Delta-oriented multi software product lines

This is the author's manuscript

Original Citation:

Availability:
This version is available http://hdl.handle.net/2318/150060 since 2015-12-22T19:54:18Z

Publisher:
ACM - Association for Computing Machinery

Published version:
DOI:10.1145/2648511.2648536

Terms of use:
Open Access
Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)
ABSTRACT

Modern software systems outgrow the scope of traditional software product lines (SPLs) resulting in multi software product lines (MSPLs) with many interconnected subsystem versions and variants. Delta-oriented programming (DOP) is a flexible, modular approach for implementing SPLs, but DOP so far does not allow the realization of MSPLs. In this paper, we extend DOP to support MSPL development and provide the first holistic modeling approach for MSPLs that spans problem, solution and configuration space. The main concept is the extension of DOP with the possibility to import other SPLs or MSPLs into a new MSPL. By expressing constraints amongst the imported SPLs, a common configuration and product generation is enabled.

Categories and Subject Descriptors
D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Design, Languages

Keywords
Java, Delta-Oriented Programming, Multi Software Product Line

1. INTRODUCTION

Modern variant-rich software systems can be managed by software product line (SPL) engineering techniques. In this paper, we assume an SPL along the lines of Czarnecki and Eisenecker [6] that consists of a problem space variability model defining the set of possible product variants in terms of product features, a solution space code base with the reusable code artifacts and a configuration space which connects problem and solution space and defines how to derive product variants from the code artifacts based on valid problem space feature selection. However, today’s software systems out-grow the scope of SPLs. MSPLs are a union of several SPLs with a common variability model [9]. MSPLs are prevalent in today’s large-scale systems, such as in industrial automation [8], data bases systems [14]. MSPLs are beneficial in large-scale system development as they allow reuse of existing SPLs in a new context, reduce complexity by decomposition of a large SPL and enable the distribution of work over several development teams.

Most existing MSPL modeling approaches have considered only the problem space variability model explicitly and treated the problem space code artifacts and the configuration mechanism as a black-box [7, 15, 8]. However, in order to provide reuse within MSPLs also on the code level, we need a MSPL modeling approach that spans problem space (variability model), solution space (code artifacts) and configuration space (product generation). An according modeling approach for MSPLs has to satisfy the following requirements: The MSPL has to import other SPLs. The variability model of the MSPL has to combine the variability models of the composed SPLs. In the variability model of the MSPL, it should be possible to define additional features and restrict the variability of the composed SPLs, e.g., by pre-selection of features of the composed SPLs. Additional dependencies between the variability models of the composed SPLs may have to be introduced in order to define valid combined variants. In the problem space, the MSPL should combine the code base of the composed SPLs, add own code or modify imported code artifacts. In the configuration space, it has to be defined how code artifacts, newly introduced and modified imported, are assembled for a particular product configuration from the MSPL’s variability model. In this paper, we propose MULTIDELTAJ in order to holistically represent delta-oriented MSPLs. MULTIDELTAJ extends DELTAJ [4] by linguistic constructs for problem space and solution space specification of MSPLs to support composing and re-defining imported SPLs. Furthermore, it provides a well-defined process for product derivation in MSPLs. The MULTIDELTAJ approach is hierarchical such that an MSPL can be composed from SPLs and other, already defined, MULTIDELTAJ MSPLs. Additionally, it is modular such that an MULTIDELTAJ MSPL can be defined by only accessing the MULTIDELTAJ specifications of the imported SPLs or MSPLs.

2. DELTA-ORIENTED SPLS

In this section, we recall the main concepts of DELTAJ [4], which is the archetypal language for delta-oriented programming of SPLs. As an example, we consider a product line of JAVA programs implementing a family of text editors called the Text SPL. Figure 1 shows the feature model of the Text SPL. The products in the Text SPL are described by the features Editor, Persist, CandP, Format
and Color. The features Editor and Persist are mandatory, all other features are optional. The feature Editor corresponds to the base functionality of an editor; feature Persist adds storage and loading functionality; feature CandP adds the functionality for Copy and Paste history; feature Color adds syntax highlighting; feature Format adds formatting; and feature Scalable adds the ability to scale text.

A delta-oriented SPL consists of a code base and a declaration. The code base consists of a set of delta modules, which are containers of modifications to a JAVA program. The modifications may add, remove or modify interfaces or classes. Modifying an interface means to change its super interfaces, or to add or to remove method signatures. Modifying a class means to change its super interfaces, or to add or to remove methods or to modify methods. The modification of a method can either replace the method body by another implementation, or wrap the existing method using the original construct (similar to the Super() call in AHEAD [3]). The original construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the original construct, other statements can be introduced. The code base for the Text SPL, shown in figure 1, consists of 6 delta modules corresponding to the respective features: DEditor, DPersist, DCandP, DFormat, and DScalable. Listing 1 shows a fragment of the code base for the Text SPL. The delta module DEditor introduces the class EditorListener and its method update, and the delta module DPersist modifies the method update by adding the persistence of the parameter into a file and then calling the original version of the method.

The product-line declaration creates the connection to the product line variability specified in terms of product features. Listing 2 shows the product-line declaration for the Text SPL. An application condition is attached to each delta module in a when clause specifying for which feature configurations the delta module has to be applied. The possible application orders of the delta modules are described by a total order on a partitioning of the set of delta modules, which (in Listing 2) is expressed by an ordered list of delta module sets enclosed by { ... }. The ordering captures semantic requires-relations that are necessary for the applicability of the delta modules. If a part contains a delta module that adds or removes a class, no other delta module in the same part may add, remove or modify the same class, and the modifications of the same class in different delta modules of the same partition must be disjoint. Deltas in the same part can be applied in any order, since the result will be the same, but the order of the parts is fixed. In order to obtain a product for a particular feature configuration, those modifications specified in the delta modules with valid application conditions are applied incrementally to the empty product.

3. DELTA-ORIENTED MSPLS

In this section, we introduce MultiDELTAJ, the extension of DELTAJ to support delta-oriented programming of MSPLs. A delta-oriented MSPL is a delta-oriented SPL that uses other SPLs by importing them. The import of an SPL allows restricting the variant space of the imported SPL by providing a pre-configuration of its features. The MSPL itself defines a set of features and binds the non-resolved features of the imported SPLs to newly declared features. A MultiDELTAJ MSPL may define delta modules (the same as in DELTAJ) which can add new classes/interfaces and remove/modify classes/interfaces introduced by the MSPL itself or made available from imported SPLs. The configuration of a MSPL determines the configuration of the used sub-SPLs by resolving all un-resolved imported SPL features via their binding. Generating a product of the declared MSPL means generating the selected product variants of the imported SPLs and modifying them as specified by the MSPL declaration.

3.1 Delta-oriented MSPL declaration

A MultiDELTAJ MSPL declaration has the syntax illustrated in Figure 2 (the extensions w.r.t. non-multi SPL declaration are highlighted in grey), where square brackets “[ ]” and “{ ” indicate optional elements; <fname> ∈ MSPL names; <fname> ∈ feature names; <fselection> denotes either <fname> (the selection of the feature <fname>) or ! <fname> (the deselection of feature <fname>); <fname> ∈ a propositional formula over feature names; <iname> ∈ JAVA interface names; <cname> ∈ JAVA class names; <usname> ∈ used sub-MSPL names; <dname> ∈ delta module names.

An MSPL declaration starts with the keyword mspl to declare the name of the defined MSPL (that we call the top-MSPL). The uses clause defines sub-(M)SPLs by importing an (M)SPL. A sub-(M)SPL definition declares the name <usname> of the sub-(M)SPL (this name will be visible when the declared MSPL will be imported—cf. the third example below) and provides (after the symbol “=”)

1. The name <fname> of the imported (M)SPL, possibly followed by a list (<fselection>, ..., <fselection>) that restricts its configurations by resolving (i.e., selecting or deselecting) some of its features. The name of a resolved feature is no longer accessible.

2. An optional with clause that further restricts the configurations of the imported (M)SPL by specifying a propositional formula <fformula> over the non-resolved features of the imported (M)SPL that has to be conjoined to the formula

---

**Listing 1:** A fragment of the code base of the Text SPL

**Listing 2:** DELTAJ declaration of the Text SPL
that describes the feature configuration specified by configurations clause in the declaration of the imported (M)SPL.

3. An optional when clause that specifies that the sub-(M)SPL is used only when the given propositional formula over the features of the top-MSPL holds.

The features clause introduces the features of the top-MSPL—each feature of the top-MSPL may be bound to (i.e., unified with) some (non-resolved) feature of a sub-(M)SPL and each non-resolved feature of a sub-(M)SPL must be bound to exactly one feature of the top-MSPL. Binding a feature \( f \) of the top-MSPL to a non-resolved feature \( f' \) of a sub-(M)SPL means that, if a product variant of the sub-(M)SPL is included in a variant of the declared MSPL (i.e., if the condition specified by the when clause is satisfied) then the selection or de-selection of the feature \( f \) propagates to the feature \( f' \). The configurations clause introduces the formula describing the valid configurations of the top-MSPL—it mentions only the features declared by the features clause.

The interfaces (or classes) clause renames interfaces (or classes) defined in some imported sub-(M)SPL—these new names can be used in the code base of the declared MSPL and will be visible when the declared top-MSPL will be imported by another MSPL.

The splist clause provides the ability to:

- Unity sub-(M)SPLs of MSPLs that are imported into the top-MSPL. If two imported MSPLs use the same sub-(M)SPL, in this way, it can be ensured that only one instance of the sub-(M)SPL is included in a product generated from the top-MSPL.

Examples. As first example, Figure 3 and Listing 3 show the feature diagram and the MSPL declaration for the AddressBook SPL. The features represent data fields for a list of contacts and are straightforward, except the Note feature. For the implementation of the feature Note, an editor from the Text SPL is required indicated by the dashed arrow. In Listing 3, the uses clause specifies that a product of the sub-SPL text, defined in terms of the Text SPL, is included when feature Note is selected. The sub-SPL text is defined by resolving the mandatory features of the imported Text SPL by selecting them. The other non-resolved features of the Text SPL are bound to newly declared features of the AddressBook SPL.

To use the editor, new names for the interface FileManagerService and the class Editor from the Text SPL are defined. Otherwise, the interface and class would not be visible to use in the code base of AddressBook SPL.

The code base for the AddressBook SPL consists of 5 new delta modules: Delta DNames is mandatory and introduces the contact list. Delta DNickname adds additional names, Delta Phone adds fields for phone numbers, delta DEMail fields for email addresses and DNote a field for other notes.

As second example, Figure 4 and Listing 4 show the feature diagram and the MSPL declaration for the Mail SPL. The Mail SPL handles the receiving and sending of mails and uses the Text SPL to write the mails in a text editor. The uses clause imports the Text SPL, if the feature Write of the Mail SPL is selected. The sub-SPL text is defined by selecting the mandatory features of the imported Text SPL. Then, the features of the declared Mail SPL are defined and the non-resolved features of the imported SPL are bound to features of the declared MSPL. The configurations clause captures the constraints of the feature diagram in Figure 4.

Figure 3: Feature diagram of the Addressbook MSPL

Listing 3: MultiDELTAJ declaration of the AddressBook SPL

MSPL—an error is reported if this is not possible.

- Introduce a new name for the sub-(M)SPLs of MSPLs that are imported into the top-MSPL—when the top-MSPL will be imported, these names (of the sub-(M)SPLs of the imported (M)SPL) will be visible in the splist clause of the importing MSPL. This will allow to make further unifications.

The delta modules defined in the top-MSPL may add new classes/interfaces and remove/modify classes/interfaces that are either introduced in other delta modules defined in the top-MSPL or made available via the interfaces or classes clause.
name the class EditorLister and class Editor of the Text SPL to MailEditorLister and MailEditor. The interface FileManagerService keeps the name it has in the Text SPL as it does not have to be modified in the declared MSPL. The code base for the Mail SPL consists of 4 delta modules: Delta DReceive is mandatory and collects new mail from a mail server. Delta DWrite introduces a mail editor and, thus, needs the Text SPL. Delta DSend allows sending mails to the recipients. Delta DReply creates a new draft of an mail with the contents of the last mail and its recipients.

As third example, Figure 5 and Listing 5 show the feature diagram and the MSPL declaration of the MailClient MSPL. The MailClient MSPL manages mail accounts and stores mail which are send and received by the Mail SPL, Optionally, it can store the mail addresses from incoming and outgoing mails with the feature Addresses which uses the AddressBook MSPL. Since both imported MSPLs use the Text SPL, we want to import only one instance of the Text SPL into a variant of the MailClient MSPL. Therefore, the features clause unifies the features of the Text SPL of both imported MSPLs. It would be also possible to use two separate instances of the Text SPL. In this case, we could bind any of the non-resolved features from both imported Text SPLs to separate features of the MailClient MSPL. The fact that the MailClient includes only one instance of the Text SPL (imported by both MSPLs Mail and AddressBook) is specified by the $sp$ clause. Both sub-MSPLs mail text and addresses text are unified as sub-SPL text. We also rename interfaces and classes from the imported SPLs, in order to make them visible in the code base of the MailClient MSPL. The code base for the MailClient MSPL consists of 5 delta modules in 4 partitions: delta MailClient builds the basic client. Delta DAddresses adds the address book. Deltas IMAP and POP3 are the supported protocols. Delta DReply adds the reply functionality, if the feature Reply from the Mail MSPL is selected, which is called MailReply.

3.2 Product generation

We outline the MultiDeltaJ product generation procedure for the above examples. The AddressBook MSPL (cf. Listing 3) uses the sub-SPL text which is defined in terms of the Text SPL when feature Note is selected. The product with features Names, Note and TextCandP is generated by performing the following steps:

1. Add the code of the product of the Text SPL with features Editor, Persist and CandP (cf. Listing 2), where the name of every class and interface is changed from the original names to those of the used SPL when appending to the original names the name of the used SPL i.e., the string "$$text$$."

2. Because of the interfaces clause, rename the interface FileManagerService$Text$ to FileManagerService (i.e., restore its original name).

3. Because of the classes clause, rename the class Editor$Text$ to Editor (i.e., restore its original name).

4. Apply the selected delta modules of the AddressBook MSPL, i.e., DNames and DNote—the delta module DNote may contain occurrences of the interface name FileManagerService and of the class name Editor introduced by the interfaces and classes clause.

The Mail MSPL (cf. Listing 4) uses the sub-SPL text which is defined in terms of the Text SPL when feature Write is selected. The product with features Send, Write and TextCandP is generated by performing steps similar to those for the AddressBook MSPL.

The MailClient MSPL (cf. Listing 5) uses the sub-SPL mail which is defined in terms of the Mail MSPL and (when feature Addresses is selected) the sub-SPL addresses which is defined in terms of the AddressBook MSPL. The product with features Protocols, IMAP, Addresses, Mail and MailTextCandP is generated by performing steps similar to those for the Mail and AddressBook MSPLs, together with additional steps for dealing with the $sp$ clause, i.e., the following steps are performed:

1. Add the code of the product of the Mail SPL with features Send, Receive, Write and TextCandP (cf. Listing 4), where every class and interface is renamed by appending to its name the string "$$mail$$."

2. Add the code of the product of the AddressBook SPL with features Names, Note and TextCandP (cf. Listing 3), where every class and interface is renamed to avoid name clashes.

3. Perform the renamings specified by the interfaces clause.
5. CONCLUSION

In this paper, we have presented MultiDeltaJ, a programming language for delta-oriented MSPLs allowing to obtain MSPLs by fine-grained reuse of delta-oriented (M)SPLs. MultiDeltaJ is the first approach for a holistic modeling of MSPLs covering problem, solution and configuration space. An implementation of MultiDeltaJ based on the existing implementation of DeltaJ is currently in progress.

6. REFERENCES