A Delta Oriented Approach to the Evolution and Reconciliation of Enterprise Software Products Lines

Gleydson Lima[2,3], Jadson Santos[1,3], Uirá Kulesza[1], Daniel Alencar[1] and Sergio Vianna Fialho[2]
[1]*Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, Natal, RN, Brazil*
[2]*Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal, RN, Brazil*
[3]*Informatics Superintendence (SINFO), Federal University of Rio Grande do Norte, Natal, RN, Brazil*

Keywords: Software Product Line Engineering, Software Product Evolution, Software Reconciliation.

Abstract: Over the last years, software product line engineering has been applied and adopted by different companies. Existing software product line approaches promote the development of a centralized infrastructure of core assets that addresses the common features and provides variation points to the integration of the variable features of the SPL. In the context of distributed development of enterprise information systems, there are several scenarios where the adoption of these centralized approaches is not enough to accommodate the several requests for the integration of new features and maintenance of existing ones. In such scenarios, the SPL engineering team needs to fork the SPL core assets in order to address the customer needs and due to the marked pressure. In this paper, we propose a delta-oriented approach that promotes the reconciliation of software product lines that are independently evolved. Our approach allows: (i) the automated detection of feature conflicts of the SPLs independently evolved; and (ii) the resolution and merge of such feature conflicts.

1 INTRODUCTION

Over the last years, software product line engineering has been applied and adopted by different companies (Product Line - Hall of Fame, 2005). Existing software product line approaches (Weiss and Lai, 1999); (Clements and Northrop, 2001); (Czarnecki and Eisenecker, 2000); (Greenfield and Short, 2005) promote – during domain engineering – the development of a centralized infrastructure of core assets that addresses the common features and provides variation points to the integration of the variable features of the SPL. In application engineering, these reusable assets are reused and customized in order to produce and generate specific applications (products). The evolution of the SPL involves to apply changes directly to its core assets, which implement the common features (commonalities) and respective variation points. This development strategy promotes a better way to manage variabilities and contribute to facilitate the evolution of the independent products in terms of changes applied to a centralized infrastructure that addresses all of them. If on one hand this SPL development strategy is very useful and has been used in many companies in the software industry, on the other hand there are many existing scenarios where it does help to cope with the great demand for changing requirements from different companies that are benefited by SPLs (Krueger, 2006); (Rubin et al, 2012); (Mende et al., 2009); (Ernst et al., 2010).

In the context of enterprise information systems, there are several scenarios where the adoption of existing SPL approaches is not enough to accommodate the requests for the integration of new features and maintenance of existing ones. In such scenarios, the SPL engineering team needs to fork the core assets in order to address the customer needs and due to the marked pressure. In addition, each different version created during the SPL forking need to be maintained by different teams, thus bringing more difficulties to the SPL evolution activities. Recent research works (Rubin et al., 2012) (Nunes et al., 2010) have proposed preliminary approaches for dealing with this challenge. However, there are no existing concrete automated approaches that support the development and reconciliation of SPLs independently evolved from the same reusable code assets. In particular, none of

Lima G., Santos J., Kulesza U., Alencar D. and Vianna Fialho S..
A Delta Oriented Approach to the Evolution and Reconciliation of Enterprise Software Products Lines.
DOI: 10.5220/0004453102550263
In *Proceedings of the 15th International Conference on Enterprise Information Systems* (ICEIS-2013), pages 255-263
ISBN: 978-989-8565-59-4
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)
255

the existing research work reflects about this problem in the context of enterprise web information systems.

In this paper, we propose a delta-oriented approach that promotes the reconciliation of software product lines that are independently evolved from the same reusable code assets. Our approach promotes: (i) the automated detection of feature conflicts of the SPLs independently evolved; and (ii) the resolution and merge of such feature conflicts in terms of changes to be applied to the code assets with the aim of reconciling the SPLs. We have implemented an initial version of our approach for the context of SPLs of enterprise web information systems. It has been developed using model-driven and code analysis tools available in the Eclipse platform. Our work also describes preliminary results from the application of our approach to the context of a product line of an enterprise academic web information system developed in our institution. This SPL has been independently evolved by other 15 federal universities in Brazil.

The remainder of this paper is organized as follows. Section 2 details the challenges of reconciling SPLs independently evolved by presenting an example. Section 3 gives an overview of our approach and describes the technologies used in its implementation. Section 4 illustrates the application of our approach to the reconciliation of enterprise information product lines. Section 5 discusses related work. Finally, Section 6 concludes the paper and indicates directions for future work.

2 PROBLEM STATEMENT

In the section, we describe a real scenario of development related to the challenge of reconciliation of a same SPL evolved independently by different institutions. The Informatics Superintendence (SINFO) from Federal University of Rio Grande do Norte (UFRN) has been facing this challenge during the development of enterprise information systems. The SINFO/UFRN is currently responsible for the development of different enterprise information systems (SINFO, 2013). The three main ones are: (i) SIGAA – enterprise information system responsible for the management of academic activities; (ii) SIPAC – enterprise information system responsible for the management of the finance, property and contracts of the university departments; and (iii) SIGRH - enterprise information system responsible for management of human resources. Given their quality, these systems have been licensed by different federal institutions of Brazil, including 15 federal universities, and the Federal Justice, Police, Culture and Planning Brazilian Departments.

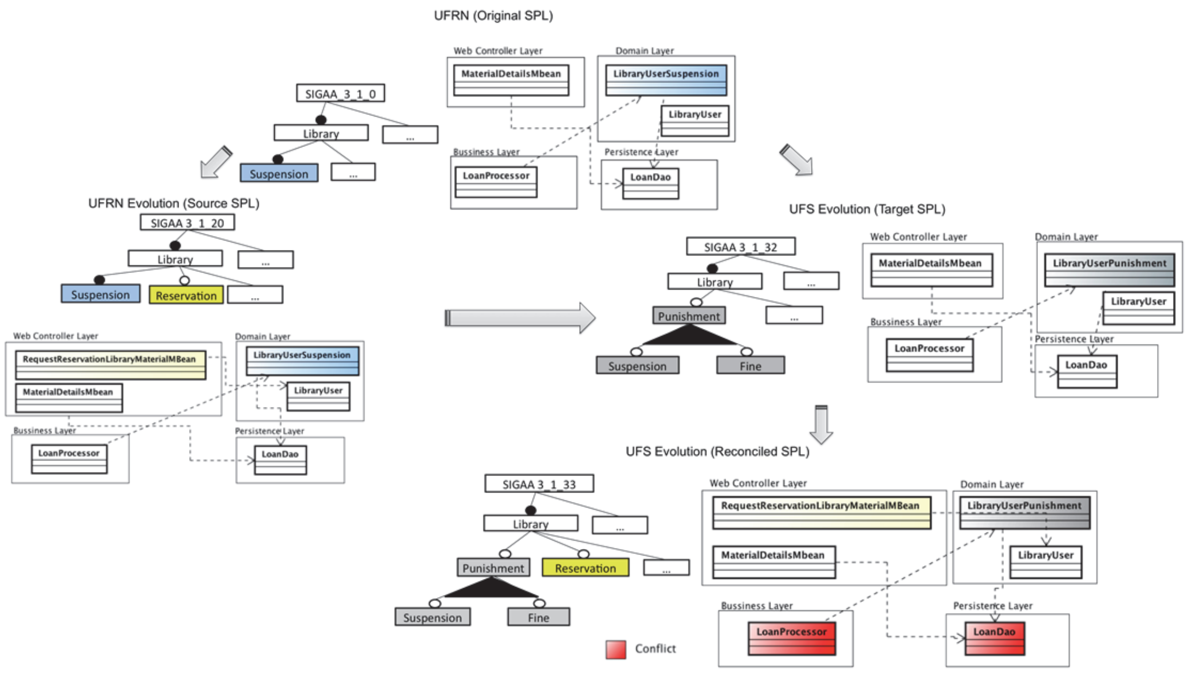


Figure 1: Feature model and code assets evolution.

The deployment of these enterprise information systems from SINFO/UFRN at different institutions demanded their adaptation to address new requirements and features, which implement specific business rules and needs of our partners. In order to deal with this scenario, each enterprise information system has been adapted (Santos et al., 2012); (Sena et al., 2012) to become a SPL that: (i) provides a common infrastructure of code assets that can be reused by different institutions; and (ii) defines variation points that can be extended to address specific needs. However, due to the dynamic environment of those institutions, specially regarding the occurrence of new demanding requirements, there is always a need to adapt the infrastructure to accommodate adequately all these new requirements. The existing variation points are not enough to address all the demanding requirements in many cases. In such scenario, the solution has been to create a separate branch of the SPL to each institution, which can freely extend and modify the implementation of the core and variation point assets. Similar strategies have been reported by other existing research work (Rubin, 2012); (Nunes, 2010). This approach is also known as clone-and-own.

Figure 1 illustrates the development and evolution scenario of the SIGAA – the enterprise information system responsible for the management of academic activities of the UFRN. First, the initial version of SIGAA/UFRN SPL – called the *Original SPL* – is made available to other partners, such as Federal University of Sergipe (UFS). Every institution then creates a fork/copy of the original SPL – named the *Target SPL*. After that, they can extend or modify the classes that implement the SPL core and variation point assets, or even the default variable features made available by UFRN. At the same time, the UFRN team also independently evolve the SPL thus generating the *Source SPL* (Figure 1).

Figure 1 also details an example of how UFRN and UFS have independently evolved one version of SIGAA. It presents the feature models of both universities that reflect these changes. It shows that UFRN has introduced a new feature Reservation as part of the Library feature. On the other hand, the UFS has modified the Suspension feature to include the Punishment and Fine features. The Suspension feature becomes a subfeature of Punishment feature.

After the parallel evolution of the SPLs, the UFS can request from UFRN the integration of the new Reservation feature to its SPL code assets. The reconciliation of the two independently evolved SPLs involves the resolution and merge of new source code developed for each of them. In particular, in this scenario, there is a need to integrate the features from *Source SPL* (UFRN) to the *Target SPL* (UFS). Different and several classes are introduced and modified during the evolution of the SPLs. The reconciliation of them involves identifying conflicting features in terms of introduction, modification and deletion of existing code assets. Figure 1 shows, for example, that the Reservation and Punishment features have demanded the creation of new classes (`RequestReservationLibraryMaterialMBean`, `LibraryUserPunishment`) and the modification of existing ones (`LoanProcessor`).

Our research problem is related to the incapacity and inefficiency of existing software engineering techniques and tools to promote the seamless integration and reconciliation of the evolved features from the source SPL to the target SPL. Existing configuration management systems only provide code merge low-level mechanisms that give all the responsibility to the engineers to perform a safe integration and reconciliation of the SPL features. On the other hand, existing SPL approaches are founded on the development of a centralized infrastructure. They do not provide advanced techniques or tools to deal with this problem.

3 EVOLVING AND RECONCILING SOFTWARE PRODUCT LINES

This section presents our delta-oriented approach to support the evolution and reconciliation of software product lines. Section 3.1 gives an overview of the approach by describing its main components. Section 3.2 illustrates the main technologies used in its implementation.

3.1 Approach Overview

The main aim of our approach is to promote the reconciliation of SPLs that are independently evolved. In order to address this aim, our approach allows: (i) the automated detection of feature conflicts of the SPLs evolved independently; and (ii) the resolution and merge of such feature evolution conflicts. Figure 2 shows an overview of the main components of our approach. Next we explain and detail each one of them.

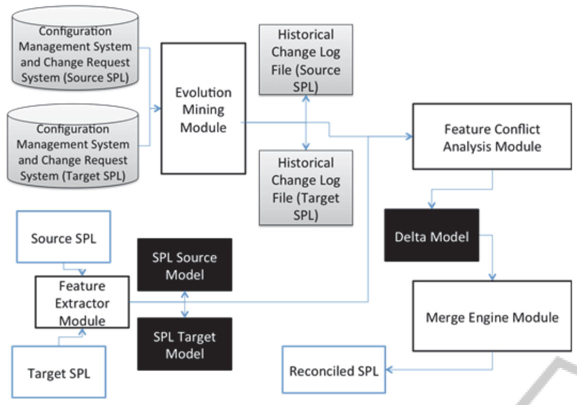

Configuration Management System and Change Request System (Source SPL)
Configuration Management System and Change Request System (Target SPL)
Evolution Mining Module
Historical Change Log File (Source SPL)
Historical Change Log File (Target SPL)
Feature Conflict Analysis Module
Delta Model
Source SPL
SPL Source Model
Feature Extractor Module
SPL Target Model
Target SPL
Merge Engine Module
Reconciled SPL

Figure 2: Approach overview.

3.1.1 Feature Extractor Module

The first step is executed using the feature extractor module. The main goal of this module is to extract the features and code of the SPLs that are being reconciled and integrated – named target and source SPLs. This information will be used to support the comparison and automatic detection of conflicts between the SPLs evolved independently. The following strategies are used to extract the information of interest during this approach step: (i) the SPL features can be obtained from existing variability management tools, such as CIDE (Kästner, 2013), GenArch (Cirilo, 2008); (Cirilo, 2012), and pure::variants (pure::variants, 2013); and (ii) the information regarding code assets are extracted as abstract syntax trees of each different asset using existing code parsing tools. This module produces as output models that maintain the information regarding the features, code assets, and respective mapping between features and code assets for each investigated SPL, which is also called configuration knowledge (Czarnecki and Eisenecker, 2000).

3.1.2 Evolution Mining Module

The second step involves the mining of the evolution of features and code assets from the SPLs target and source. It is supported by the evolution mining module, which interacts with existing change request and configuration management systems, in order to extract information related to the evolution of the features and code assets from each SPL independently evolved. Thus, this module extracts all the changes applied to the features and code assets of the SPLs. This information will also be useful to allow the automatic detection of feature conflicts between the source and target SPLs during their evolution. It produces as output historical change logs about the evolution of features and code assets for each SPL analyzed.

3.1.3 Feature Conflict Analysis Module

This module is responsible for the automatic detection of conflicts of features that evolve independently along the SPLs target and source. It uses as input the information generated by the feature extractor and evolution mining modules, which are, respectively, the source and target models, and the historical change logs.

The module executes an algorithm that compares the source and target models searching for changes applied to the features and code assets of the SPLs. It produces as output a delta model containing: (i) the new features created in the source SPL that can be integrated to the target; (ii) the modified features in the source SPL compared to the target; and (iii) the removed features in the source SPL compared to the target.

3.1.4 Merge Engine Module

Once identified the feature conflicts from the integration between the source and target SPLs, our approach is prepared to analyze these conflicts in order to promote the merge of them.

During this last step, the merge engine module first asks to the engineer, which feature changes from the source SPL he/she is interested to integrate to the target SPL. After that, the module analyzes the dependencies between the features and code assets in order to verify if the merge of the selected features from the source SPL can be applied automatically, semi-automatically or manually to the target SPL.

After this analysis, the tool can recommend and apply specific merge actions to integrate feature changes from the source to the target SPL. The merge actions are implemented according to the existing variability implementation technologies involved in the modularization of the SPL. Finally, the merge engine module is also responsible to indicate which specific features (or use cases) should be retested in order to verify that everything is working well after the SPL integration.

3.2 Approach Implementation

Our approach has been implemented as an Eclipse plugin using existing model-driven and code manipulation technologies available in this platform. Figure 3 shows the infrastructure of our tool by illustrating the technologies used in its implementation. It was developed based on the Squid (Vianna et al, 2012) – an extensible infrastructure for analyzing software product line implementations.

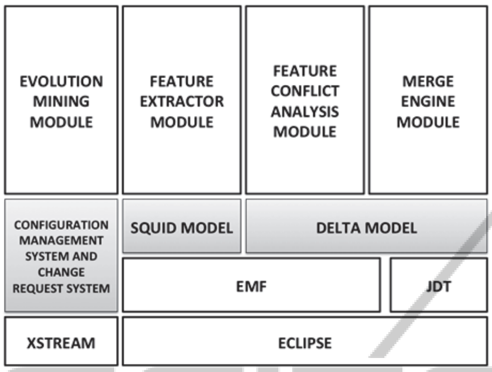


Figure 3: Tool infrastructure.

The Feature Extractor module is implemented as an extension of the Squid, which is responsible for parsing the code assets and annotations embedded in these code assets that indicate the implementation of specific variable features. This parsing functionality is implemented using the Java Developing Tooling (JDT) API. As a result of the parsing, the module produces Squid models as output, which maintain information regarding the features, code assets and mappings between these elements. Every Squid model is implemented and manipulated using Eclipse Modeling Framework (EMF) technology.

The current implementation of the Evolution Mining module integrates with an in-house change request system – called iProject – developed at SINFO/UFRN, and the Subversion configuration management system. The module obtains information regarding: (i) the evolution of every code asset from Subversion; and (ii) the change feature request from iProject.

The Feature Conflict Analysis module manipulates as input two Squid models and the historical change logs (XML files), in order to produce the delta model that indicates the changes applied to the source SPL that are not part of the target SPL. The delta model is also manipulated and implemented using the EMF plugin. Finally, the Merge Engine module is currently implemented to manipulate the abstract syntax tree (AST) of the SPL code assets using the JDT API. Different changes and refactorings can be applied to the source code of the target SPL, depending on the recommended merge actions of this module.

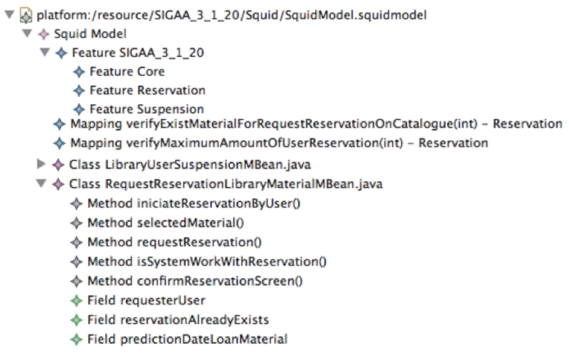
4 APPROACH IN ACTION

This section describes how our approach is used in practice by illustrating its application to the SIGAA software product line. Next sections describe how the different approach modules can be applied to the resolution and merge of conflicts from SIGAA.

4.1 Extracting Feature and Code Assets from the SPLS

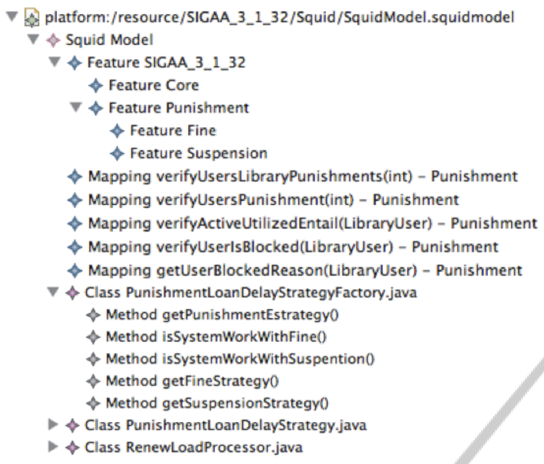
The Feature Extractor module is responsible for processing the feature annotations and for generating the Squid target and source models as output that contain their respective features, code assets, and mapping between them. Our current implementation uses the feature annotations from GenArch product derivation tool (Cirilo, 2008); (Cirilo, 2012).

Figures 4 and 5 show a partial view of the extracted models for the SIGAA SPL. It considers the evolution of the Library feature (Section 2) for the UFRN and UFS universities, respectively. It shows how the sub-features of Library have been evolved. Figure 4 shows, for example, that the UFRN SIGAA SPL has introduced the Reservation feature and associated code assets, such as `verifyExistingMaterialForReservation()` and `verifyMaximumAmountOfReservation()` methods. On the other hand, Figure 5 illustrates that the UFS SIGAA SPL has included the Punishment feature and associated code assets, such as the `PunishmentLoanDelayStrategyFactory` and `PunishmentLoanDelayStrategy` classes. It is interesting to notice that the Squid models also include information about the mappings between features and code assets. For example, the `verifyUserLibraryPunishment()` method is mapped to the Punishment feature (Figure 5).



```
▼ platform:/resource/SIGAA_3_1_20/Squid/SquidModel.squidmodel
  ▼ Squid Model
    ▼ Feature SIGAA_3_1_20
        Feature Core
        Feature Reservation
        Feature Suspension
      Mapping verifyExistMaterialForRequestReservationOnCatalogue(int) – Reservation
      Mapping verifyMaximumAmountOfUserReservation(int) – Reservation
    ▶ Class LibraryUserSuspensionMBean.java
    ▼ Class RequestReservationLibraryMaterialMBean.java
        Method iniciateReservationByUser()
        Method selectedMaterial()
        Method requestReservation()
        Method isSystemWorkWithReservation()
        Method confirmReservationScreen()
        Field requesterUser
        Field reservationAlreadyExists
        Field predictionDateLoanMaterial
```

Figure 4: Extracted model of the source SPL (UFRN).

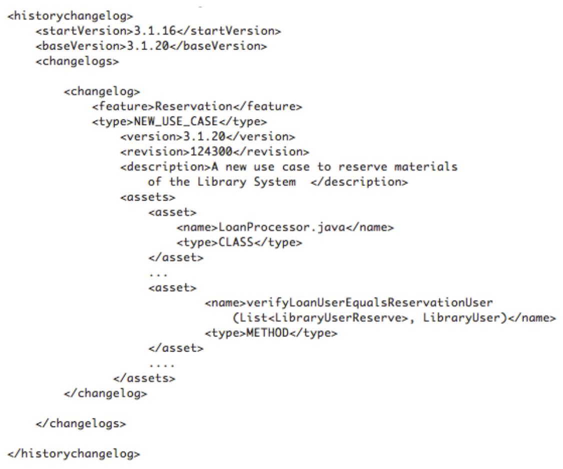


```
▼ platform:/resource/SIGAA_3_1_32/Squid/SquidModel.squidmodel
  ▼ Squid Model
    ▼ Feature SIGAA_3_1_32
        Feature Core
      ▼ Feature Punishment
          Feature Fine
          Feature Suspension
      Mapping verifyUsersLibraryPunishments(int) – Punishment
      Mapping verifyUsersPunishment(int) – Punishment
      Mapping verifyActiveUtilizedEntail(LibraryUser) – Punishment
      Mapping verifyUserIsBlocked(LibraryUser) – Punishment
      Mapping getUserBlockedReason(LibraryUser) – Punishment
    ▼ Class PunishmentLoanDelayStrategyFactory.java
        Method getPunishmentEstrategy()
        Method isSystemWorkWithFine()
        Method isSystemWorkWithSuspention()
        Method getFineStrategy()
        Method getSuspensionStrategy()
    ▶ Class PunishmentLoanDelayStrategy.java
    ▶ Class RenewLoadProcessor.java
```

Figure 5: Extracted model of the target SPL (UFS).

4.2 Feature Evolution Mining

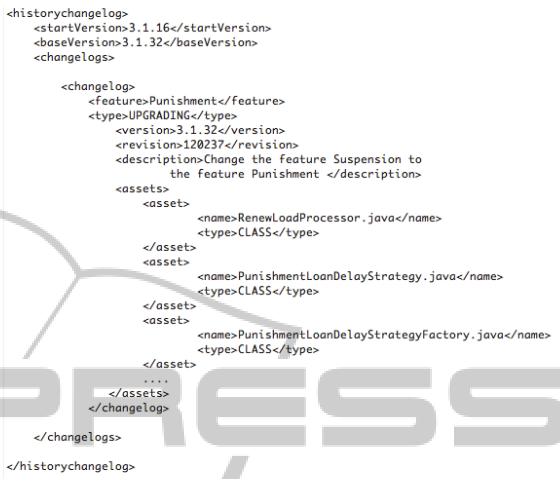
The second step consists on mining the evolution of the features and code assets from the source and target SPLs. The historical data of the change request and configuration management systems from each SPL are mined in order to generate as output historical change logs for the source and target SPLs.



```
<historychangelog>
    <startVersion>3.1.16</startVersion>
    <baseVersion>3.1.20</baseVersion>
    <changelogs>

        <changelog>
            <feature>Reservation</feature>
            <type>NEW_USE_CASE</type>
                <version>3.1.20</version>
                <revision>124300</revision>
                <description>A new use case to reserve materials
                    of the Library System  </description>
                <assets>
                    <asset>
                        <name>LoanProcessor.java</name>
                        <type>CLASS</type>
                    </asset>
                    ...
                    <asset>
                            <name>verifyLoanUserEqualsReservationUser
                                (List<LibraryUserReserve>, LibraryUser)</name>
                            <type>METHOD</type>
                    </asset>
                    ....
                </assets>
        </changelog>

    </changelogs>

</historychangelog>
```

Figure 6: Source SPL historical evolution file.

Figures 6 and 7 show a partial view of the historical change log files produced as output for the UFRN and UFS SPLs, respectively. As you can see, every change log file contains a root tag called *historychangelog* that has several *changelogs* siblings. Each *changelog* specifies: (i) the feature; (ii) the nature of the feature (UPGRADING, NEW_USE_CASE, BUG_FIX); (iii) the version and revision submitted to the Subversion system; (iv) a brief description of the change; and (v) the code assets that have been modified. For instance, Figure 6 shows that the `LoanProcessor` class and the `verifyLoanUserEqualsReservationUser()` method of the Reservation feature were modified in the revision 124300 of the Subversion system.

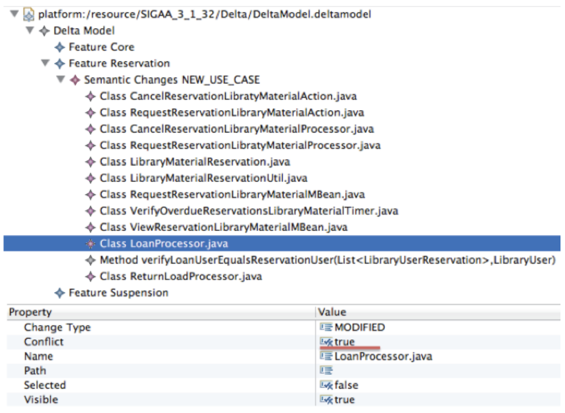


```
<historychangelog>
    <startVersion>3.1.16</startVersion>
    <baseVersion>3.1.32</baseVersion>
    <changelogs>

        <changelog>
            <feature>Punishment</feature>
            <type>UPGRADING</type>
                <version>3.1.32</version>
                <revision>120237</revision>
                <description>Change the feature Suspension to
                        the feature Punishment </description>
                <assets>
                    <asset>
                            <name>RenewLoadProcessor.java</name>
                            <type>CLASS</type>
                    </asset>
                    <asset>
                            <name>PunishmentLoanDelayStrategy.java</name>
                            <type>CLASS</type>
                    </asset>
                    <asset>
                            <name>PunishmentLoanDelayStrategyFactory.java</name>
                            <type>CLASS</type>
                    </asset>
                    ....
               </assets>
            </changelog>

    </changelogs>

</historychangelog>
```

Figure 7: Target SPL historical evolution file.

4.3 Detecting Feature Conflicts between Source and Target SPLs

The third step of our approach consists on automatically generating the delta model, which shows the feature conflicts from the evolution of source and target SPLs. In order to generate this model, the *Feature Conflict Analysis* module receives as input the information generated in the previous steps for the source and target SPLs, which are: (i) the Squid models; and (ii) historical change logs. The module processes this information to identify the feature conflicts in terms of changes applied to the code assets and stores this information in the delta model.



```
▼ platform:/resource/SIGAA_3_1_32/Delta/DeltaModel.deltamodel
  ▼ Delta Model
      Feature Core
    ▼ Feature Reservation
      ▼ Semantic Changes NEW_USE_CASE
          Class CancelReservationLibratyMaterialAction.java
          Class RequestReservationLibraryMaterialAction.java
          Class CancelReservationLibraryMaterialProcessor.java
          Class RequestReservationLibratyMaterialProcessor.java
          Class LibraryMaterialReservation.java
          Class LibraryMaterialReservationUtil.java
          Class RequestReservationLibraryMaterialMBean.java
          Class VerifyOverdueReservationsLibraryMaterialTimer.java
          Class ViewReservationLibraryMaterialMBean.java
          Class LoanProcessor.java
          Method verifyLoanUserEqualsReservationUser(List<LibraryUserReservation>,LibraryUser)
          Class ReturnLoadProcessor.java
      Feature Suspension
```

Property	Value
Change Type	MODIFIED
Conflict	true
Name	LoanProcessor.java
Path	
Selected	false
Visible	true

Figure 8: Delta model.

Figure 8 illustrates the resulted delta model of the UFRN source and UFS target SPLs. The delta model shows if there are conflicts in terms of code assets between the features independently created or modified in the SPLs. For every feature presented, the model indicates the kind of change that has been applied to the code assets from the source to the target SPL. Figure 8 shows, for example, that the `LoanProcessor` class has been modified in the source SPL, when compared to the target. The Properties View indicates the conflict in this class for the Reservation feature that represents a NEW_USE_CASE change.

4.4 Merging Source and Target SPLs

In the final step of our approach, the *Merge Engine* module is used and the conflicts identified between the source and target SPLs are prepared to be handled. The engineer informs which feature changes from the source SPL he/she is interested to apply to the target SPL. The dependencies between the features and code assets are then analyzed to verify which strategy – automatic, semi-automatic or manual - of merge can be applied.

Figure 9 shows examples of the different merge strategies executed by our tool in the context of the Reservation feature. The `RequestReservationLibraryMaterialMBean` class is a new asset created in the UFRN source SPL that need to be added to the UFS target SPL. This class only has a dependency to the `LibraryUser` class of the original SPL, which has not changed during the evolution of both SPLs. Thus, the *Merge Engine* module can automatically move this class to the target SPL, after calculating and recognizing that it has no dependency to any existing class.

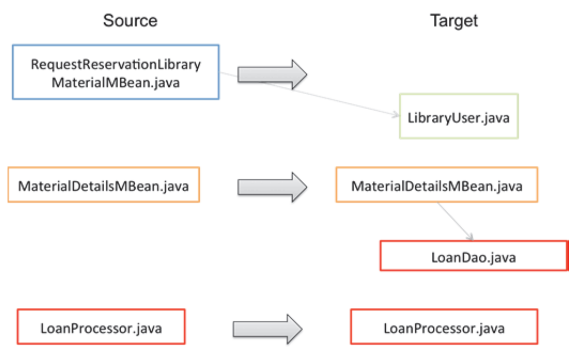

Source
Target
RequestReservationLibrary MaterialMBean.java
LibraryUser.java
MaterialDetailsMBean.java
MaterialDetailsMBean.java
LoanDao.java
LoanProcessor.java
LoanProcessor.java

Figure 9: Examples of merge strategies.

Figure 9 also exhibits the `MaterialDetailsMBean` class that has been modified in the source SPL to address specific needs of the Reservation feature. The modifications of this class only involve the introduction of new independent methods. On the other hand, the `MaterialDetailsMBean` class has not been modified during the evolution of the target SPL. But there is a dependent class – called `LoanDAO` – that was modified in the target SPL. Because of that, the *Merge Engine* can merge the new methods of the `MaterialDetailsMBean` class from the source into the target SPL, but it cannot guarantee that the integration is going to work adequately, due to the changes applied to the `LoanDAO` class. In those scenarios, the module uses a semi-automatic merge strategy, where it indicates explicitly which classes will be merged to the target SPLs, but it also notifies the engineers about the specific needs to inspect or re-test the merged classes due to changes in the dependent classes, such as the `LoanDAO` class, in our example. We are currently extending the implementation of this merge strategy to allow the automatic recommendation of automated testing classes to be re-executed over the merged classes that have modified dependent classes.

Finally, the last merge strategy currently available in our *Merge Engine* module is the manual one. It only indicates the code assets conflicts to the software engineers during the integration of specific features. The engineers are then responsible for deciding the best strategy to integrate them based on the information provided regarding the changes applied to the existing features. Figure 9 shows that the `LoanProcessor` class has been modified in both source and target SPLs in order to address the Reservation and Punishment features, respectively. Due to the large amount of conflicting changes over similar methods of the `LoanProcessor` class, our *Merge Engine* module uses the manual merge strategy to highlight to the engineers the changes applied to the code assets that are associated to each new feature (Reservation and Punishment). The information is then used to help the developers during the manual merge of the features.

5 RELATED WORK

(Rubin et al., 2012) propose the improvement of the efficiency of forking practices in the context of SPLs, while mitigating their disadvantages. The work defines the Product Line Change Set Dependency Model (PL-CDM), which captures the necessary information required for managing forked product variants. PL-CDM contains information about the entire product line, such as: its products, features of these products, and relationships between the features. This model contributes to keep information about the SPL that will aid the developer to manage the fork product variants. The authors also demonstrate their approach for the management and evolution of forked product variants using a real example. However, the authors have not developed a concrete implementation of PL-CDM that automates the forking variants management. In contrast, our work proposes a delta-oriented approach that helps the merging of SPLs independently developed. In this paper, we have also presented the tool support that automates our approach.

(Nunes et al., 2010) propose the analysis of historical evolution of family members in order to classify the implementation elements according to their variability nature. The work proposes history-sensitive heuristics for feature recovering in the code of degenerated program families. The historical evolution analysis considers: (i) the history of each member of the program family, called horizontal history; and (ii) all the family members, called vertical history. Through the usage of such heuristics, the authors verify how features change considering the vertical and horizontal perspectives and classify them. Although the authors deal with the problem of SPL evolution by identifying how each feature has evolved, their research work has not proposed concrete solutions to repair the feature degeneration in SPLs.

(Ferreira et al., 2012) propose the implementation and evaluation of four testing based approaches and their implementations for checking SPL refinement. Their work considers that a SPL is safely evolved, when it addresses at least the same products of the previous version. The first testing approach – called *All Product Pairs* – checks all products generated by the SPL after the evolution against all products before evolution. It verifies if the SPL continues generating at least all products generated before evolution. However, how this strategy is very onerous, they have proposed and assessed three other approaches – called *All Products, Impacted Products and Impacted Classes* – which are optimization of the first one that have lower precision, but it improves the execution performance. Their approach only deals with the evolution of the same SPL. In our work, we focus on the reconciliation of SPLs independently evolved after they are forked from the same initial version. The testing approaches proposed by (Ferreira, 2012) can be used to verify if the final reconciled target SPL is a safe evolution of the target SPL. This is an interesting research work that we are planning to consider in the future.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a delta-oriented approach that provides support for the reconciliation of SPLs independently evolved by different teams. Our approach allows: (i) the automated detection of feature conflicts in terms of code assets of the SPLs evolved independently; and (ii) the resolution and merge of such feature evolution conflicts. The preliminary tests and results of our current implementation show the potential of our approach to deal with such the SPL reconciliation challenge during the evolution of SPLs.

We are currently refining its implementation to apply it to a case study of large-scale reconciliation scenarios using the enterprise information SPLs from SINFO/UFRN, independently evolved by several federal Brazilian institutions. In particular, new extensions are being developed to support: (i) the automatic and semi-automatic merge of XML template documents, such as Java Server Faces (JSF) pages – that are usually used to implement web pages; (ii) to automatically identify which manual or automated testing cases needs to be re-executed to verify the behavior preservation of the reconciled features.

ACKNOWLEDGEMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) - CNPq under grants 573964/2008-4 and CNPQ 560256/2010-8.

REFERENCES

Cirilo, E., Kulesza, U., Lucena, C. J. P., 2008. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. *Journal of Universal Computer Science*, vol. 14, no. 8.

Cirilo, E., Nunes, I., Kulesza, U., Lucena, C. J. P., 2012. Automating the Product Derivation Process of Multi-Agent Systems Product Lines. *Journal of Systems and Software (JSS)*, pp 258-276, vol. 85, number 2, February 2012 .

Clements, P. and L. Northrop. Software Product Lines: Practices and Patterns. *Addison-Wesley Professional*, 2001.

Greenfield, J., and K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. *John Wiley and Sons*, 2005.

Czarnecki, K. and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. *Addison-Wesley*, 2000.

Ernst, N. A., Easterbrook, S. M. and Mylopoulos, J. Code Forking in Open-Source Software: a Requirements Perspective. *CoRR*, abs/1004.2889, 2010.

Ferreira, F., et al. 2012. Making Software Product Line Evolution Safer. *6th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2013)*, Natal, Brazil, IEEE Computer.

Kästner, C. CIDE: Virtual Separation of Concerns retrieved at http://wwwiti.cs.uni-magdeburg.de/ iti_db/research/cide/, February 2013.

Krueger, C. W. "New Methods in Software Product Line Development." *SPLC. 2006*, IEEE , 95-102.

Mende, T., Koschke, R. and Beckwermert, F. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *Journal of Software Maintenance and Evolution: Research and Practice*. 21(2):143–169. 2009.

Nunes, C. et al. 2010. History-Sensitive Heuristics for Recovery of Features in Code of Evolving Program Families. *Proceedings of the Software Product Line Conference (SPLC 2012)*, September 2–7, Salvador, Brazil.

Product Line - Hall of Fame. 2005. Product Line - Hall of Fame. Product Line - Hall of Fame. [Online] September 2005. [Cited: 14 January 2013.] http://splc.net/fame.html.

Pure::Variants, retrieved at
http://www.pure-systems.com/, February 2013.

Rubin, J. et al. 2012. Managing Forked Product Variants. *Proceedings of the Software Product Line Conference (SPLC 2012)*. Salvador. Brazil.

Santos, J. et al. 2012. Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines. *Proceedings of 9th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP 2012)* (in Portuguese).

Sena, D. et al. 2012. Modularization of Variabilities from Web Information Systems Software Product Lines. *Proceedings of 9th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP 2012)* (in Portuguese).

SINFO. 2013. Informatics Superintendence of the Federal University of Rio Grande do Norte. Available at: http://www.info.ufrn.br/wikisistemas/doku.php

Vianna, A. et al. 2012. Squid: An Extensible Infrastructure for Analyzing Software Product Line Implementations. *SPLC'12-Workshops*. September 2–7, Salvador, Brazil.

Weiss, D. and C. Lai. Software Product-Line Engineering: A Family-Based Software Development Process. *Addison-Wesley Professional*, 1999.