

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

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

Ina 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.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

*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>.

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

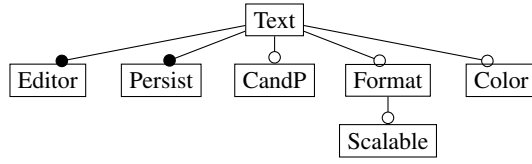


Figure 1: Feature diagram of the *Text* SPL

```
delta DEditor {
  adds class EditorListener {
    Editor editor;
    void update(String s) { editor.show(s); }
  }
  ... }
delta DPersist {
  modifies class EditorListener {
    adds File file;
    modify void update(String s) { file.store(s); original(s); }
  }
  ... }
... }
```

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

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 class or implemented interfaces, to add or to remove fields or 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, DColor, 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

```
spl Text
features Editor, Persist, CandP, Format, Color, Scalable
configurations Editor & Persist & (Scalable -> Format)
deltas
{ DEditor when Editor }
{ DPersist when Persist }
{ DCandP when CandP, DColor when Color }
{ DFormat when Format }
{ DScalable when Scalable }
```

Listing 2: DELTAJ declaration of the *Text* SPL

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; $\langle \text{sname} \rangle \in \text{MSPL names}$; $\langle \text{fname} \rangle \in \text{feature names}$; $\langle \text{fselection} \rangle$ denotes either $\langle \text{fname} \rangle$ (the selection of the feature $\langle \text{fname} \rangle$) or ! $\langle \text{fname} \rangle$ (the deselection of feature $\langle \text{fname} \rangle$); $\langle \text{fformula} \rangle$ denotes a propositional formula over feature names; $\langle \text{iname} \rangle \in \text{JAVA interface names}$, $\langle \text{cname} \rangle \in \text{JAVA class names}$; $\langle \text{usname} \rangle \in \text{used sub-MSPL names}$; $\langle \text{dname} \rangle \in \text{delta module names}$.

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 $\langle \text{usname} \rangle$ 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 “=”) a definition body consisting of three parts:

1. The name $\langle \text{sname} \rangle$ of the imported (M)SPL, possibly followed by a list ($\langle \text{fselection} \rangle, \dots, \langle \text{fselection} \rangle$) 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 $\langle \text{fformula} \rangle$ over the non-resolved features of the imported (M)SPL that has to be conjoined to the formula

```

m spl <sname>
[uses <uname> = <sname>[(<fselection>, ..., <fselection>)]
  [with<fformula>][when<fformula>]
  ...
  <uname> = <sname>[(<fselection>, ..., <fselection>)]
  [with<fformula>][when<fformula>]
]

features <fname> [= <uname>.<fname>... = <uname>.<fname>],
...
<fname> [= <uname>.<fname>... = <uname>.<fname>]
configurations <fformula>
[interfaces <iname> = <uname>.<iname>,
...
<iname> = <uname>.<iname>]
]
[classes <cname> = <uname>.<cname>,
...
<cname> = <uname>.<cname>]
]
[spls [<uname>].<uname> = <uname>.<uname>
[= <uname>.<uname>... = <uname>.<uname>],
...
[<uname>].<uname> = <uname>.<uname>
[<uname>.<uname>... = <uname>.<uname>]
]
deltas
{ <dname> [when<fformula>],
...
<dname> [when<fformula>]}
...
{ <dname> [when<fformula>],
...
<dname> [when<fformula>]}

```

Figure 2: Syntax of MSPL declarations

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 **spls** clause provides the ability to:

- Unify 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-

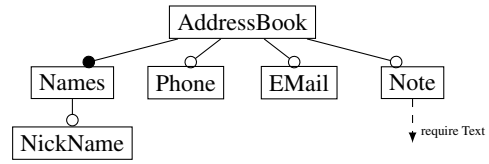


Figure 3: Feature diagram of the Addressbook MSPL

```

mspl AddressBook
uses text = Text(Editor,Persists) when Note
features Names, Phone, EMail, NickName, Note,
  TextCandP=text.CandP, TextFormat=text.Format, TextColor=text.Color,
  TextScalable=text.Scalable
configurations Names
interfaces FileManagerService = text.FileManagerService
classes Editor = text.Editor
deltas
{ DNames when Names}
{ DNickname when NickName}
{ DPhone when Phone}
{ DEmail when EMail}
{ DNote when Note}

```

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 **spls** 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 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* MSPL. 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* MSPL. 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 *DPhone* 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* MSPL. The *Mail* MSPL 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* MSPL 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* MSPL 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. We re-

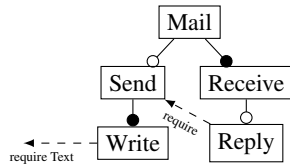


Figure 4: Feature diagram of the Mail MSPL

```
mspl Mail
uses text = Text (Editor,Persist) when Write
features Send, Receive, Write, Reply,
  TextCandP=text.CandP, TextFormat=text.Format, TextColor=text.Color,
  TextScalable=text.Scalable
configurations Receive & (Send <-> Write) & (Reply -> Write)
interfaces FileManagerService = text.FileManagerService
classes MailEditor=text.Editor,
  MailEditorListener=text.EditorListener
deltas
{DReceive when Receive}
{DSend when Send}
{DWrite when Write}
{DReply when Reply}
```

Listing 4: MULTIDELTAJ declaration of the Mail MSPL

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 a 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* 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 **spls** 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

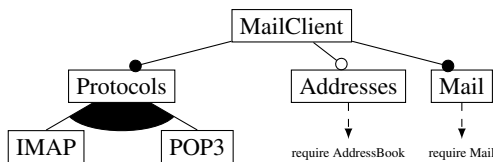


Figure 5: Feature diagram of the MailClient MSPL

```
mspl MailClient
uses mail = Mail (Send, Write, Receive) when Mail,
  addresses = AddressBook (Names, EMail) when Addresses
features Protocols, IMAP, POP3, Addresses, Mail,
  MailReply=mail.Reply, AddressesNickName=addresses.NickName,
  AddressesPhone=addresses.Phone,AddressesNote=addresses.Note,
  MailTextCandP=mail.TextCandP=addresses.TextCandP,
  MailTextFormat=mail.TextFormat=addresses.TextFormat,
  MailTextColor=mail.TextColor=addresses.TextColor,
  MailTextScalable=mail.TextScalable=addresses.TextScalable
configurations Mail & Protocols & (IMAP | POP3)
interfaces ContactManagerService = addresses.ContactManagerService
classes Contact = addresses.Contact
  MailAccount = mail.MailAccount
  Mail = mail.Mail
spls text = mail.text = addresses.text
deltas
{DClient when Mail}
{DAddresses when Addresses}
{DIMAP when IMAP, DPOP3 when POP3}
{DReply when MailReply}
```

Listing 5: MULTIDELTAJ declaration of the the MailClient MSPL

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 **spls** 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 be 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 MULTIDELTAJ, 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 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

- [1] A. Abele, Y. Papadopoulos, D. Servat, M. Törnigren, and M. Weber. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *VaMoS'10*, pages 101–105, 2010.
- [2] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Sci. Comput. Program.*, 78(6):657–681, 2013.
- [3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [4] L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50:77–122, 2013.
- [5] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability*, 76(12):1130–1143, 2011.
- [6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [7] S. El-Sharkawy, C. Kröher, and K. Schmid. Supporting heterogeneous compositional multi software product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11*, pages 25:1–25:4. ACM, 2011.
- [8] G. Holl, C. Elsner, P. Grünbacher, and M. Vierhauser. An infrastructure for the life cycle management of multi product lines. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 1742–1749. ACM, 2013.
- [9] G. Holl, P. Grünbacher, and R. Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Inf. Soft. Technol.*, 54:828–852, 2012.
- [10] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '12*, pages 773–792. ACM, 2012.
- [11] M. Keunecke, H. Brummermann, and K. Schmid. The feature pack approach: Systematically managing implementations in software ecosystems. In *VaMoS '14*, pages 20:1–20:7. ACM, 2013.
- [12] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-dimensional variability modeling. In *VaMoS'11*, pages 11–20. ACM, 2011.
- [13] K. Schmid and E. S. de Almeida. Product line engineering. *IEEE Software*, 30(4):24–30, 2013.
- [14] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated analysis of dependent feature models. In *VaMoS'13*, pages 9:1–9:5. ACM, 2013.
- [15] M. Vierhauser, G. Holl, R. Rabiser, P. Grünbacher, M. Lehofer, and U. Sturmer. A deployment infrastructure for product line models and tools. In *SPLC'11*, pages 287–294. IEEE, 2011.