML diagram that results from mapping the ABS feature model in Figure 3 to UML based on the stereotypes defined above. There is a UML *component* `ProgramData` with stereotype «`feature`» that represents the feature `ProgramData` in the ABS model. The same applies to the child features `Continuous`, `Periodic` and `Eventual`. In the ABS feature model, `Eventual` is mandatory and the others are optional, denoted by the `[1..*]` cardinality. The cardinality is mapped to the property of the UML port of the corresponding component.

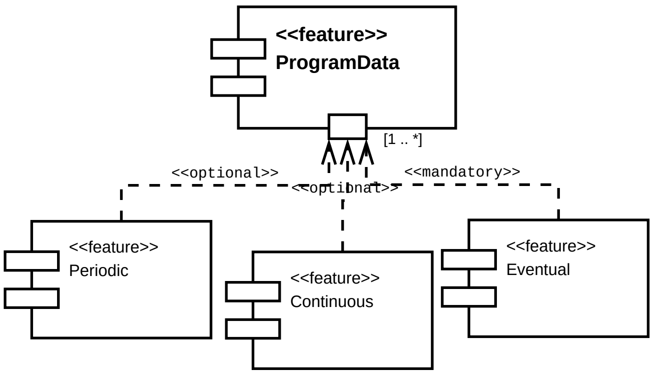


Fig. 4. UML representation of ABS feature model

4.3 Delta Modeling

ABS delta modules modify a set of ABS core modules. As mentioned in Section 3.1, deltas can modify interfaces, classes, methods and attributes. The code in Figure 5 is an example of a delta module named `DContinuous`. It modifies class `ProgramImpl` to add two new attributes `end_date` and `payment`. The delta module also adds a method `setEndDate` and modifies the method `setProgram`.

Mapping to UML We extend UML to represent the central concept of DOP, delta modules. A delta describes a set of changes to some 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 modified 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`».

```
delta DContinuous;
uses MProgram;
modifies class ProgramImpl {
  adds String end_date = "";
  adds Pair<Int, String> payment = Pair(0, "");

  adds Unit setEndDate (String e) { end_date = e; }
  modifies Unit setProgram(String n, Int a, String d,
      String desc, String c) {
    name = n; payment = Pair(a, d);
    description = desc; contact = c;
  }
}
```

Fig. 5. The ABS delta module `DContinuous`

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 6 shows the UML diagram that represents the ABS delta module `DContinuous`. The delta module is represented as a UML *package* `DContinuous` with stereotype «`delta`». The class `ProgramImpl` modified by the delta adds new attributes and a method; it also modifies an existing method. This is represented in the modified class `ProgramImpl` with the `end_date` and `payment` properties, stereotyped with «`adds`». The new and the modified method are represented with the operation *setProgram* stereotyped «`modifies`» and the operation *setEndDate* stereotyped «`adds`».

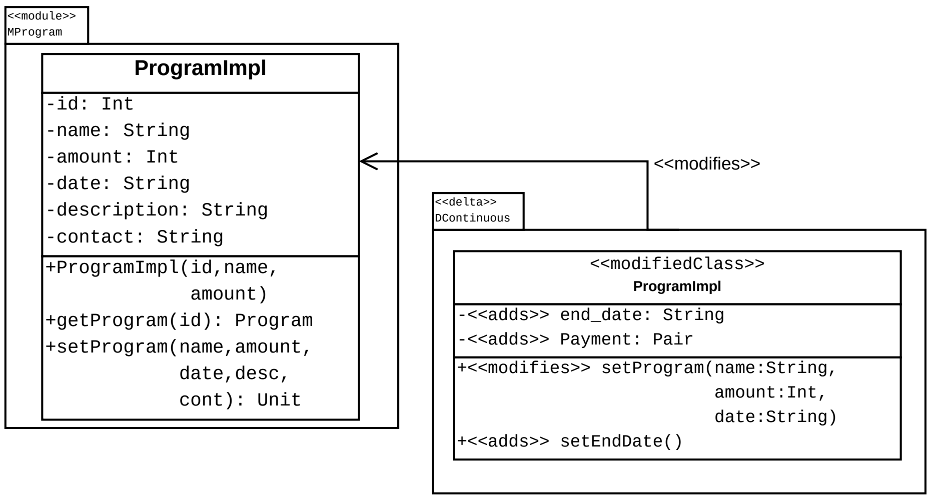


Fig. 6. UML representation of ABS delta module `DContinuous`

4.4 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 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.

The ABS code snippet in Figure 7 is part of a product line configuration for the AISCO product line, which has sixteen features altogether (see also Figure 12). The **when** clause specifies an *application condition* [17]. For example, the delta `DPeriodic` is applied whenever the `Periodic` 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. This is not shown here.

```
productline AISCO;
features ProgramData, Periodic, Eventual, Continuous, PublicationSystem,
  MemberNotification, StoryBoard, AutomaticReport, FinancialReport,
  Income, Expense, Donor, Summary, ObjectiveTarget, Product,
  InstitutionalBeneficiary, IndividualBeneficiary,
  DonationData, Money, Item, Confirmation;

delta DPeriodic when Periodic;
delta DEventual when Eventual;
delta DContinuous when Continuous;
...
```

Fig. 7. An ABS product line configuration

Mapping to UML In Sections 4.2 and 4.3 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 needed to be applied for a feature, we can specify the order using a UML *dependency* between two deltas with stereotype «`after`».

Figure 8 shows a UML diagram that models the product line configuration of Figure 7. It consists of the dependency relationship between the UML *component* `Continuous` with stereotype «`feature`» and the UML *package* `DContinuous`

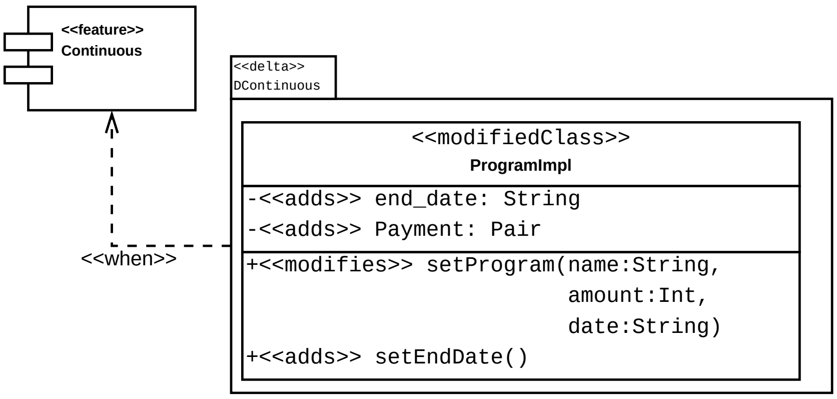


Fig. 8. UML representation of AISCO product line configuration

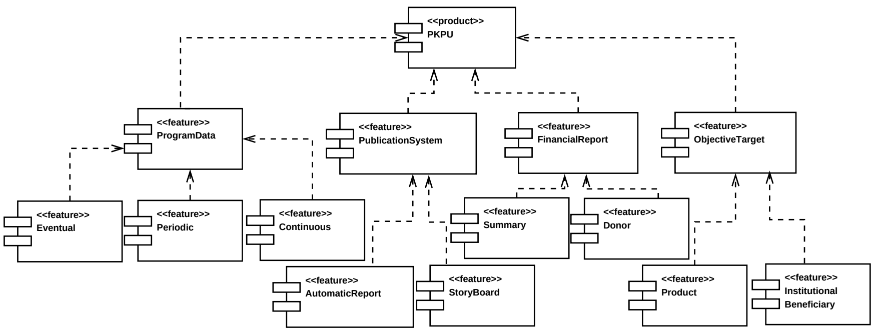


Fig. 9. UML representation of AISCO product `PKPU`

with stereotype «`delta`». Based on the diagram, we can infer that ABS delta `DContinuous` is applied **when** feature `Continuous` is selected.

4.5 Product Selection

A product selection defines product variants based on the features that they include. An ABS **product** declaration starts with the product name and is followed by the list of features that are requested for the product. The ABS compiler checks whether the specified set of features satisfy the constraints defined by the feature model.

The AISCO case study (cf. Section 5) defines several such products; the code snippet in Figure 10 shows the declaration of the `SekolahBermainMatahari` and `PKPU` products. The set of requested features is declared after the product name inside parentheses. For example, the `PKPU` product implements the features `ProgramData` including `Periodic`, `Eventual`, and `Continuous`; `Publication-System` including `StoryBoard` and `MemberNotification`; `FinancialReport` including `Summary` and `Donor`; `ObjectiveTarget` including `Product` and `InstitutionalBeneficiary`.

```
product SekolahBermainMatahari
  (ProgramData, Periodic, Eventual,
    PublicationSystem, AutomaticReport, StoryBoard,
    FinancialReport, Income, Expense, Donor);
product PKPU
  (ProgramData, Periodic, Eventual, Continuous,
    PublicationSystem, StoryBoard, AutomaticReport,
    FinancialReport, Summary, Donor,
    ObjectiveTarget, Product, InstitutionalBeneficiary);
...
```

Fig. 10. An ABS product selection

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 4.2 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 9 shows the UML representation of the product `PKPU` declared in Figure 10. The product is represented as a UML component with stereotype «`product`» having dependencies to the various features (represented by UML *components*) that it implements: `Periodic`, `Eventual`, etc.

4.6 UML-DOP Profile Summary

A summary of the mapping between 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. A visualization of the UML-DOP profile with a UML profile diagram is shown in Figure 11. The diagram contains the stereotype classes with tagged values and the extended UML metaclasses. Each stereotype must extend one or more metaclasses.

5 Evaluation

We evaluate the completeness of our UML-DOP profile by applying the profile to a medium-sized, representative ABS model. We chose the “Adaptive Information System for Charity Organizations” (AISCO), a software system that helps charity

Table 1. UML-DOP profile, mapping of DOP elements to UML stereotypes

Element Name	**UML Base Class**	**Stereotype Name**
Module	Package	«module»
Export Module	Dependency	«export»
Import Module	Dependency	«import»
Delta Module	Package	«delta»
Delta Parameter	Property	«deltaParam»
Module Modifier	Association Association Association Class Interface	«adds» «removes» «modifies» «modifiedClass» «modifiedInterface»
Modifier	Operation; Property	«adds» «removes» «modifies»
Feature	Component	«feature»
Optional	Dependency	«optional»
Mandatory	Dependency	«mandatory»
Require	Dependency	«require»
Exclude	Dependency	«exclude»
Product	Component	«product»
When	Dependency	«when»
After	Dependency	«after»

organizations to publish their activities and to generate financial reports. The case study was selected for the following reasons:

1. it is a typical product line development that shows a variety of aspects, including multiplicity constraints, cross feature relations, etc.;
2. it is not an academic toy study, but a medium-sized project driven by real-world requirements;
3. it has been developed on the basis of ABS and DOP and we have full access to the source code.

The AISCO system is designed to help charity organizations in reporting their work. These organizations share many characteristics: basically, they all obtain funds from donors and then distribute them to beneficiaries. However, their activities vary widely and include building public facilities, child support, education matters, to name just a few. After obtaining funds and running their program, charities are obliged to report their programs' results and the financial flow. Hence, the system is designed to record income, expenses, donors, and beneficiaries. The report can be generated automatically or created manually.

Based on four different charity organization in Indonesia, we analyzed the requirements for AISCO. We modeled the feature for monitoring a charity

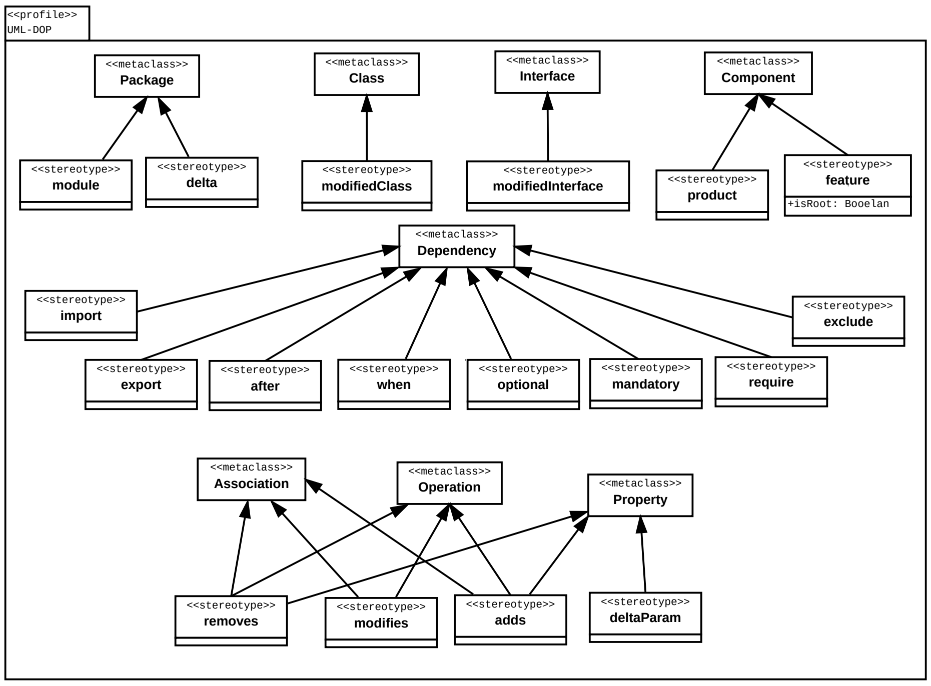


Fig. 11. UML-DOP profile as a UML profile diagram

program (`ProgramData`) as mandatory, because all organizations need this feature. The organizations have programs with different scheduling behavior, which could be periodic (weekly, monthly, or yearly), eventual (a program that can be run anytime), or continuous (a program that runs gradually). All these require different system behavior. It is possible that an organization runs several programs of a different nature. There is an organization that only has periodical program, another organization has eventual program, and a big organization can have these three kinds of program. We modeled the different scheduling types as optional child features of `ProgramData`. Further, mandatory features are a publication system and financial reporting—these are needed by all organizations. Other optional features include objective/target data that specifies the recipients information, and donation reporting. The full feature model of AISCO, which describes its commonality and variability in detail is shown in Figure 12.

The AISCO case study is fully modeled in the ABS language and uses DOP for the implementation of features.[4] There are six core ABS modules and sixteen delta modules. In addition, there is the feature model represented in μTVL, the product line configuration, and a product selection.

We made sure that all ABS elements of the AISCO case study can be modeled with our UML-DOP profile to evaluate completeness of the profile. As outlined

[4] `https://gitlab.com/IS4Charity/aisco-abs.git`

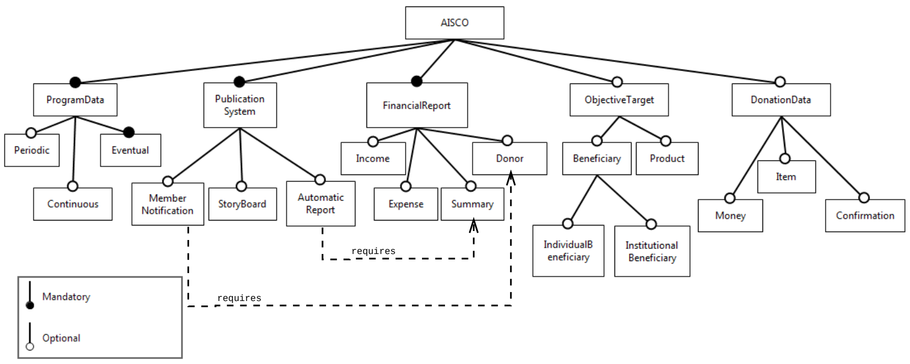


Fig. 12. Feature diagram of the AISCO case study

in Section 4, core ABS modules are mapped to UML packages containing a UML class diagram, ABS delta modules are mapped to UML *packages* that contain UML classes with a modifier stereotype, features are mapped to UML *components* with stereotype «`feature`», and products are mapped to UML *components* with stereotype «`product`».

A fragment of the resulting UML diagram is displayed in Figure 13. On top is the `MProgram` module, containing the ABS core `Program` interface and `ProgramImpl` class. There is also the `ProgramData` feature with three optional child features, `Periodic`, `Eventual`, and `Continuous`. The latter feature is implemented by applying the delta `DContinuous`, which adds two new attributes to the `ProgramImpl` class. It also adds a method and modifies method `setProgram`. Product selection is illustrated with the product `PKPU` that has nine features. Their implementation is not shown.

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.

6 Related Work

There exists a UML Profile for Software Product Lines [22], which is defined based on the UML 2.0 metamodel. The profile is defined on UML class and sequence diagrams. The authors propose eight stereotypes and one tagged value related to optionality and variations, such as «`optional`», «`variation`», and «`variant`». There exist UML profiles designed for feature diagrams [15,20]. These are intended for integrating feature diagrams with UML models. Possompès et al. [15] define a UML profile for the cardinality-based feature diagram of Czarnecki-Eisenecker [5]. The profile of Vranic and Snirc [20] is based on their own feature metamodel and implemented in IBM Rational Software Architect to support linking the feature

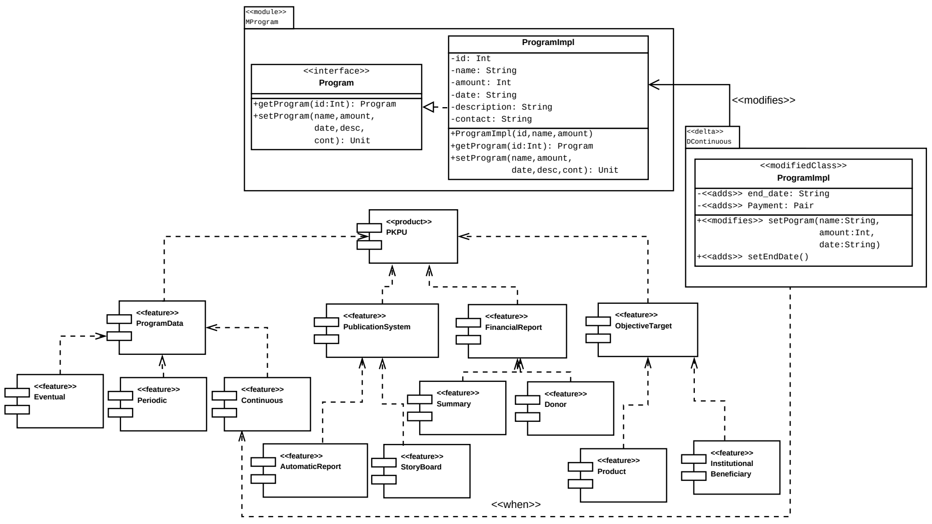


Fig. 13. AISCO UML diagram with UML-DOP profile

diagrams with UML artifacts. Both propose stereotypes related to features and products that are more complete than the UML profile of Ziadi [22].

The main difference of all mentioned approaches to our work is that we use DOP as a unifying framework for modeling product variability. This approach has a number of important advantages:

1. a very slim UML profile is sufficient that has a systematic and easy-to-remember nomenclature;
2. the implementation of features (in the form of a product line configuration) is part of the static view and hence traceable;
3. not merely features and their implementation is represented, but also product selection: as a consequence, a static view of the whole SPL development can be represented in the form of UML diagrams.

Extended UML class diagrams are also used to evaluate a DOP-based framework for automated product derivation in [18]. There core classes and deltas are represented in a single diagram which can be problematic from the point of readability and does not scale to multiple deltas. No UML profile is defined, instead an ad-hoc notation that decorates UML model elements with symbols $+$, $*$, $-$ as well as a color schema is used. Features are not represented.

While we have refrained from defining a dedicated UML profile for feature diagrams, this would be easily possible and it is orthogonal to the UML-DOP profile.

UML profiles can be defined based on the syntax of an underlying OO language. The OMG published a UML profile based on Java syntax [12]. We took advantage of this possibility to create a UML-DOP profile based on ABS and DeltaJ syntax.

7 Conclusion and Future Work

We defined a UML-DOP profile that permits to represent software product line variability using the popular UML notation. Because it is based on delta-oriented programming, the profile is very lightweight: by reusing stereotyped UML elements all DOP 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 careful reading and valuable feedback on earlier drafts of this manuscript. Thanks to members of RSE Lab Fasilkom UI for their support in identifying the case study.

References

1. *The ABS Language Specification*, ABS version 1.2.0 edition, Apr. 2013. `http://tools.hats-project.eu/download/absrefmanual.pdf`.
2. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, 2014.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
4. L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *Proc. of Object-Oriented Programming Languages and Systems (OOPS), Track of ACM SAC*, 2010.
5. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Software Product Lines*, pages 162–164. Springer-Verlag, 2004.
6. 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 — A comparative study. *Sci. Comput. Program.*, 93:65–85, 2014.

7. L. Fuentes-Fernandez and A. Vallecillo-Moreno. An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, V:6–13, 2004.
8. R. Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In M. Bonsangue, F. de Boer, E. Giachino, and R. Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer-Verlag, 2013.
9. 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 International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.
10. K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute, 1990.
11. 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. ACM Press, 2014.
12. Object Management Group. *Metamodel and UML Profile for Java and EJB Specification*, 2004. available at `http://www.omg.org`.
13. Object Management Group. *Unified Modelling Language: Superstructure*, 2004. available at `http://www.omg.org`.
14. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering.* Springer, 2005.
15. 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. Knowledge Systems Institute, 2011.
16. C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.
17. I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *International Software Product Line Conference*, SPLC '10, pages 77–91. Springer, 2010.
18. I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A model-based framework for automated product derivation. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE)*, 2009.
19. P. Schobbens, P. Heymans, and J. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering, 14th IEEE International Conference*, pages 139–148, 2006.
20. V. Vranic and J. Snirc. Integrating feature modeling into UML. In *Net Object Days/Grid Service Engineering and Management*, Lecture Notes in Informatics, pages 3–15. Gesellschaft für Informatik, 2006.
21. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.
22. T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML profile for software product lines. In *Software Product-Family Engineering*, volume 3014 of *LNCS*, pages 129–139. Springer, 2004.