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)
Delta-Oriented Multi Software Product Lines∗

Ferruccio Damiani
Università di Torino, Italy
damiani@di.unito.it

Iva Schaefer
TU Braunschweig, Germany
i.schaefer@tu-bs.de

Tim Winkelmann
TU Braunschweig, Germany
t.winkelmann@tu-bs.de

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.3.3 [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], database 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 blackbox [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 SLP. Figure 1 shows the feature model of the Text SLP. The products in the Text SLP are described by the features Editor, Persist, CandP, Format

∗Work partially supported by MIUR (proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPLC ’14, September 15 - 19 2014, Florence, Italy.
Copyright 2014 ACM This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in 978-1-4503-2740-4/14/09..$15.00. http://dx.doi.org/10.1145/2648511.2648536.
**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; <fname> denotes either <fname> (the selection of the feature <fname>) or ! <fname> (the deselection of feature <fname>); <fname> denotes a propositional formula over feature names; <fname> ∈ **JAVA** interface names; <fname> ∈ **JAVA** class names; <fname> ∈ used sub-**MSPL** names; <fname> ∈ 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-(**MSPL**s) by importing an (**MSPL**). A sub-(**MSPL**) definition declares the name <fname> of the sub-(**MSPL**) (this name will be visible when the declared **MSPL** will be imported—cf. the third example below) and provides (after the symbol “=”: ) a definition body consisting of three parts:

1. The name <fname> of the imported (**MSPL**), possibly followed by a list (<fname>, ...<fname>) 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 (**MSPL**) by specifying a propositional formula <formula> over the non-resolved features of the imported (**MSPL**) that has to be conjoined to the formula
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 deselection 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 deltas clause provides the ability to:
- 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 sps 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.

Examples. As first example, Figure 3 and Listing 3 show the feature diagram and the MSPL declaration for the AddressBook MSPL. The features represent data fields for a list of contacts and the editor, new names for the interface FileManagerService and the class AddressBook. 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 EditorListener and class Editor of the Text SPL to MailEditorListener 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 mails 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 SPL. 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 sps 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 DClient 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 to avoid name clashes by 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 clauses.

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 sps 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.
4. Perform the renamings specified by the **classes** clause.
5. Because of the **spls** clause: (i) The names of the interfaces and classes coming from the instance of in the **Text SPL** imported by the **Mail** MSPL are made equal to the names of the interfaces and classes coming from the instance of in the **Text SPL** imported by the **AddressBook** MSPL—an error is reported if any of those interfaces and classes has been modified during the generation of the product of the **Mail** MSPL or of the product of the **AddressBook** MSPL; and (ii) Only one copy of the code of the two (now identical) products of the **Text SPL** is kept.
6. Apply the selected delta modules of the **MailClient** MSPL, i.e., **DClient**, **DAddresses** and **DIMAP**.

4. RELATED WORK

Research on modeling MSPLs can categorized according to the spaces (problem, solution and/or configuration space [6]) which it covers. For **problem space variability** modeling, tools originally developed for SPLs are fully or partially capable of modeling MSPLs, such as TVL by Classen et al. [5] or FAMILIAR by Acher et al. [2]. In the CVM framework [1], different feature models can be connected via configuration links expressing requirements for feature selection between several feature models. Velvet is a textual variability modeling language designed by Rosenmüller et al. [12] for multi-dimensional variability modeling, explicitly intended for MSPL variability. All these approaches are located only in the problem and configuration space and do not consider the solution space. To modularly capture **solution space variability**, Kästner et al. [10] introduce a variability-aware module system where variability within modules is represented by presence conditions. This approach can only be applied in an MSPL context when connected to problem space variability modeling. The EASY-Producer [7] is a tool for multi-dimensional variability modeling and configuration of MSPLs. It uses a decision-oriented problem space variability modeling approach. For solution space variability, instantiators are used as wrappers to support the variability realization mechanisms of the imported SPLs. In this way, the EASY-Producer allows multi-product line configuration on the solution space level, however, dependencies on the solution space level are not captured as the imported SPLs are treated as black-boxes. Several commercial tools, such as pure::variants (www.pure-systems.com) or GEARS (www.biglever.com), have support for multi-product line configuration in a similar manner [13]. In the DOPLER tool suite [15, 8], product line bundles (PLIBs) integrate several SPLs into one MSPL from a tool-oriented perspective. PLIBs go beyond pure problem space variability by including configuration, but do not explicitly cover the solution space artifacts. Keunecke et al. [11] propose feature packs for software eco systems consisting of a problem space variability model and the corresponding variable solution space artifacts. Feature packs modularize problem and solution space variability for software ecosystems, hence, no unified explicit variability model, as for MSPLs in MultiDELT AJ, is supported. To summarize, the state-of-the-art in MSPL modeling/programming concentrates either on the problem space, on the solution space, or integrates only problem space and configuration space.

5. CONCLUSION

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

6. REFERENCES